The AAAI-17 Workshop on
What's Next for AI in Games?
WS-17-15

Rewards Structure in Games: Learning a Compact Representation for Action Space

Margot Lisa-Jing Yann, Yves Lespérance, Aijun An
York University
4700 Keele St., Toronto, Canada
{lisayan}@cse.yorku.ca

Abstract

Learning approximate payoff functions is important to understand the dynamics in multi-player interactions. In general repeat games, each player's payoff can be represented as a combination of all other players' action choices using normal forms, which grow exponentially as the number of action choices increases. Graphical games, however, provide a compact representation to specify the inter-relations where one player's action choice is influenced by its neighbourhood. In this paper, we present how to learn players' approximate payoff functions from normal-form representations, yet also learn a compact graphical game representation of the inter-relations among the players. In this normal form representation, we explore the structural connections of mutual influence between players' action choices in game playing. We formally describe the problem of learning a player influence network and give a novel reward structure-learning algorithm for multiagent graphical games, called the Multi-Descendent Regression Learning Structure Algorithm (MDRLSA). We evaluate MDRLSA on random graphical games generated in GAMUT. Experiments show that MDRLSA can efficiently identify the independence among players and extract the influence graph accurately. The running time of MDRLSA increases linearly with the number of strategy profiles of a game. Compared with state-of-the-art graphical game model learning methods, MDRLSA shows efficiency in terms of time and accuracy.

Introduction

Computer game playing is an interesting and challenging area that has important practical applications and also serves as a key AI testbed (Finnsson 2012; Silver et al. 2016; Genesereth and Love 2005). Game theory (Shoham and Leyton-Brown 2009) studies how agents' strategies affect game outcomes/rewards. In this paper, we focus on games from a game-theoretic perspective, but the techniques we develop should apply to actual game playing in the sense of computer games. In a game, each player is given a number of action choices. While playing, players explore their individual action space combined with other players' action choices. Depending on the goal of each player, this action-choosing process is non-stationary and dynamic. If we keep a complete and uncompressed representation of a game, as either the number of actions increases, or the number of the players increases, the number of combinations of action choices for a game grows exponentially. However, commonly, a player's action choice is influenced by a subset of all other players' choices. Thus, it is practical and efficient for a player to explore related players' action spaces, instead of doing a complete exhaustive search.

Using a compact representation to describe the inter-dynamics among all the players provides a critical strategy for global planning during the game. For example, in general repeat games, each player's payoff can be captured in normal forms. Normal forms are represented as all possible combinations of action choices with the payoff obtained for each player. Graphical games (Kearns, Littman, and Singh 2001), however, provide a compact graphical representation, which represents the inter-relations among the players (nodes) where a player's action choice is influenced by its connected neighbourhood (edges). This compact representation[1] is useful for abstract reasoning, and then utilize standard methods for general game playing.

The inspiration for this work comes from observation of rational agents' common behaviour choice preference. When a player chooses his action, he looks at other players' action choice and varies the choices of his own action, depending on other players' joint strategy played, and co-adapts to other players' choices. Rationally, a player chooses an action with a positive impact on his payoff to benefit from it co-adapting with another players, and avoids actions with a negative impact. The other players that do not have an impact on his current payoff (called "irrelevant choice") can be ignored. Our interest is to identify these irrelevant players through exploring players' payoff space, and to create a compact representation of player interconnection to eliminate them from the search space of an individual's action choice. Therefore, the ultimate objective is to learn the payoff functions for each player. We assume that we can represent the payoff function (also called "utility function") using linear model that combines all the action choices with associated coefficient parameters. Second, we determine the inter-connections among players based on the coefficient pa-

[1]In conclusion, we will discuss a "dual" representation: action-graph games (Bhat and Leyton-Brown 2004), where nodes are actions and the edges between nodes express context-specific inter-dependencies.

rameters. A graphical structure model is learned to compactly represent relationships in a given game. We discuss this in more details later in the paper.

Our contribution is: 1) to use regression methods to learn payoff functions efficiently; 2) use the payoff functions to identify independence among players and further generate a graphical game structure representation. To our knowledge, this is the first study to directly learn structures of graphical games from payoff functions induced using regression models for normal-form games. We compare our work with a state-of-art graphical game model learning method and show that our method performs better in terms of time and accuracy. We also show that the running time of our method increases linearly with respect to the number of strategy profiles of a game.

Related Work

Much multiagent systems (MAS) learning research has been performed from both machine learning and game theoretic perspectives. However, when characterizing a multiagent system as a multiple players game, little research on how to abstract structure among players' actions has been performed.

General game playing (GGP) (Genesereth and Love 2005) research focuses on designing artificial intelligence programs to play general strategic games successfully based solely on formal game descriptions, for example, computer Poker programs, Chess and GO. One bottleneck issue is the large size of both rewards and action search space. The search space increases exponentially as the number of actions increases, as well as the number of players increases. Much research has been done to address searching in reward/action space, such as branch-and-bound search, breadth first search and Monte Carlo tree search (Finnsson 2012). On the other hand, GGP drives research to playing a set of significantly different games, even new games unknown a priori. Therefore, the main challenge of GGP is that the agent/player needs to be general enough to learn the structure of any game, guide the search to relevant states of the search space, and be able to adapt to a wide variety of situations in the absence of game-dependent heuristics.

Game theory (Shoham and Leyton-Brown 2009) gives a theoretical framework for self-interested player reasoning, where it requires complete specification of players' payoff function. This game information is exploited to compute an equilibrium of optimal strategies for all players. How long does it take until rational agents converge to an equilibrium? The complexity of the problem of computing a mixed Nash equilibrium in a game can be high. There are games in which convergence to such an equilibrium takes prohibitively long. Traditionally, computational problems fall into two classes: those that have a polynomial-time algorithm, and those that are NP-hard. However, graphical games have been shown to substantially speed up game-theoretic equilibria computation or approximation, or other solution concepts (Daskalakis, Goldberg, and Papadimitriou 2006).

Research has been done in this field to reduce the space for presentation of normal form games. Using Bayesian networks learning methods, Duong et al. (2009) assumes the inter-connection as causal relations, and learns the graph structure using a branch-and-bound (BB) learning algorithm. However, as the size of the data set becomes large, the advantage of BB over the greedy algorithm dissipates, while its computation time increases considerably. Vorobeychik, Wellman, and Singh (2007) addressed approximation payoff-function learning in normal-form games. However, further difficulties are introduced by intractably large strategy sets. We seek to identify the full game with real-value strategies and payoff information available only in the form of data from a given sample of strategy profiles. We not only address payoff-function learning as a standard regression problem, but also provide for capturing structure in the multiagent environment. In this paper, we present how to learn a graphical games representation to describe the interrelations among the players from payoff given in a normal form presentation.

Preliminaries

In this paper, we focus on one-stage games, where each player chooses an action simultaneously and is then rewarded with a payoff, and the game ends. This single-shot game representation is quite general since strategies can be arbitrary histories of past play. For any multi-stage game (also called "extensive-form games"), it may be converted to a one-stage game with strategy sets for actions of all possible histories of play.

Notions

A generic normal-form game is formally expressed as $[I, (\boldsymbol{x}_i), \boldsymbol{y}_i(s)]$, where I refers to the set of players and $n_p = |I|$ is the number of players. $\boldsymbol{x}$ is the set of pure strategies (actions) available to player $i \in I$. The utility function, $\boldsymbol{y}_i(s) : x_1 \cdots x_{n_p} \to \mathbb{R}$ defines the payoff of player i when players jointly play $\boldsymbol{x} = (x_1, \cdots, x_{n_p})$, where each player's strategy (action) x_i is selected from its strategy set, X_i.

Definition 1. For every player i, $i \in I$, a strategy profile $\boldsymbol{x} = (x_1, \cdots, x_{n_p})$ is *a pure Nash Equilibrium* of game $[I, (\boldsymbol{x}_i), \boldsymbol{y}_i(s)]$, if for any available strategy, $x'_i \in X_i$,

$$y_i(x_i, x_{-i}) \geq y_i(x'_i, x_{-i}).$$

where, x_i is i's strategy, and the rest of the players' joint strategy is denoted as x_{-i}.

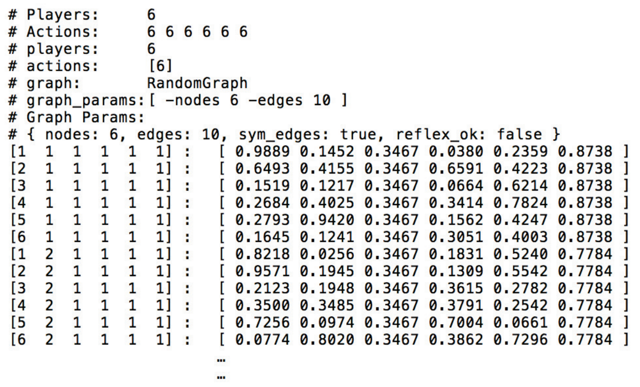
Graphical Games

Graphical games (Kearns, Littman, and Singh 2001) are a representation of multiplayer games to capture direct influence among players. A graphical game is described as an undirected graph G in which players are represented as vertices, and each edge identifies influence between two vertices. In natural settings, a player, represented as vertex v, has payoffs that are specified in terms of the action of vertex v and that of neighbours of v who have influence over vertex v. Each player's payoff is given by a matrix with all combinations of players' action choices using normal form representation. However, each player is influenced by its neighbours, which normally consist of some of the players in the game. In other words, a player's neighbour set is a subset of the complete player set. Rather than describing a game using the normal form (i.e. by way of a matrix), a graphical structure gives a direct and visual representation of the relationships among all the players. Graphical games are a suitable representation when sparse strong influences exist, whereas when there exists a large number of weak influences on each player, congestion games (Rosenthal 1973) are applicable.

In our research, we tackle the problem of learning a graphical game representation that captures the relationships among players from a normal-form game. We randomly generate multiplayer graphical games using GAMUT (Nudelman et al. 2004). GAMUT is a suite of game generators designed for testing game-theoretic algorithms.

Graphical Game Example

For example, Figure 1 describes a six player random graphical game. In this game, each player has a choice of actions represented from 1 to 6, and the total connections among players are represented as 10 reflexive edges (we assume mutual influential connections which exist among players, called "reflexive edges", simply called "edge" in this paper). Each edge is a randomly selected connection between two players which determines/influences the payoff received for each player [2]. In order to compare different games, we normalize each game's payoff between 0 and 1. A strategy set $[2, 2, 1, 1, 1, 1]$ represents all players' action choices at one stage of the game, which indicates that player 1 chooses action 2, while player 2 chooses action 2, and players 3, 4, 5 and 6 all choose action 1. These action combinations (also called a "joint strategy"), gives payoffs for player 1 to player 6 as follows respectively: $[0.95, 0.19, 0.34, 0.13, 0.55, 0.77]$. In Figure 1, the normal form representation of this 6-player game states the total number of 46656 (6^6) action profiles and the corresponding utilities for each player.



```
# Players:      6
# Actions:      6 6 6 6 6 6
# players:      6
# actions:      [6]
# graph:        RandomGraph
# graph_params:[ -nodes 6 -edges 10 ]
# Graph Params:
# { nodes: 6, edges: 10, sym_edges: true, reflex_ok: false }
[1  1  1  1  1  1] :   [ 0.9889 0.1452 0.3467 0.0380 0.2359 0.8738 ]
[2  1  1  1  1  1] :   [ 0.6493 0.4155 0.3467 0.6591 0.4223 0.8738 ]
[3  1  1  1  1  1] :   [ 0.1519 0.1217 0.3467 0.0664 0.6214 0.8738 ]
[4  1  1  1  1  1] :   [ 0.2684 0.4025 0.3467 0.3414 0.7824 0.8738 ]
[5  1  1  1  1  1] :   [ 0.2793 0.9420 0.3467 0.1562 0.4247 0.8738 ]
[6  1  1  1  1  1] :   [ 0.1645 0.1241 0.3467 0.3051 0.4003 0.8738 ]
[1  2  1  1  1  1] :   [ 0.8218 0.0256 0.3467 0.1831 0.5240 0.7784 ]
[2  2  1  1  1  1] :   [ 0.9571 0.1945 0.3467 0.1309 0.5542 0.7784 ]
[3  2  1  1  1  1] :   [ 0.2123 0.1948 0.3467 0.3615 0.2782 0.7784 ]
[4  2  1  1  1  1] :   [ 0.3500 0.3485 0.3467 0.3791 0.2542 0.7784 ]
[5  2  1  1  1  1] :   [ 0.7256 0.0974 0.3467 0.7004 0.0661 0.7784 ]
[6  2  1  1  1  1] :   [ 0.0774 0.8020 0.3467 0.3862 0.7296 0.7784 ]
                       …
                       …
```

Figure 1: Data sample from a 6-player random graphical game

In n-player games, assuming that each player has the same number of a actions, a normal-form representation requires a^n entries of action profiles to describe multiple players' utilities. However, as the number of players n increases, the action profiles size grows exponentially. Thus, a compact representation to capture how every player's action choice influence others' utilities is interesting and critical.

Next, we introduce an algorithm for learning a graphical structure representation of a multiplayer game from its normal form representation using a multi-gradient descendent regression model.

Structure Learning Algorithm: MDRLSA

In an action space, not all actions have direct impact on a given action at a certain stage. Let each node denote a player. We identify the influence between paired actions as a connection, described as one edge. Knowing the influence between players' action choices can provide a compact representation for player's utility function, as well as reduce the search space for each player's utility function learning process. In this paper, we provide a structure learning algorithm, "Multi-Descendent Regression Learning Structure Algorithm" (MDRLSA), to extract the action connections/edges from graphical games.

We are given a set of data points $(\boldsymbol{x}, \boldsymbol{y})$, each describing an instance where players choose a pure strategy profile x and realized value $\boldsymbol{y} = (y_1, \cdots, y_{n_p})$. For deterministic games of complete information, y is simply $f(x)$. We address payoff-function learning as a standard regression problem: selecting a function $\hat{f}$ to minimize some measure of deviation from the true payoff function f. The MDRLSA algorithm uses a regression model to learn a player's utility function. Our hypothesis is that each player's utility function can be represented as a linear function of all players' individual action choices.

The algorithm proceeds in two steps.

Step 1: given all players' action profiles as input, define parameters $\boldsymbol{\theta}$ and fit $\boldsymbol{\theta}$ to all players' utility profiles $\boldsymbol{y} = [y_1 y_2 \dots y_{n_p}]$, where n_p is the total number of players. For any player k, the hypothesis $h_{\boldsymbol{\theta}_k}(x)$ to approximate its utility y_k is given as the Eq. 1 linear model:

$$\begin{aligned} h_{\boldsymbol{\theta_k}}(x) &= \boldsymbol{\theta}_k^{\intercal} \boldsymbol{x} + \varepsilon_k \qquad (1) \\ &= \theta_{1k} x_1 + \cdots + \theta_{ik} x_j + \cdots + \theta_{n_a k} x_{n_p * n_a} + \varepsilon_k \end{aligned}$$

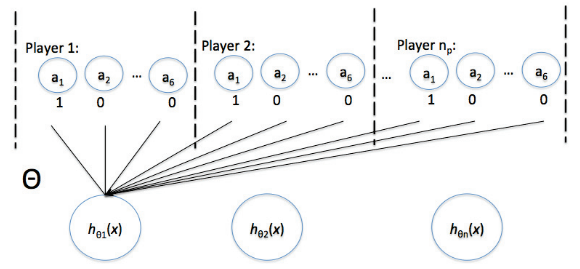
where,

$$\boldsymbol{\theta_k} = \begin{bmatrix} \theta_{1k} \\ \theta_{2k} \\ \vdots \\ \theta_{ik} \\ \vdots \\ \theta_{n_a k} \end{bmatrix}, i \in [1, n_a], j \in [1, n_p * n_a], k \in [1, n_p].$$

Here, $\boldsymbol{\theta}_k^{\intercal}$ is the *transpose* of $\boldsymbol{\theta_k}$, for every θ_{ik}, $\theta_{ik} \in \mathbb{R}$; x_j represents a player's action choice, $x_j \in \{0, 1\}$, where $x_j = 1$ indicates taking action x_j and $x_j = 0$ indicates action x_j is not chosen. Variable ε_k is an unobserved random variable to add noise to the linear relationship. To simplify the explanation, we assume all n_p players have the same number of action choices, denoted by n_a. For any player k in a n_p-player

[2]If there is not an edge between i and j, it means their payoff is completely independent (as the payoff has been generated without additional noise.) In real games, it needs to consider a cutoff threshold in the situation when the dependence is very small.

game, joint strategy of $[x_1, x_2, \ldots, x_{n_a}]$ describes player 1's action profile, and $[x_{n_a+1}, x_{n_a+2}, \ldots, x_{n_a+n_a}]$ represents player 2's action profile, and so on. Here, $j \in [1, n_p * n_a]$, $n_p \times n_a$ is the total number of bits used to represent a joint strategy profile. For example, in a 6-player game, where each player has 6 action choices, the total number of bits used to represent a strategy profile is 36. The first 6 bits are used to represent action choice of player 1, and only 1 bit can be 'on' to show a single action choice of this player. In Figure , it shows joint strategy $\boldsymbol{x} = [x_1, x_2, \ldots, x_{36}]$ which includes each player's action choice towards $a_1, a_2, \ldots, a_6$.



$\{n_p$ player' action mapping

The objective is to adjust the parameter $\boldsymbol{\theta}_k$ values to approximate the function value $h_{\boldsymbol{\theta}_k}$ to the payoff value y_k given all players' joint strategy profiles. Thus, we define the cost function $J(\boldsymbol{\theta}_k)$ as the difference in values between the approximated $h_{\boldsymbol{\theta}_k}$ and the obtained payoff value y_k of any given player k, see Eq. 2:

$$J(\boldsymbol{\theta}_k) \quad = \quad \frac{1}{2m} \sum_{l=1}^{m} \left(h_{\boldsymbol{\theta}_k}(x^{(l)}) - y_k^{(l)} \right)^2 . \qquad (2)$$

Here, m is $n_p^{n_a}$, the total count of joint strategy profiles. We apply multi-gradient descent to achieve the objective of using linear regression for each player k to minimize the cost function value $J(\boldsymbol{\theta}_k)$ by adjusting the θ_{ik} values. In batch gradient descent, each iteration simultaneously updates θ_j for all j in Eq. 3:

$$\theta_{ik} := \theta_{ik} - \alpha \frac{1}{m} \sum_{l=1}^{m} \left(h_{\boldsymbol{\theta}_k}(x_k^{(l)} - y_k^{(l)}) \right) x_{ik}^{(l)} . \qquad (3)$$

In this experiment, we randomly initialize the parameters Θ to 0, the learning rate α to 0.01 and number of iterations as 400. In order to avoid the model over-fitting or under-fitting the data, we use Figure 2 to observe the performance of our hypothetical linear regression model. Figure 2 shows the cost function $J(\boldsymbol{\theta}_k)$ as each k^{th} player's utility loss function is decreasing as the number of iteration increases. This decrease of the cost function J demonstrates that our linear model hypothesis stated in Eq. 1 has accurately captured the function between players' action choices and their utilities.

However, this batch gradient descent learning is quite slow, as Eq. 2 is summed over all samples. We can rewrite the least-squares cost function by replacing the explicit sum by matrix multiplication, as follows:

$$J(\boldsymbol{\theta}_k) \quad = \quad \frac{1}{2m} \left(X\boldsymbol{\theta}_k - \boldsymbol{y_k} \right)^{\mathsf{T}} \left(X\boldsymbol{\theta}_k - \boldsymbol{y_k} \right) . \qquad (4)$$

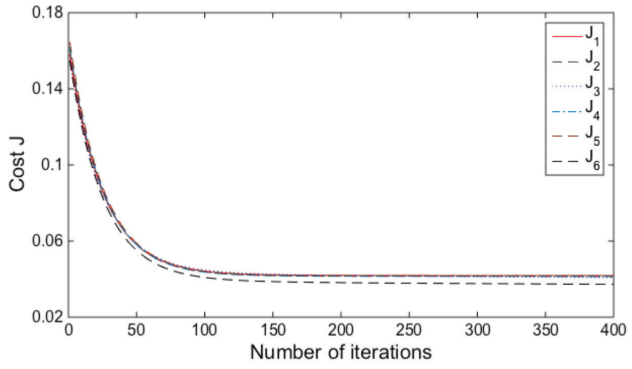


Figure 2: Convergence of gradient descent with learning rate $\alpha = 0.01$

In order to find where the cost function has a minimum, we take derivative by θ and set it to 0, as follows:

$$\frac{\partial J}{\partial \boldsymbol{\theta}_k} = X^{\mathsf{T}} X \boldsymbol{\theta}_k - X^{\mathsf{T}} \boldsymbol{y_k} = 0. \qquad (5)$$

Thus, when the matrix $X^{\mathsf{T}}X$ is invertible, we have,

$$\hat{\boldsymbol{\theta}}_k = (X^{\mathsf{T}}X)^{-1} X^{\mathsf{T}} \boldsymbol{y_k}, \qquad (6)$$

where

$$\boldsymbol{y_k} = \begin{bmatrix} y_{0k} \\ y_{1k} \\ \vdots \\ y_{jk} \end{bmatrix} .$$

$\hat{\boldsymbol{\theta}}_k$ is the optimal $\boldsymbol{\theta}_k$ that minimizes $J(\boldsymbol{\theta}_k)$. In other words, in a large number of data entries, m is large, we choose normal equation Eq. 6 to optimize $\boldsymbol{\theta}$ instead of Eq. 3 if the matrix $X^{\mathsf{T}}X$ is invertible. As we can see, Eq. 6 includes no loop in the program, and the learning rate α is not required; the optimal $\hat{\boldsymbol{\theta}}_k$ is computed directly.

Step 2: We define the influence coefficiency between each player's action choice and all possible combination of action choices, denoted as Θ:

$$\Theta = [\boldsymbol{\theta}_1 \ldots \boldsymbol{\theta}_k \ldots \boldsymbol{\theta}_{n_p}].$$

According to the regression model, the parameter Θ is learned as the best fit for the payoff functions for all players. Next, we map the coefficiency into players action influential relationships based on the given utilities. The guideline is as follows: for any player p, for any move by p and all other players other than q, if the payoff obtained by p from the joint strategy by any choice of q, differs little between the best move of q and the worst move of q, then we note player p and q do not have an influential relationship between them; otherwise, an influential edge exists between player p and q. This independence between p and q also corresponds to coefficiency between the two players strategy choices being constant.

Definition 2. Consider a game $[I, (\boldsymbol{x}), \boldsymbol{y}(s)]$, player p and q have *no influential relationship* and are δ*-independent,* if for every $x_p, x_p \in X_p$, and for any available joint strategy of x_{-pq},

$$\max_{i \in [1,N_a]} y_p(x_p, x_q^i, x_{-pq}) - \min_{j \in [1,N_a]} y_p(x_p, x_q^j, x_{-pq}) \leq \delta$$

where, N_a is the total number of action choice, x_p is p's strategy with a reward y_p, x_q is q's strategy, and the rest of the players' joint strategy is denoted as x_{-pq}.

We define an influence graph as a $n_p \times n_p$ binary matrix to represent the influential connections among players, as follows:

$$\text{graph\_param}(i,j) = \begin{cases} 0, & \text{if } i \,\&\, j \text{ are } \delta\text{-independent;} \\ 1, & \text{otherwise.} \end{cases}$$

δ is set as a parameter to control the tolerance level of the influence among players.

Alg. 1 describes the graphical structure learning algorithm MDRLSA, which extracts the influence graph between players, i.e. whether they are related or independent. Step 1, essentially uses Eq. 6 to compute Θ to linearly approximate payoff functions for all players. Step 2, first initializes a $n_p \times n_p$ matrix to an all-ones matrix as graph parameters, to represent full connections among between any two players; second, in the matrix, for each column player, it searches all row players, according to Def. 2, to identify the δ-independent players and set the corresponding value as '0' in the graph parameter matrix. For each row player, we take the coefficiency of this player's learned payoff function, if for all of the row player's strategies combined with all other players' joint strategy provides a utility for the given column player that differs within δ, we identify this row player as having no influence ('δ-independence') on this column player. We repeat this until every pair of players has been checked.

Emperical Evaluation

We implemented the Multi-Descendent Regression Learning Structure Algorithm (MDRLSA) in Matlab, which computes a graphical structure representation of players actions' influence for multiagent graphical games.

Experimental Results

Taking one random graphical game with six players, six actions and ten influence edges as an example, the influence graph learned among the players is shown in the following matrix:

$$graph\_param = \begin{array}{c} player \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \end{array} \quad (7)$$

Figure 3 illustrates the conversion of Eq. 7 into a graph representation.

From the graph in Figure 3, we can produce utility functions for this 6-player game, shown in Eq.8. f_{p_i} represents

Algorithm 1: MDRLSA

```
Data: $X$ := action_profile;                         // read game file;
  $U$ := utility_profile;
  $\delta$ := 0.00001                        // set $\delta$ to a very small value;
Result: $\Theta$, $E$, graph_param;
        // coefficiency data, estimated error, influence graph
parameters;
begin
    Step 1: Use Regression Model to Learn Payoff
      Functions: compute $\Theta$ ;
    Initialize: $\Theta$ = [ ];
    for each player $i$ do
        $y = U(:, i)$;     // get player i's payoff: $i^{th}$ column;
          // notation: U(:,i) takes the vector of the ith column
        from the matrix U; similarly to others;
        $\Theta = [\Theta$, normalEqn($X, y$)];
                              // Compute $\Theta$ according to Eq. 6;
    end
    Step 2: Convert to Graphical Representation: map
      coefficiency $\Theta$ into the graph connections;
    Initialize:
    data := $\Theta$(1:end,:);
        // all players' coefficiency excluding modelled noise;
    $[m, n]$ := size(data);                // $n$ is number of player;
    graph_param := ones(n,n);
          // initial graph parameter matrix to all connected;
    temp_coef := zeros(n,n);       // initial a temporary matrix;
    for each column player $i$ do
        $a$ := data(:,i);
          // take the vector of the $i^{th}$ column: player $i$'s data
        for each row player $j$ do
            index = (j-1)*(m/n);
              // get the index for player $j$'s action choice in
            player $i$'s data;
            temp = a((index+1):(index+n));
                  // extract all data reflecting $i$ and $j$'s joint
            strategy combinations;
            player_coef_max = max(temp);
                                                    // find the max;
            player_coef_min = min(temp);
                                                    // find the min;
            temp_coef(j,i) = player_coef_max $-$
              player_coef_min;
             // compute the difference of $j$'s action choices of
            max and min to player $i$;
            if temp_coef(j,i) $\leq \delta$ then
                    // all action choices of $j^{th}$ row player's
                influence on $i^{th}$ column player less than $\delta$;
                graph_param(j,i) = 0;
                  // set graph parameter as '0', player i and j
                $\delta$-independent;
            end
        end
    end
end
```

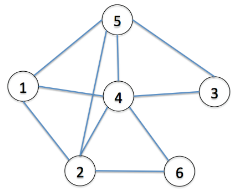


Figure 3: 6 player graphical game structure

the utility function for the i_{th} player. In this 6-player game, a complete normal-form representation (matrix) includes 6^6 entries for each player to query in order to make an optimal choice decision. However, according to Eq.8, the complete matrix can be partitioned into smaller matrices. For example: player 1 requires a matrix of 6^4 entries; player 6 requires a size of 6^3 entries. For all 6 players, the total size of the matrix required to represent the game is 18576 ($= 6^5 * 2 + 6^4 * 2 + 6^3 * 2$), which is approximately 40% of the original.

$$F = \begin{cases} f_{p_1} &= f_1(p_1, p_2, p_4, p_5); \\ f_{p_2} &= f_2(p_1, p_2, p_4, p_5, p_6); \\ f_{p_3} &= f_3(p_3, p_4, p_5); \\ f_{p_4} &= f_4(p_1, p_2, p_4, p_5, p_6); \\ f_{p_5} &= f_5(p_1, p_3, p_4, p_5); \\ f_{p_6} &= f_6(p_2, p_4, p_6); \end{cases} \quad (8)$$

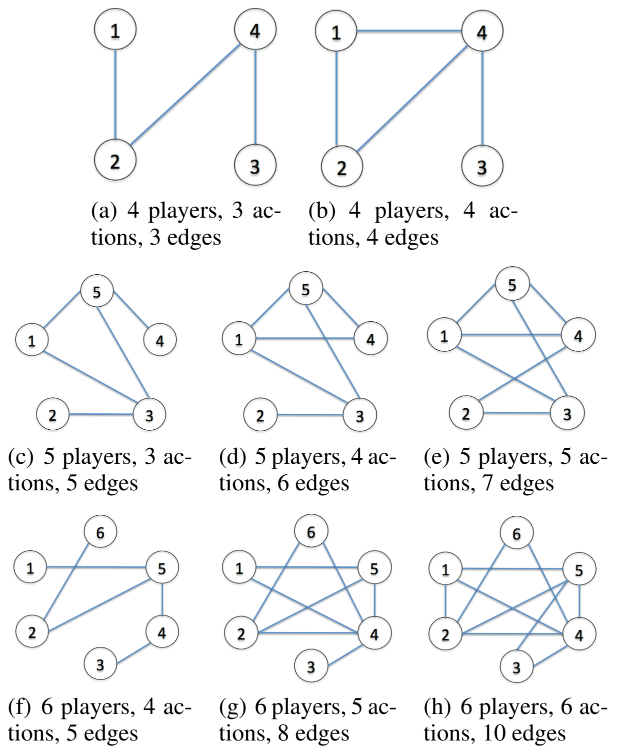
Comparing the structure shown in Eq. 7 with the benchmark generated by GAMUT, MDRLSA learns an accurate underlying structure for a 6-player random graphical game.

We tested MDRSLA on a set of random graphical games with a different number of players, actions and number of influence edges generated from GAMUT. As we can see in Table 1, MDRSLA can efficiently identify the independence among players and extract the influence graph accurately. In Figure 4, the graphical structural representation learned using MDRSLA is shown for the given set of random graphical games, a to h. The running time for each game, using an average of ten runs of MDRLSA, is listed in Table 1. The program runs in Matlab on Mac OS X, with Processor 2.8 GHz Intel Core i7, Memory 8GB 1067 MHz.

Game	Player Number	Action Number	Edge Number	Runtime (Seconds)	Accuracy (%)	Normal Form Profile Entries
a.	4	3	3	0.0063	100	81
b.	4	4	4	0.0091	100	256
c.	5	3	5	0.0136	100	243
d.	5	4	6	0.0156	100	1024
e.	5	5	7	0.0193	100	3125
f.	6	4	5	0.0340	100	4096
g.	6	5	8	0.1029	100	15625
h.	6	6	10	0.2999	100	46656

Table 1: MDRLSA performance on random graphical games experiments

Given the listed random generated graphical games in the table, the structures learned are shown in Figure 4. MDRLSA shows robust promising results of learning structure representation efficiently and effectively in random graphical games. While the number of strategy profiles grows exponentially with the number of players or actions, the runtime increases linearly to the number of strategy profiles.



(a) 4 players, 3 actions, 3 edges (b) 4 players, 4 actions, 4 edges

(c) 5 players, 3 actions, 5 edges (d) 5 players, 4 actions, 6 edges (e) 5 players, 5 actions, 7 edges

(f) 6 players, 4 actions, 5 edges (g) 6 players, 5 actions, 8 edges (h) 6 players, 6 actions, 10 edges

Figure 4: MDRLSA learned graphical structures

Comparison

Duong et al. (2009) give a state-of-art work of structure learning algorithm for graphical game structure learning. This approach comes from a game theoretical perspective, which constructs a loss function and focuses on minimizing the loss of utility function in strategy choice. However, our approach comes from a machine learning perspective, and focuses on revealing the coefficiency between all the action choices and the outcome utility, directly converting the payoff functions to graphical structures. Through the correlated coefficiency, the relative neighbour influence is identified. Both approaches are tested on GAMUT generated games.

In terms of structure accuracy of the learning algorithms, MDRLSA has shown consistent 100% accuracy for any given random number of influence edges which exist among players in a game (see Figure 1); Duong et al., (measured accuracy as structural similarity), demonstrated approximately 90% structural similarity when the maximum of 6 edges is allowed for any player. In terms of the runtime efficiency, without demonstrating this in the same programming languages, we cannot evaluate the runtime efficiency difference for both methodologies. However, Duong et al.'s structure learning algorithms (written in Java) have shown that runtime increases exponentially as the number of constraints (maximum number of connections) increases for each player. It has also shown that the runtime takes approximately 200 seconds for a maximum of 3 edges allowed for any player, and above 500 seconds for a maximum of 5 edges allowed for any player. Comparing with MDRLSA, for a maximum of 5 edges each player, see Figure 4 (h), runtime is approximately 0.3 seconds (written in Matlab), which is significantly faster (even though Matlab has slower performance compared to Java). Furthermore, comparing both approaches, there are methodological advantages: MDRLSA is intuitive, straightforward and simple, and shows efficacy compared with state-of-the-art results.

Concluding Remarks and Future Work

In this paper, we presented and developed a method and algorithm to identify the structure in a self-interested multiagent environment. The Multi-Descendent Regression Learning Structure Algorithm learns a compact representation among agents' action influence towards each other. MDRLSA was tested on a set of randomly generated graphical games generated using GAMUT. Experiments demonstrate that MDRLSA is suited to various graphical game applications, and provides promising results. The structure representation compared with normal form utility matrices reduces the search space and identifies the mutual action influence among agents. In fully observable games, MDRLSA can learn an accurate representation for games where there are strong influences in individual player payoff.

We believe that MDRLSA can be applied for games with either a large number of players or a large number of actions. Our method is useful and practical to achieve some reduction in the search space, when the expected payoff of certain actions is only affected if some other player chooses certain other actions. In future work, we will work to scale up MDRLSA and extend it to deal with a large number of actions or a large number of players in computer games where this abstraction technique is practical. On one hand, we can adjust the parameter δ to balance the tradeoff between the amount of computation of a game and approximation. The larger we set the δ parameter, the coarser the approximation of the game, but the smaller the number of connections in the graphical game, resulting in larger computational gains. On the other hand, we can extend MDRLSA from graphical games representation to other types of games, such as action-graph games. Action-graph games (Bhat and Leyton-Brown 2004; Jiang, Leyton-Brown, and Bhat 2011) provide a "dual" representation where nodes are actions and the edges between nodes express context-specific interdependencies of the utilities. The proposed approach could be applied even for games with small numbers of players, for example, a two-player game with large numbers of actions, while still aiming to achieve some reduction in the search space. Using a learned compact representation, it can speed up search in the action space and estimate the payoff for global strategy planning, then utilize standard methods for game playing.

Acknowledgments

We are grateful for the support from NSERC, and the anonymous reviewers for the helpful suggestions.

References

Bhat, N. A. R., and Leyton-Brown, K. 2004. Computing Nash equilibria of action-graph games. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI '04, 35–42. Arlington, Virginia, United States: AUAI Press.

Daskalakis, C.; Goldberg, P. W.; and Papadimitriou, C. H. 2006. The complexity of computing a Nash equilibrium. In *SICOMP*, 71–78. ACM Press.

Duong, Q.; Vorobeychik, Y.; Singh, S.; and Wellman, M. P. 2009. Learning graphical game models. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 116–121. Morgan Kaufmann.

Finnsson, H. 2012. Generalized Monte-Carlo tree search extensions for general game playing. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, AAAI'12, 1550–1556. AAAI Press.

Genesereth, M., and Love, N. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* 26:62–72.

Jiang, A. X.; Leyton-Brown, K.; and Bhat, N. A. 2011. Action-graph games. *Games and Economic Behavior* 71(1):141–173.

Kearns, M.; Littman, M.; and Singh, S. 2001. Graphical models for game theory. In *Proceedings of the Seventeenth Conference Annual Conference on Uncertainty in Artificial Intelligence*, 253–260. Morgan Kaufmann.

Nudelman, E.; Wortman, J.; Shoham, Y.; and Leyton-Brown, K. 2004. Run the GAMUT: A comprehensive approach to evaluating game-theoretic algorithms. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '04, 880–887. Washington, DC, USA: IEEE Computer Society.

Rosenthal, R. W. 1973. A class of games possessing pure-strategy Nash equilibria. *International Journal of Game Theory* 2(1):65–67.

Shoham, Y., and Leyton-Brown, K. 2009. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529(7587):484–489.

Vorobeychik, Y.; Wellman, M. P.; and Singh, S. 2007. Learning payoff functions in infinite games. *Machine Learning* 67(1-2):145–168.