A Game Playing System for Use in Computer Science Education

James MacGlashan
University of Maryland,
Baltimore County
1000 Hilltop Circle
Baltimore, MD
jmac1@umbc.edu

Don Miner
University of Maryland,
Baltimore County
1000 Hilltop Circle
Baltimore, MD
don1@umbc.edu

Marie desJardins
University of Maryland,
Baltimore County
1000 Hilltop Circle
Baltimore, MD
mariedj@cs.umbc.edu

Abstract

The MAPLE Game Playing System is a web application and website that allows students to design and program game playing agents using the Python programming language. The system provides a platform for assignments in introductory computer science courses and senior and graduate-level A.I. courses. The website allows users to upload, use, and share agents that play games such as the Prisoner's Dilemma, Stag Hunt, and Matching Pennies. In this paper, we discuss the features and functionality of the system and suggest possible assignments within A.I. or introductory programming courses.

Introduction

We have developed the *MAPLE Game Playing System* (MGPS), an online two-player iterated normal-form game playing system for students in introductory and advanced college-level courses. MAPLE—an acronym for the research lab in which the system was developed—stands for Multi-Agent Planning and LEarning. Normal-form games are a class of multi-agent competitive games that can be represented by a payoff matrix. The most well known normal-form game is the *prisoner's dilemma* game. Both prisoners dilemma and other normal-form games are reviewed more in the background section of this paper.

Since normal-form games have very simple rules, they are an effective context within which to teach introductory computer science. At the same time, designing an agent that optimally plays normal-form games is a challenging task, providing a platform for more advanced A.I. education and research. Currently, tools for educators and researchers to use for normal-form game development are limited. The MGPS is designed to provide a centralized source for educators and researchers to create new agents and hold tournaments to test the effectiveness of agents against each other. With the MGPS, instructors can assign their students to create their own agents for playing both specific and general normal-form games. Students would be able to test their agents against their classmates' agents, or against top-performing agents designed in the research community. At the end of the assignment, the instructor could organize tournaments with all of the students' agents competing in a variety of different normal-form games.

	cooperate	defect
cooperate	$3, 3$	$0, 5$
defect	$5, 0$	$1, 1$

Figure 1: The payoff matrix for the prisoner's dilemma game. If both agents cooperate, they both get a payoff of 3, which is a better payoff than both defecting (1). If one prisoner defects and one cooperates, the cooperator gets a payoff of 0, and the defector gets a payoff of 5.

MGPS can also be used as a effective tool for research. While research tournaments for normal-form game playing agents are sometimes held, such as the Iterated Prisoner's Dilemma Competition (Kendall, Darwen, and Yao 2009), to our knowledge there does not exist a centralized system of agents against which researchers can test their agents. As a result, researchers are forced to re-implement each agent they want to test against before submitting their agent to a competition. By using the MGPS, researchers can quickly test their ideas against other cutting-edge agents without having to re-implement the algorithms themselves. Furthermore, if a researcher missed an opportunity to participate in an official tournament, they can easily recreate the tournament using MGPS to see how their agent would have fared.

We first review the structure of normal-form games and a number of specific examples. We then discuss the organization and features of the MGPS, and how how the MGPS can be effectively used in both introductory CS courses and advanced A.I. courses. Finally, we present our conclusions and future improvements that we would like to make to the MGPS.

Background

In this section, we review the structure of normal-form games, and give examples of specific games. Normal-form games describe n-player games in which each of the n players can take one of m actions. All players choose their actions simultaneously so that no player knows the choice of its opponents before making its own. Given the choice of each player's action, each player will receive a defined re-

	stag	hare
stag	4,4	1,3
hare	3,1	3,3

(a) Stag Hunt

	movie 1	movie 2
movie 1	3,2	0,0
movie 2	0,0	2,3

(b) Battle of the Sexes

	heads	tails
heads	1,-1	-1,1
tails	-1,1	1,-1

(c) Matching Pennies

	swerve	straight
swerve	0,0	-1,+1
straight	+1,-1	-10,-10

(d) Chicken

Figure 2: The payoff matrices for the two-player normal-form games Stag Hunt, Battle of the Sexes, Matching Pennies, and Chicken

ward. These conditional rewards are usually represented by a *payoff matrix*. In a two-player game with two possible actions, the payoff matrix would be a 2x2 matrix. The first row would define the rewards the players would receive if player one took the first action; the second row, the rewards received if player one took the second action. Similarly, the two columns would represent the rewards for each action of the second player. For example, the cell in row one and column one defines the rewards if both player one and two took the first action. While a player is not aware of what action its opponents will take, it is aware of the payoff matrix.

The most well known normal-form game is the *Prisoner's Dilemma* (PD) (Axelrod 1980) (Axelrod and Hamilton 1981), in which two prisoners find themselves in the following situation:

> Two suspects are arrested by the police. The police have insufficient evidence for a conviction, and, having separated both prisoners, visit each of them to offer the same deal. If one testifies (defects from the other) for the prosecution against the other and the other remains silent (cooperates with the other), the betrayer goes free and the silent accomplice receives the full 10-year sentence. If both remain silent, both prisoners are sentenced to only six months in jail for a minor charge. If each betrays the other, each receives a five-year sentence. Each prisoner must choose to betray the other or to remain silent. Each one is assured that the other would not know about the betrayal before the end of the investigation. How should the prisoners act? (Wikipedia 2009b)

The PD game can be represented using a variety of different payoff matrices provided that the temptation to defect is greater than the reward for both cooperating, that the reward for both cooperating is greater than both defecting, and that the reward for both defecting is greater than reward when a player cooperates and their partner defects. One such payoff matrix is shown in Figure 1.

The optimal strategy for an agent playing the PD, when faced with only one game or a fixed known number of games, is to defect, since defect-defect is the Nash Equilibrium for PD. A Nash Equilibrium is defined as a solution concept for a game where no player would benefit from changing their strategy if their opponents did not change theirs (Nash 1950).

When the number of PD games to be played is not known, and each player has a history of their opponent's previous choices, the optimal strategy depends on the mixture of strategies that each player employs. Games that are repeated in this way are known as *iterated* games. For the iterated PD (IPD), the best strategies are variations of tit-for-tat, where the agent starts by cooperating in the first game and then performs the same action as the opponent took in the previous game. (Axelrod and Hamilton 1981).

While PD is a very common normal-form game with some interesting properties, there are many other two-player games, each with different properties and requiring different strategies. Other two-player iterated normal-form games include Stag Hunt, Battle of the Sexes, Matching Pennies, and Chicken. The payoff matrix for each of these can be found in Figure 2. A more detailed description of these games and more can be found on Wikipedia (2009a).

Stag Hunt represents a game of trust. In this game, each player receives the highest reward when they both cooperate (hunt for a large animal—a stag—together). If both defect (independently hunt for an easy hare), then they each receive a moderate reward. If one defects and one cooperates (cooperator left waiting for the other to hunt for the stag), then the cooperator receives a low reward, while the defector receives a moderate reward. If they both defect and hunt for a hare on their own, they both receive a moderate reward. In this game, each agent must trust the other agent to cooperate in order to maximize each others' rewards. If for some reason, the agent believes that its opponent would defect, then it would be better for the agent to defect in turn.

In the Battle of the Sexes game, each player has a different preferred action, representing a preferred movie. If player one and two both choose to watch player one's preferred movie, then they each receive a reward, with player one receiving a slightly higher reward than player two. The inverse is true if they both choose to watch player two's preferred movie. If the players can't agree on what movie to watch, then no player receives a reward. Players need to follow a cooperative strategy that allows them both to receive at least some reward. Ideally, a player should be able to coerce its opponent into its preferred choice merely by the choices it makes in previous games.

The Matching Pennies game is a two-action variant of the classic children's game of Rock-Paper-Scissors. When the two players' actions match, player one receives a positive reward and the player two receives a negative reward. When the players' actions are different, player two receives a positive reward and player one receives a negative reward. Generally the best strategy for this game is to play randomly. However, if one player can identify the other player's strategy, they may be able to exploit it for higher total reward.

Chicken is based on the "game of chicken," in which two automobile drivers are driving head-on towards each other. If both players swerve their cars, neither wins, but they both

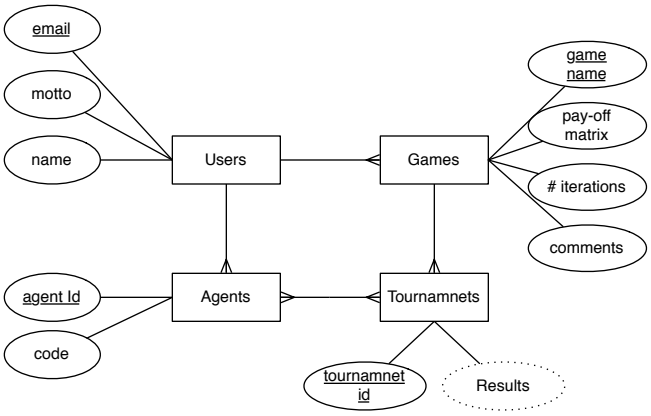

Figure 3: Shown here is an ER diagram of the MGPS database. MGPS allows registered users to create any number of agents that are programmed in the Python Programming language. Users can also define their own normal-form games. Any user can start their own tournament and can choose the participating agents, regardless of whether that user created the agents.

Figure 4: The user page; from here, users can view any agents a user has created and any other details the user has provided about themselves.

survive. If one player swerves, they both survive and one player (the non-swerver) wins. If neither swerves, then the players collide, and both die. The normal-form game is represented by "swerving" and "going straight" actions. When both players swerve they each receive zero reward. When one player swerves, the player who swerved receives a small negative reward and the player who kept going straight receives a small positive reward. If both players go straight, they each receive a large negative reward. The interesting aspect of this game is that it is best for a player to go straight when the other player swerves—but the risk of going straight is very large, because if the other player doesn't swerve, then there is a very large penalty.

Features and Organization

MGPS is implemented in CherryPy,[1] an HTTP framework. A feature of CherryPy is the ability to run the web application within the built-in web server, which is convenient for instructors or students if they want to run MGPS privately. The only system requirements for running a private server are Python 2.6 and CherryPy 3.1.2. The system is available for public use on our servers for convenience.[2]

The structure of the MGPS is shown in an ER diagram in Figure 3. MGPS provides user management, a centralized database of all submitted agents, a database of submitted normal-form games, and the ability to host tournaments that the MGPS will run and then report the results for.

Users make accounts on the website that store a personal profile, as well as a list of the user's agents (Figure 4). Agents are publicly available to all users to add to tournaments or to view. We plan on adding a privacy functionality to protect the source code of an agent, which would be useful in a class environment where cheating might be a problem.

An agent's source code must follow a strict template, having four specific interface methods and one method that instantiates the agent. Conforming to this template is important because the system must know how to interact with the agent. It is important to note that agents are not reinstantiated between games (thus the need for *start* and *end* methods). In this way, agents can be designed to learn over the course of an entire tournament. The four class methods that must be available are the following:

- `def start(self, board)` – Tells the agent that a new game will be starting. Any sort of initialization to prepare for the game should be done here. The parameter *board* is the game payoff matrix to be used.
- `def get_action(self)` – Returns the action the agent would like to perform next. Actions are enumerated as numbers; in prisoner's dilemma, 0 is cooperate and 1 is defect. An agent may identify these differences by examining the board object passed in the start method.
- `def add_result(self, iteration, your_choice, your_reward, your_score, others_choice, others_reward, others_score)` – Tells the agent what happened in the previous round: what the agent's last choice was, the agent's opponent's last choice, the payoff each received, and the two agents' respective cumulative payoffs from their match.
- `def end(self)` – Tells the agent that the round is over. Any kind of maintenance that needs to be done can be done here.

Once a user has finished programming their agent, they can upload the Python code from their local computer to the MGPS website using a simple web form.

[1] CherryPy website: http://www.cherrypy.org/
[2] http://maple.cs.umbc.edu

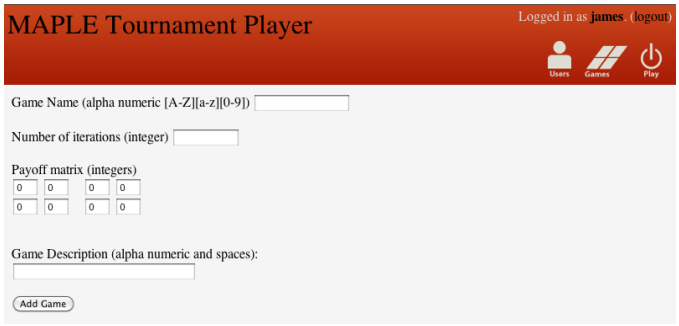

Figure 5: The interface for creating a new game. Users can specify the payoffs for each choice, and the number of iterations to play the game.

Because different normal-form games can require different strategies, another ability of the system is allowing any user to add games, as shown in Figure 5. A game definition includes the name of the game, the number of iterations that agents should perform, the payoff matrix, and a short game description. Once the game is added, anyone can start a new tournament with the game. By default, the MGPS comes with classic games such as Prisoner's Dilemma, Stag hunt, Battle of the Sexes, Matching Pennies, and Chicken (Wikipedia 2009a).

MGPS has supporting functionality that makes playing games easy and convenient. First, users select a subset of agents from a list of all agents in the database to be included in the tournament. Then, they select what user-generated or standard game will be played by these agents. The interface for this setup is shown in Figure 6. Once the preferences have been submitted, the server runs the tournament, playing each of the selected agents against each other agent. Several statistics are recorded, including the sum of utility scores, individual rounds and every action taken by an agent. As shown in Figure 7, these statistics are provided in a detailed report that is displayed once the tournament is completed; the report is also saved for future viewing.

We are planning on improving and expanding the functionality of MGPS in the future. Most notably, we will add noise to the game options. Noise in a game means that with a certain probability, an agent's desired action is replaced with a random action. This complicates the problem and introduces concepts such as forgiveness and detection of noise. Also, we will expand the system to include games that are not normal-form, such as chess and checkers. This will make the system more general and allow for a wider diversity of games.

Usage in Introductory Computer Science Courses

MGPS is well suited for introductory computer science classes. The only prerequisite for students using the system is a basic knowledge of Python and basic knowledge of game theory. Python is an easy language to learn and has been shown to be an excellent language for teaching introductory computer science (Agarwal and Agarwal 2005) (Ranum et al. 2006). A lecture on the basics of normal-form games could be very abbreviated or explained in the assignment prompt. For instance, students only need to understand the payoff matrix formulation and how to determine the rewards players receive for any combination of actions.

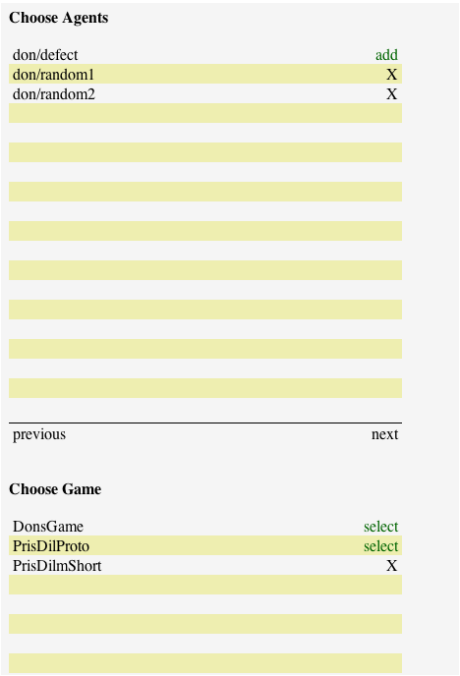

Figure 6: Shown here is the interface for setting up a tournament. A user can select which agents they would like to participate, and the game to play.

In the MGPS, students can see immediate results of their programs and can compare themselves to any number of baselines provided by instructors or other students. Other work studying the effects of games in computer science education has shown that immediate feedback is effective in engaging students (Barnes et al. 2008). This sense of engagement will encourage students to design agents that perform well. Also, the game theory topic is deep, and may inspire computer science students to research the field deeper and get a head start on their senior classes.

A number of programming topics can be explored using the MGPS infrastructure. The role of functions, parameters, and return values can be taught by explaining how to use and implement each of the required agent methods. The effect of return values for instance can be easily demonstrated by defining an agent that always cooperates (returns zero) and having it play against an agent that always defects (returns one).

The MGPS also serves as a good framework to describe top-down design as each agent already has the top-level required methods listed. Students can practice designing from a high level and defining stubs for the lower level methods the top level methods would reference.

Object-oriented design can also be explained using the

Figure 7: Shown here are the results displayed for a tournament consisting of two agents. The individual payoffs for each iteration are reported, as well as the final cumulative scores for each game.

MGPS, since each agent is a defined class. This provides students with a framework to learn how to define additional methods and class variables that can be accessed by any method in the object.

Since agents range widely in complexity, a wide range of projects, in terms of difficulty, can be assigned. For example, when working in PD, agents that always cooperate or defect are the simplest, and can be used as a tutorial for learning the system. From there, students can implement tit-for-tat (TFT), in which the agent acts as the other agent acted in the previous round. For example, if agent A (TFT) is playing against another agent, it will cooperate if the other agent cooperated, and will defect if the other agent defected. To implement this strategy, the student must store the opponent's previous action in a class variable that is assigned in the *add_result* method. Then, the student should program the *get_action* method to return this stored action. Next, students can implement more complicated strategies such as ones that determine the most common action by the opponent and then choose an action to maximize its reward, assuming that the other agent continues its trend. Random numbers can also be used to make decisions, such as cooperating, but sometimes defecting. For example, in the Matching Pennies game, it is best to choose a random action.

Students can gain practice using lists by designing a master-slave system where agents collude by performing a specific sequence of actions, and then allow one agent (the master) to receive the maximum reward by following a strategy that is beneficial to the master. To implement this, students would store a list containing the sequence of introductory actions they should execute to signal to the other colluding agents who they are, and to check to see whether they are playing against one of their fellow colluders.

Finally, students should be allowed to design and implement their own ideas and see how they compare to other students' agents.

Usage in A.I. Courses

MGPS can be used as a supplement to an introduction to artificial intelligence class or a game theory class. Students can be assigned to implement classic strategies in class, which will give deeper understandings of the workings of the agents, as well as the properties of the games. Scientific experiments can be performed to compare strategies and analyze the differences. For example, in which games are Nash Equilibria relevant? Possible assignments, some of which are inspired by research questions, include:

- Implement an agent that identifies Nash Equilibriums and plays accordingly
- Implement a general game playing agent that learns a strategy based on multiple games with multiple agents
- Implement an agent that models its opponents strategy and computes expected utilities to determine the best action
- Implement reinforcement learning for use in PD (Sandholm and Crites 1996)
- Implement evolving strategies in PD (Fogel 1993)
- Implement a master/slave collusion strategy in the noisy PD (Rogers et al. 2007)

Assignments where students need to develop an agent that can play any kind of two-player normal-form game can be particularly interesting since different games can require very different strategies.

MGPS was originally developed for the game theory section of a senior/graduate-level multi-agent systems course, taught by two of the authors of this paper. We ran two competitions over the course of the semester and students appeared to be very engaged and invested in their work. The competitions pitted agents against each other in two events: a standard prisoner's dilemma competition and a general game-playing competition. In the general game-playing competition, the students did not know what games were going to be played and thus had to design their agents to adapt and analyze different game situations and how the other agents were behaving. An interesting aspect of using these two separate formats was that agents which did well in the Prisoners Dilemma often did poorly in some of the other games. This inspired students to write even better general-purpose agents that could perform just as well in PD as PD-specific agents, and also perform well in any other game. Students enjoyed reimplementation and open-ended assignments using the system.

Conclusion

We have developed an entertaining and useful tool to teach students basic programming concepts, as well as teaching more advanced students about normal-form games. We believe that engaging students in exercises such as the ones described in this paper motivate them to perform better in the class and put more effort into their work. In the future, we plan to add support for a wider range of game types; source code privacy options; noise; and the ability for users to submit agents to tournaments, rather than having the tournament host select the agents.

References

Agarwal, K., and Agarwal, A. 2005. Python for CS1, CS2 and beyond. *Journal of Computing Sciences in Colleges* 20(4):262–270.

Axelrod, R., and Hamilton, W. 1981. The evolution of cooperation. *Science* 211(4489):1390–1396.

Axelrod, R. 1980. Effective choice in the Prisoner's Dilemma. *Journal of Conflict Resolution* 3–25.

Barnes, T.; Powell, E.; Chaffin, A.; and Lipford, H. 2008. Game2learn: improving the motivation of CS1 students. In *GDCSE '08: Proceedings of the 3rd international conference on Game development in computer science education*, 1–5. New York, NY, USA: ACM.

Fogel, D. 1993. Evolving behaviors in the Iterated Prisoner's Dilemma. *Evolutionary Computation* 1(1):77–97.

Kendall, G.; Darwen, P.; and Yao, X. 2009. The Iterated Prisoner's Dilemma competition. `http://www.prisoners-dilemma.com/`.

Nash, J. 1950. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences of the United States of America* 48–49.

Ranum, D.; Miller, B.; Zelle, J.; and Guzdial, M. 2006. Successful approaches to teaching introductory computer science courses with Python. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, 396–397. ACM New York, NY, USA.

Rogers, A.; Dash, R.; Ramchurn, S.; Vytelingum, P.; and Jennings, N. 2007. Coordinating team players within a noisy Iterated Prisoner's Dilemma tournament. *Theoretical Computer Science* 377(1-3):243–259.

Sandholm, T., and Crites, R. 1996. Multiagent reinforcement learning in the Iterated Prisoner's Dilemma. *Biosystems* 37(1-2):147–166.

Wikipedia. 2009a. List of games in game theory — Wikipedia, the free encyclopedia. [Online; accessed 9-September-2009].

Wikipedia. 2009b. Prisoner's Dilemma — Wikipedia, the free encyclopedia. [Online; accessed 9-September-2009].