Game Theory Modeling with Prolog

Tyler Yasaka
1847 Palmer St
Florence, AL 35630
256-856-3131
tyasaka@una.edu

ABSTRACT

The goal of this paper is to explore the use of the programming language Prolog to simulate game theory models.

1. INTRODUCTION

Game theory is a relatively new and specialized discipline that lies at an intersection of broader disciplines; among these are computer science, economics, psychology, and philosophy. Its name can be somewhat misleading. The *game* in *game* theory refers not merely to entertainment games that are commonly associated with the word (such as board games or video games), but to the more general concept of a game, in which there are some number of *players* interacting within a given set of rules to reach some objective. So while the concept of game theory could certainly applied to entertainment games, it also has much more interesting applications. Students in a classroom, consumers and producers in an economy, and even life itself can be theoretically represented by a game theory model (though some scenarios are easier to represent than others).

Models for simple games can be constructed and represented with little difficulty; however, more interesting applications of game theory can become more complex. Some automated system to manage the models and perform calculations would be of great use. This is where the programming language *Prolog* comes in.

Prolog is a logic programming language that consists of two basic components: facts and rules. Facts are standalone units of information assumed to be true, while rules are links between facts that allow deductions to be made. In other words, Prolog is a programmatic way to represent logical statements and arguments.

Interestingly, the structure of Prolog and a game theory model are similar in concept. This should not come as a surprise, as both are based heavily in the concepts of logic. For example, the rules of a game might be represented by Prolog rules. Likewise, the players of the game, along with their objectives, could perhaps be represented by facts. This leads us to the central question of this research paper: Is Prolog a viable tool for representing game theory models?

2. BACKGROUND

2.1 Historical Background

First, let us take a step back and examine the roots of the language we would like to study. Prolog was created in 1972 by Alain Colmerauer and Phillipe Roussel [1]. Prolog stands for *PROgrammation en LOGique*, a name proposed by Roussel's wife. Colmerauer and Roussel, among others, were conducting research in topics surrounding the idea of language processing by computers. Interestingly, they were not interested in creating a new language for humans to communicate with computers, so much as they were in enabling computers to understand human language. Their work combined the concepts of automated theorem proving, language processing, and computer programming [2]. Essentially, though they may not have used this term at the time, they were exploring the concept of artificial intelligence.

As work on Prolog progressed, emphasis shifted from developing a single system for intelligent communication and towards developing a *language* for use in creating such systems. The idea was to design a language that, when given a set of facts and rules to relate those facts, could determine other relationships between facts which were logically implied – without ever being explicitly being told how to do so. In other words, the language had to be capable of logical deduction. This would be its definitive feature.

In order to do this, the team focused on the idea of automated theorem proving, which is a form logical deduction. One of the collaborators, Jean Trudel, had conducted considerable research on automated theorem proving; he and Roussel were tasked with using these techniques to develop the deductive capabilities of Prolog. But this would not be an easy task [2].

Not only did the system have to possess powerful and reliable deduction capabilities; it also had to be able to do so efficiently. The team had to take into consideration the speed at which queries would be processed, and especially be wary of combinatorial explosion (computation times that grow exponentially with input size) – all the while keeping in mind the physical constraints of the computer that the program would have to run on. To this end, they turned to *SL-resolution*, a technique discovered by a team that included a man named Robert Kowalski. This technique would form the foundation of the language [2].

2.2 Language Overview

2.2.1 Introduction

Prolog is not a typical language. It is not *imperative*, like C++, Java, or Python. These programs allow the programmer to give the computer a sequential set of instructions which it will then follow. The computer does not know what the problem is, and does not care. Its job is to do as it is told and let the user interpret meaning from the results.

Prolog, on the other hand, is a *declarative* language. No instructions are given to the computer directly by the programmer. Rather, the programmer simply *describes* the problem he/she would like to solve. The computer, understanding the problem, takes it from there and returns the answer to the problem. If this sounds too good to be true, that is probably because it is. In practice, there is a trade-off between validity, efficiency, and ease of use (at least with current computer hardware and currently known algorithms). Prolog attempts to strike a balance.

As previously stated, Prolog uses SL-resolution to make its logical deductions. SL resolution stands for *linear resolution with selection function*. It is a technique for deducing conclusions using first-order logic which, at the time of its discovery, was far less redundant (and thus more efficient) than previously-known techniques. SL-resolution can be implemented in multiple ways; in Prolog, it is implemented depth-first. This has the benefit of being highly efficient (compared to other implementations), but it is also incomplete (which means the program may go into an infinite loop and not find all solutions for certain scenarios) [3].

Because Prolog places a high emphasis on validity and efficiency, ease of use is somewhat sacrificed. That is, problems must be declared in a very explicit format, one that is not always intuitive to humans (though it can be mastered with practice). That said, Prolog is a very simple language.

2.2.2 Basic Usage

So how is Prolog actually used? Prolog relies on two fundamental types of *clauses*: *rules* and *facts* [4]. A rule is simply a statement of the form “If X, then Y”, with *X* and *Y* representing other statements. We can think of a rule as a logical argument, in which there is a premise (e.g. *X*) which, when true, implies a conclusion (e.g. *Y*). Facts are statements that are assumed to be true (e.g. “The sky is blue”). A fact is actually just the conclusion of a rule whose premise is always true.

Below is an example of a fact in Prolog:

```
sky.
```

And here is an example of a rule:

```
clouds :- sky.
```

The fact is just what is looks like. *sky* has been stated to be unconditionally true; hence it is a fact. Looking at the rule, we see *sky* again, but we also see a strange token (*:-*) as well as the word *clouds*. The colon followed by the hyphen signifies that the statement holds only if whatever follows the symbol is true. The rule can be read, “*clouds* if *sky*.”

A user is able give Prolog a set of rules and a set of facts, and then ask questions relating to that information. Prolog will do its best to find any relevant information that it either knows or can deduce, and will return that information to the user.

So using our examples of a rule and a fact, how would we ask Prolog a question? For example, say we wanted to know if there were clouds. We could just ask prolog:

```
clouds.
```

Prolog would respond, “true”. Notice that *clouds* was not one of our facts. Prolog was able to deduce from the fact *clouds* and the rule *clouds :- sky* (“clouds if sky”) that because *sky* was true, *clouds* also must be true.

2.2.3 Data Types

2.2.3.1 Numbers

Prolog has built-in support for basic numbers, including integers and (in most implementations) decimal numbers [4].

2.2.3.2 Atoms

Atoms are non-numeric, constant terms. They can either be represented as a sequence of characters or as a sequence of characters enclosed in single quotes [4]. As their name implies, they are used as building blocks for other aspects of a program.

2.2.3.3 Variables

Variables are just what one would assume they are. They are placeholders that can store values of other types. This could be a number, an atom, or some other type. Also worth noting is that a variable must begin with either a capital letter or an underscore [4].

2.2.3.4 Compound Terms

A compound term, as its name implies, a combination of other terms. Specifically, a compound term begins with an atom and is followed by a set of parentheses, inside which is a list of “arguments” [4].

Compound terms, interestingly, can be used to either represent data or produce an output (similar to that of a function). On their own, compound terms are just facts. However, when followed by the *:-*, they become rules. When they are rules, the *arguments* of the compound term can behave much like arguments to a function.

While rules can be used like functions, they are not actually functions. This is an important distinction to make.

2.2.3.5 Lists

Lists are an important feature in Prolog. They allow sets of data to be grouped together and stored in a single variable, just as one would expect. Lists, like variables, can store values of any type [4]. A typical list might look like this:

```
[1,2,3,4,potato]
```

However, lists in Prolog are very different from lists in most other languages.

First of all, only one item in a list can be accessed; it is called the *head*. The head can be "popped" off of the list with a special type of syntax:

```
[Head|RemainingList]
```

Here, Head represents the head of the list, while RemainingList represents the rest of the list, which will have a new head. This leads to an interesting question: how does one access an item after the head?

The solution is to use recursion, which Prolog lists rely heavily on. One can keep popping off the head of the list until there are no more items in the list. How does one know when there are no items in the list? Prolog has a special symbol for denoting an empty list:

```
[]
```

Suppose there was a fact such as the following:

```
getToEndOfList([]).
```

Passing an argument into getToEndOfList would only evaluate to true if an *empty* list was the argument.

2.2.4 Rules and Goals

The true power of Prolog comes from rules and goals. As discussed previously, rules are conditional statements. For a given set of arguments passed into a rule, the rule will evaluate to either true or false, depending on the body of the rule.

Goals are very similar to rules. Like rules, a goal can evaluate to either true or false, depending on how it is defined. A goal differs from a rule in that it does not have a name and does not take any arguments. Rules, then, are used to establish logical relationships between other facts and rules, while goals are used to ask questions based on the facts and rules.

We have already seen one example of a rule. But what if we wanted a rule to depend on more than one condition? We can add additional conditions, separated by commas. For example:

```
house(NumFloors) :- roof, door, windows, NumFloors is 3.
```

This rule has an argument as well as 4 conditions which must be true for the rule to be true. *Roof*, *door*, and *windows* do not depend on the arguments passed into the rule. However, *NumFloors is 3* depends on the value of *NumFloors*. The statement means exactly what it says; it will evaluate to true if the value of *NumFloors* is the number 3.

If we think a prolog rule as a logical statement, then the commas can be interpreted to signify *and*. The rule could be read: "*house* if *roof* and *door* and *windows* and *NumFloors* is 3).

Since goals are similar to rules, that rule can also be represented as a goal:

```
NumFloors = 5, roof, door, windows, NumFloors is 3
```

This would only work if roof, door, and windows were already defined. Because this is a goal rather than a rule, we had to give a value to NumFloors in our goal (*NumFloors = 5*). This is a trivial example, and it should be obvious why this would evaluate to false.

So, we know how to use *and* in rules and goals. How about *or*? Or can be represented in prolog with a semicolon, and this does exactly what one would expect. For example:

```
warmColor(Color) :-
        Color = red;
        Color = orange;
        Color = yellow.
```

warmColor(red), then, would evaluate to true, while *warmColor(green)* would be false.

Goals and rules, as one can see, form the logical basis for Prolog. But one of the most powerful features of Prolog has not been mentioned. Up until now, we have written statements that evaluate to either true or false. After all, this is the basis of logic. However, supposed we wanted to ask a question without a yes or no answer? It turns out that Prolog can do this, and it does it quite well.

One asks Prolog a true/false question by posing it as a goal, as we have seen. Likewise, open-ended questions are also asked in the form of goals. To make a goal open-ended, all one has to do is include an undefined variable in the goal. For example, suppose we modified our colors program like so:

```
color(red).
color(orange).
color(yellow).
color(green).
color(blue).
color(purple).
warmColor(Color) :-
        color(Color),
        not(Color = green),
        not(Color = blue),
        not(Color = purple).
```

As you might notice, we have (more or less) defined the same program, albeit in a stranger way. We have a list of facts which define valid colors, and we have changed our warmColor rule to evaluate to true if the color is a valid color and is not green, is not blue, and is not purple. It will still only return true for red, orange, and yellow, but we have not explicitly told it this. Now suppose we run the following goal:

```
warmColor(Color).
```

We have an undefined variable (Color) in our goal; therefore we have asked an open ended question. What we have asked is: "For what values of *Color* will *warmColor* evaluate to true?" Prolog, after solving this problem with the SL-Resolution technique described previously, and will answer like so:

```
Color = red;
Color = orange;
Color = yellow.
```

This is quite a remarkable feat, as Prolog has *logically deduced* information from a set of facts and rules. This is the real power of Prolog.

2.3 Literature Review

2.3.1 Introduction to Game Theory

The origination of game theory is attributed to the work of John von Neumann and Oskar Morgenstern in a book titled *The Theory of Games and Economic Behavior.* According to the Stanford Encyclopedia of Philosophy:

Game theory is the study of the ways in which interacting choices *of* economic agents *produce* outcomes *with respect to the* preferences *(or* utilities*) of those agents, where the outcomes in question might have been intended by none of the agents.* [5]

In other words, game theory is the study of how individual people (or some other "economic agent") interact according to their preferences, and how sometimes this interaction can produce an outcome that none of the individuals desired. The word "game" is used not because the theory contemplates merely trivial matters, but because every scenario is analyzed like a game, with a set of rules that apply to the players and a set of objectives for each player. An "economic agent" is an abstract representation of some entity with one or more objectives. This is often a human being, but could also be a computer or another animal [5].

Game theory is heavily focused on the idea of *anticipation*. That is, the actions of one economic agent can be influenced by expectations of the decision another agent will make, and vice versa [5]. For example, consider an army attempting to overtake a walled city. The most apparent method of attack might be to scale the walls. The city, then, *anticipating* this attack, might allocate much of its resources to prevent this type of attack. The invading army, however, might *anticipate* this response, and instead choose to enter via the sewage drains underneath the walls, hoping to catch the defending city off guard. But if the defending city were especially clever, it might also anticipate this move by the invading army and allocate resources to defending the sewage drains. This type of reasoning could go on indefinitely for both sides, which is what makes game theory both complex and interesting.

2.3.2 The Prisoner's Dilemma

The most well-known problem in game theory is called the *Prisoner's Dilemma*. Consider the following scenario: there are 2 prisoners who have been arrested as suspects for a crime. The authorities only have enough evidence to send each suspect to jail for 2 years. However, the authorities are clever. They offer a deal to each prisoner. If one of the prisoners will confess to the crime, that prisoner will be let free, while the other will receive a 10-year sentence. However, if both of the prisoners confess, they will each receive only 5 years.

We see that there are 4 outcomes for each prisoner: the prison sentence will be either 0 years, 2 years, 10 years, or 5 years. If we assume that each prisoner is purely selfish, and that each prisoner's goal is to minimize the time he/she spends in prison, we can rank the 0-year sentence as the most preferable, the 2-year sentence next, the 5-year sentence after that, and the 10-year sentence as the least preferable. The prisoner, then, if he/she is rational, will attempt to make a decision that results in the smallest prison sentence for him/her.

However, the situation is not so simple. The prison sentence for a prisoner will be determined not only by that prisoner's decision, but also by the decision of the other prisoner. In other words, each prisoner must anticipate the other prisoner's decision. Does this sound familiar?

Let us consider the possible outcomes for one of the prisoners. First, the other prisoner might choose to confess. In this case, our prisoner would receive 5 years if he/she also confessed, and 10 years if he/she chose to abstain. In this case, it would be preferable for our prisoner to confess.

The other possibility is that the other prisoner might choose to abstain. In this case, our prisoner would receive 0 years for confessing and 2 years for abstaining. Both scenarios sound fairly good, but confessing would certainly be the preferable option.

So if our prisoner anticipated that the other would confess, our prisoner would also confess. And if our our prisoner anticipated that the other one would abstain, our prisoner would *still* confess. In other words, regardless of what the other prisoner chooses, the best option would be to confess. Thus our prisoner will confess.

What about the other prisoner? Assuming he/she has the same goals and rationality as the first prisoner, he/she will also choose to confess. The result, then, will be that each prisoner will confess and each will receive 5 years in prison. This is an interesting result, because there is another outcome that would have been better for both prisoners: the 2/2 sentence. What happens in the prisoner's dilemma, then, is that even though both players act in their own self-interest, the end result is in neither of their best interests [5].

This is a result that many find very fascinating. To some, it does not make sense. Surely there are comparable situations in real life where both prisoners choose not to confess? The answer to that question is (probably) *yes*.

The prisoner's dilemma is mathematically and logically valid – for the assumptions that it makes. However, in real life, those assumptions may not necessarily be the case. The prisoner's dilemma assumes pure selfishness (no empathy for the other prisoner), that there are no other factors which might influence the prisoner's decision (perhaps the prisoner has a hard life outside of jail, so jail does not seem so bad), and that the prisoner is purely rational (that he/she is capable of the logical reasoning necessary to deduce that confessing is preferable no matter what). Each of these are major assumptions which must be taken into account when attempting to apply the prisoner's dilemma to real-life situations.

2.3.3 Representation of Games

In order to analyze and understand various "games," theorists have devised some techniques for representing games in a uniform manner. One way of representing games is by a matrix.

In a matrix for a 2-player game, the potential actions of one player are represented by rows, while the potential actions of the other player are represented by columns. At the intersection of a particular column and row is an outcome that results from that combination of actions. The outcome can be represented in terms of the utility of that outcome for each player, measured in arbitrary units.

So for the prisoner's dilemma, the matrix in Table 1 could be used.

Table 1: Matrix representation of the prisoner's dilemma

	Partner confesses	**Partner abstains**
Prisoner confesses	-5	0
Prisoner abstains	-10	-2

In this matrix, the numbers in the cells correspond to the *payoff* value of that outcome for the Prisoner represented by the rows [5]. The payoff is simply the negative of the number of years in prison, since the players are assumed to prefer less time in prison. It is worth noting that this is a simplified scheme that does not take into account other potential factors, such as the fact that the payoff might decrease exponentially (rather than linearly) as the prison sentence increases (a prisoner might consider 20 years more than twice as bad as 10 years). For the purposes of this example, though, it will suffice.

The prisoner gets to select the row, but the partner gets to select the column. An observant player will notice that, for each column, the upper row has the highest (or least negative) value. This represents that the fact that each prisoner will be better off confessing, no matter what the other chooses.

Games that can be represented by a matrix are *strategic-form* games [5]. It is strategic-form games and specifically the prisoner's dilemma that I would like to focus on in this paper.

3. METHODOLOGY

3.1 Introduction

As stated previously, it is my goal to determine whether Prolog is a suitable language to represent game theory models. The methodology I am using in this study is straightforward. I will write a Prolog program to represent the prisoner's dilemma as an example of a simple strategic-form game, then analyze the program for completeness, readability, and brevity (approximately how many lines of code are needed). I will then write an equivalent program in an imperative language (in this case Python), analyze it by the same metrics, and then compare the results of the two languages. It is worth nothing that any interpretation of the results of this study will necessarily be subjective. This is due to the subjective nature of the research question itself. I will attempt to give my best analysis of the results, and encourage anyone who is interested to do likewise.

3.2 The Generic Code

Working with Prolog to develop the fundamental parts of the code quickly revealed a need for some basic underlying code to abstract away details in dealing with lists and performing mathematical computations not built into the language. There are no arrays in Prolog, only lists that are traversed recursively. This makes dealing with a matrix unnecessarily tedious, so I wrote some Prolog rules which simulate the behavior of an array for a list and provide convenient ways to manipulate them, such as accessing an element by index. The generic code is placed into the files called *array* and *math*. The generic code was written for Prolog only, as I did not encounter these problems in Python.

3.3 The Matrix Code

3.3.1 Overview

In order to make my code extensible for future projects involving strategic-form games, I attempted to modularize as much as possible. Before getting to the specifics of the prisoner's dilemma, I created code to represent a generic *N* by *M* matrix, where the *N* rows represent the choices available to the player in question, and the *M* columns represent every possible combination of events that are beyond the player's control.

In the prisoner's dilemma, the columns will simply represent the choices available to the partner of the prisoner in question. However, for more complex games, columns could represent any combination of events that are beyond the player's control.

As a trivial example, suppose there were 2 events beyond the control of a player in a 3-person card game: the cards held by the player on the right and the cards held by the player on the left. Both of these events would affect the outcome of the game, so now there are 2 uncontrollable events to consider rather than just 1.

Theoretically, we could represent every possible combination of cards that could be held by each player as a column on the matrix. Of course, this would be a large number even if each player had only one card left. (Assuming a 52-card deck and that the player could eliminate 2 cards as possibilities, there would be 50 x 49 = 2,450 total possibilities to consider.) Though not practical to compute by hand while playing a card game, an analysis of this type of scenario would be possible with a computer program such as the one I am creating.

3.3.2 Likelihood

In the prisoner's dilemma, the decision for each prisoner was straightforward, because it not matter what the other prisoner would choose; confessing would always be preferable. Confessing in this scenario is what game theorists call a *strictly dominant* strategy [5]. It is easy to pick a strictly dominant strategy, but what if there happened to be no strictly dominant strategy? Suppose, instead, that the matrix of the prisoner's dilemma were changed to the matrix in Table 2.

Table 2: Matrix without any strictly dominant strategies

	Partner confesses	**Partner abstains**
Prisoner confesses	-2	-10
Prisoner abstains	-10	-2

Of course, this would no longer be a prisoner's dilemma. But notice that there is no longer a readily apparent choice. If our prisoner picks the first row, he/she will hope that the partner chooses the first column. If our prisoner picks the second row, he/she will hope that the partner picks the second column. If our prisoner has no idea what the partner will do, then the odds are 50 to 50. Our prisoner will just have to randomly pick a row and hope for the best.

However, let's say our prisoner has a reasonable guess (at least in his/her mind) of what the partner will do. Maybe the partner is stubborn, so our prisoner does not believe he/she will talk. In this case, our prisoner would rationally choose to also not talk, because this would increase our prisoner's odds of getting the 2-year sentence rather than the 10-year sentence.

This was my reasoning behind incorporating *likelihood* into my matrix code. Technically, the likelihood is not part of the matrix. It can be represented by a list of likelihoods parallel to the columns of the matrix, with each value corresponding to the likelihood that the player estimates that column will occur.

Again, the likelihood is irrelevant in the prisoner's dilemma (and in any game with a strictly dominant strategy). Even if our player is 99% certain that the partner will refuse to confess, the rational decision will still be to confess and be immediately released from prison.

3.3.3 The Algorithm

I took great care to make the matrix code as generic as possible so that it could be used to represent a wide variety of games. Because it is generic it is also quite simple. It provides, first of all, the ability to create game players and to give them simple attributes for a game. The attributes are the payoff, or utility, matrix for the game, and the list of likelihoods corresponding to each column in the utility matrix.

The matrix code also provides 2 basic methods of analyzing a game from the perspective of a given player. The first is to *rationally evaluate* the options, or the rows of the matrix. This aspect follows a simple formula which sums up the *weighted* values for each cell in a row. The weighted value of a cell is its utility value for the player multiplied by the likelihood for the column that it is in. Thus the more likely a column is believed by a player to occur, the more influential the utility values for that column will be in selecting a row.

Then, the matrix code can select a *rationally preferable* row based on the rational evaluation of the rows. This will simply be the row with the highest evaluated value.

This is all that the matrix code provides. How the likelihood is determined, how the utility is determined, and relating matrices of multiple players in a game will all be dependent on the particular game, and thus is not handled in this code.

3.4 The Prisoner's Dilemma Code

3.4.1 Overview

Finally, once the code for the generic matrix was built, I could build the specific extensions for the prisoner's dilemma. The challenge I was faced with was writing an interesting program based on a scenario that results in the same outcome every single time.

My idea was to challenge one of the assumptions made by the prisoner's dilemma: that each of the players is purely selfish. Of course, this is no longer technically a prisoner's dilemma, but I think it is of some interest how the outcome might differ if one or both of the player's is not purely selfish.

Thus, I decided to implement the concepts of *empathy* and *apathy*. For the purposes of this program, *empathy* is meant to represent the opposite of selfishness; it represents sincere concern of the outcome of the other prisoner. *Apathy*, then, is used to mean selfishness in this program. Thus, an apathy value of 1 or greater, along with an empathy value of 0, for each player, will result in the actual prisoner's dilemma.

To simplify things, I introduced a *precision* variable to represent the level of precision used to measure empathy and apathy. Thus, a precision of 100 would mean that empathy and apathy could have values in the range of 0 to 100. In addition, I implemented the interface to set the empathy and apathy values such that setting one would automatically result in setting the other to its complement in the precision range. So for a precision of 100, setting an empathy value of 30 would automatically set the apathy value to 70. This is intended to simplify analysis, as the empathy and apathy values can always be easily seen as a fraction of the total.

3.4.2 Functionality

The prisoner's dilemma code provides a code interface for creating a matrix to represent the possible prison sentences, for setting *empathy* and *apathy* values for each player, for setting the *precision* of the empathy and apathy values, for dynamically calculating the utility matrix for a player, for getting the outcome of a game between 2 players, as well as some rules/functions for obtaining other potentially interesting information about a player.

The utility matrix is dynamically calculated based on the current apathy/empathy values of the player, the years-in-prison matrix, and the likelihood values of the player. The utility of a cell is calculated by the following formula:

$$U = (A * -1 * Yself) + (E * -1 * Yother)$$

U represents the calculated value of the cell, A represents the player's apathy value, Yself represents the years that the player will go to prison, E represents the empathy value, and Yother represents the years that the other prisoner will go to prison. The -1's represent the fact that longer prison sentences result in smaller utilities. The higher the apathy value for a player, the more the utility value will reflect the amount of time he/she will spend in prison; the higher the apathy value, the more the utility value will reflect the amount of time the other player will spend in prison. In this way the apathy and empathy values achieve their desired goals. The higher the empathy value, the more likely a player will be to make a move that will benefit the other player.

Once the utility matrix has been constructed for a player, the code can, using the generic matrix code, tell us whether the player will confess or refuse. Likewise, a game can be simulated between 2 players, and the code will tell us the prison sentence that each player will receive.

In addition, I created a few more rules/functions that may provide some interesting information. The program can find all empathy or apathy values that will lead to a given decision (confess or refuse). It can also find the minimum and maximum of such values, allowing us to ask questions such as "What is the maximum empathy value this player can have and still rationally choose to confess?"

4. RESULTS

4.1 Overview

I was able to implement the same functionality in both Prolog and Python. To my knowledge, they are both complete in that they both perform the desired functionality. For the most part, there was a direct correlation between a Prolog rule and a Python function. To make comparison as easy as possible, I used the same function/rule and variable names whenever possible. Additionally, both the Prolog and the Python programs are organized in the same manner.

4.2 Brevity

One metric we can use to compare the Prolog and Python programs is the length of the program, which I will measure in line count. Note that I compare only the matrix and prisoner programs; the array and math programs used for the Prolog implementation were ignored, as they do not directly solve the problems specific to this research. If the code were to be expanded, this code would not grow significantly, if any at all. Thus it can be seen as a constant factor and can be safely ignored.

The line count comparison can be seen in Table 3. Note that the numbers are only meant to be approximations.

Table 3: Line Count Comparison

Language	**Matrix Code**	**Prisoner Code**
Prolog	122	263
Python	62	223

We can see that Python clearly used less lines of code than Prolog. There are a fewe reasons for this. First of all, with Python I implemented the game player as a class; of course, there is no such thing in Prolog. Being able to condense the player functions into a single class saved some lines. In the *prisoners* program, the player class was extended, so the same savings were applied again.

Other reasons include the way items in a list are accessed (using my custom rules), the way output is handled in Prolog, and the recursive nature of Prolog, which often requires multiple "overwritten" (if we can use that word) versions of the same rule. These characteristics, while undoubtedly beneficial in certain scenarios, did not lend themselves as naturally to the type of problems I was trying to solve. That being said, the difference is arguably rather small. And in either case, counting the lines of code is not necessarily the best way to evaluate a program.

4.3 Readability

Readability is inherently subjective, so what I consider readable is not necessarily what someone else will consider readable. Thus, the reader is welcome to look at the code for him/herself to come to his/her own conclusions about the readability of the 2 versions of the code.

The readability of both the Prolog and Python programs, was, for the most part, very good in my opinion. Since they were structured almost identically, I think they had very similar levels of readability. The Prolog programs will certainly seem confusing to someone new to the language, but assuming familiarity with the language, most of the Rules were fairly intuitive.

The most confusing parts in the Prolog programs were probably the way items were accessed from a list (using an index as if it were an array) and the rules pertaining to setting/retrieving values for individual players. Both of these issues stem from an absence of data structures (array and class) found in many imperative languages, like Python. The classes, especially, made the code much cleaner and understandable in the Python programs.

4.4 Conclusion

Going into this study, my hypothesis was that the logical structure of Prolog might lend itself especially well to modeling game theory scenarios. I thought perhaps that Prolog might be adept at allowing the user to enter in data for scenario and easily run many queries on the scenario, some which might not have been necessarily built into the program. The idea behind Prolog, after all, is that it can make logical connections to find relationships that it has not been explicitly programmed to know.

What I found is that Prolog did not provide any of this, at least for my limited experiment with the prisoner's dilemma. In fact, for most parts of the code, writing the Prolog version was either just as straightforward or more indirect as the Python equivalent.

There is only one place that I noticed that the Prolog code *almost* worked more intuitively. In the *getApathyForAction* and *getEmpatyForAction* functions/rules, Prolog was able to give me a list of possible solutions without being explicitly told to do so; in Python, I had to give explicit instructions to try all of the possibilities one by one with a loop. There was one catch – I had to restrict the range of possible solutions, because otherwise there would be an infinite number of possibilities to try. I built a *restrict* rule to handle this. Essentially, the *restrict* rule works just like a loop; it fills a list with consecutive integers in a specified range. Thus, in the end even this aspect turned out to work out no better in Prolog than in Python.

Does this mean that Prolog offers no advantage to representing game theory models over a language like Python? Not necessarily. As far as I know, it is entirely possible that Prolog could still prove to be a valuable tool for representing exceedingly complex models, and that my simple prisoner's dilemma game simply did not exploit its strengths. To arrive at a more decisive conclusion, one would need to build a much more complex model.

I also believe that Prolog would have been just as good as Python for this problem if only it had beter data structures, comparable to those of imperative languages. Of course, Prolog was designed for a very specific purpose rather than as a general programming language, so the lack of more data structures is understandable.

5. FUTURE WORK

As suggested in the previous section, in order to arrive at a better conclusion to this research question, more experiments are needed. Specifically, a much more complex game theory model needs to be created in Prolog, and in a comparable imperative language like Python. With more players and more variables in a more complex game, who knows, Prolog may prove to be especially useful still. I wrote my code from the beginning with this possibility in mind, so that perhaps something could be taken from it if one were to continue this study.

6. REFERENCES

[1] R. A. Kowalski, “The Early Years Of Logic Programming,” *Commun. of the ACM*, vol. 31, no. 1, pp. 38-43, Jan. 1988.

[2] A. Colmerauer and P. J. Roussel, “The birth of Prolog,” in *History of Programming Languages*, Cambridge, MA, 1993, pp. 37-52.

[3] J. H. Gallier, “SLD-Resolution And Logic Programming (PROLOG),” in *Logic for Computer Science*, 2nd ed. Mineola: Dover, 2015, ch. 9, pp. 410-447.

[4] M. A. Bramer, *Logic Programming with Prolog*. London: Springer, 2013.

[5] D. Ross. (2014, Dec. 9). *Game Theory* [Online]. Available: http://plato.stanford.edu/entries/game-theory/

Appendix A: Prolog Program

```
%	file: array.pl

/*
	Rule: access
	Description:
		Access item from list by index, like an array.
	Input:
		[] (List): list to access from
		Index (Integer): location in list to acccess from
	Output:
		Element: the content at the specified location
*/
access([Head|List],Index,Element) :-
	Element = Head,
	Index is 0,!;
	access(Element,Index,List,0).

access(Element,Index,[Head|List],Current) :-
	Element = Head,
	Index is Current+1,!;
	access(Element,Index,List,Current+1).

/*
	Rule: addToList
	Description:
		Add an item to the front of a list
	Input:
		Element: Item to insert
		OldList (List): list to be added to
	Output:
		NewList (List): list with the new item
*/
addToList(Element,OldList,NewList) :-
	NewList = [Element|OldList].

/*
	Rule: inToList
	Description:
		Check if an item is in a list
	Input:
		Item: Item to check for
		[] (List): List to be checked
	Output:
		true/false: true if in liist, false otherwise
*/
inList(Item,[L|List]) :-
	Item = L;
	inList(Item,List).

implode([L|List],Separator,Result) :-
	implodeHelp(List,Separator,ResultList),
	concat(L,ResultList,Result).

implodeHelp([L|List],Separator,Result) :-
	implodeHelp(List,Separator,ResultList),
	concat(Separator,L,S),
	concat(S,ResultList,Result).

implodeHelp([],_,'').
```

```
%		file: math.pl

:- [array].

/*
		Rule: restrict
		Description:
				Restrict a variable to a certain range of consecutive integers
		Input:
				Min (Integer): beginning of range (inclusive)
				Max (Integer): end of range (inclusive)
		Output:
				Integer: restricted variable
*/
restrict(Min,Max,Integer) :-
		enumeratedList(Min,Max,List),
		inList(Integer,List).

/*
		Rule: enumeratedList
		Description:
				create a list of consecutive integers
		Input:
				Start (Integer): beginning of range (inclusive)
				Stop (Integer): end of range (inclusive)
		Output:
				List (List): the resulting list
*/
enumeratedList(Start,Stop,List) :-
		enumeratedList(Start,Start,Stop,List).

enumeratedList(Index,Start,Stop,[H|List]) :-
		Index =< Stop,
		H is Index,
		enumeratedList(Index+1,Start,Stop,List),!.

enumeratedList(Index,_,Stop,[]) :-
		Stop is Index -1,!.

/*
		Rule: max
		Description:
				finds the maximum value of a list of integers
		Input:
				[] (List): list to search
		Output:
				Max (Integer): the largest value in the list
*/
max([Max],Max).

max([Data|List],Max) :-
		max(List,MaxList),
		(Data >= MaxList -> Max is Data,!; Max is MaxList,!).

/*
		Rule: indexMax
		Description:
				finds the index of the maximum value of a list of integers
		Input:
				List (List): list to search
		Output:
				Max (Integer): the index of the largest value in the list
*/
indexMax(List,Max) :-
```

```
        indexMax(List,0,_,Max).

indexMax([MaxData],MaxIndex,MaxData,MaxIndex).

indexMax([Data|List],Index,MaxData,MaxIndex) :-
        NextIndex is Index + 1,
        indexMax(List,NextIndex,ListMaxData,ListMaxIndex),
        (
                Data >= ListMaxData -> MaxIndex is Index,MaxData is Data,!;
                MaxIndex is ListMaxIndex, MaxData is ListMaxData,!
        ).
```

```
%		file: matrix.pl

:- [math].

/*
		Rule: setLikelihood
		Description:
				set the likelihood value for a player
		Input:
				Player (String): identifier for the player
				Likelihood (List): new value
*/
setLikelihood(Player,Likelihood) :-
		setProp(Player,'likelihood',Likelihood).

/*
		Rule: getLikelihood
		Description:
				get the likelihood value for a player
		Input:
				Player (String): identifier for the player
		Output:
				Likelihood (List): current value
*/
getLikelihood(Player,Likelihood) :-
		getProp(Player,'likelihood',Likelihood).

/*
		Rule: setUtility
		Description:
				set the utility value for a player
		Input:
				Player (String): identifier for the player
				Utility (List): new value
*/
setUtility(Player,Utility) :-
		setProp(Player,'utility',Utility).

/*
		Rule: getUtility
		Description:
				get the utility value for a player
		Input:
				Player (String): identifier for the player
		Output:
				Utility (List): current value
*/
getUtility(Player,Utility) :-
		getProp(Player,'utility',Utility).

/*
		Rule: setProp
		Description:
				helper for setting a property for a player
		Input:
				Player (String): identifier for the player
				Property (String): identifier for the property
				Value: new value
*/
setProp(Player,Property,Value) :-
		implode([Player,Property],'.',Name),
		nb_setval(Name,Value).

/*
```

```
	Rule: getProp
	Description:
		helper for getting a property for a player
	Input:
		Player (String): identifier for the player
		Property (String): identifier for the property
	Output:
		Value: current value
*/
getProp(Player,Property,Value) :-
	implode([Player,Property],'.',Name),
	nb_getval(Name,Value).

/*
	Rule: preferRational
	Description:
		determine the best course of action assuming pure rationality
	Input:
		Player (String): identifier for the player
	Output:
		Choice (Integer): index of the chosen row of the matrix
*/
preferRational(Player,Choice) :-
	getLikelihood(Player,Likelihood),
	getUtility(Player,Utility),
	evalRational(Utility,Likelihood,WeightedUtility),
	indexMax(WeightedUtility,Choice).

/*
	Rule: evalRational
	Description:
		evaluate the available options, assuming pure rationality
	Input:
		[] (List): the utility matrix
		Likelihood (List): list of weights representing how likely the player thinks each column of the matrix will occur
	Output:
		[] (List): weighted utility matrix, taking into account both original utility values and likelihood
*/
evalRational([U|Utility],Likelihood,[WU|WeightedUtility]) :-
	evalRational(Utility,Likelihood,WeightedUtility),
	weightedSum(U,Likelihood,Sum),
	WU is Sum.

evalRational([],_,[]).

/*
	Rule: weightedSum
	Description:
		sum a list of integers, each integer multiplied by its corresponding value in a list of weights
	Input:
		[] (List): list of integers
		[] (List): list of weights
	Output:
		Sum (Integer): the weighted sum
*/
weightedSum([A|Addends],[W|Weights],Sum) :-
	weightedSum(Addends,Weights,SumBelow),
	Sum is A * W + SumBelow.

WeightedSum([],[],0).
```

```
%		file: prisoners.pl

:- [matrix].

/*
		Rule: setPrecision
		Description:
				set the precision value for the game
		Input:
				Precision (Integer): new value
*/
setPrecision(Precision) :-
		nb_setval(precision,Precision).

/*
		Rule: getPrecision
		Description:
				get the precision value for the game
		Output:
				Precision (Integer): current value
*/
getPrecision(Precision) :-
		nb_getval(precision,Precision).

/*
		Rule: setYearsInPrison
		Description:
				set the yearsInPrison value for the game
		Input:
				YearsInPrison (List): new matrix
*/
setYearsInPrison(YearsInPrison) :-
		nb_setval(yearsInPrison,YearsInPrison).

/*
		Rule: getYearsInPrison
		Description:
				get the yearsInPrison value for the game
		Output:
				YearsInPrison (List): current matrix
*/
getYearsInPrison(YearsInPrison) :-
		nb_getval(yearsInPrison,YearsInPrison).

/*
		Rule: setEmpathy
		Description:
				set the empathy value for a player; also implicitly sets the apathy value to its inverse
		Input:
				Player (String): identifier for the player
				Empathy: new value
*/
setEmpathy(Player,Empathy) :-
		getPrecision(Precision),
		setProp(Player,'empathy',Empathy),
		Apathy is Precision - Empathy,
		setProp(Player,'apathy',Apathy).

/*
		Rule: getEmpathy
		Description:
				get the empathy value for a player
		Input:
				Player (String): identifier for the player
```

```
	Output:
		Empathy: current value
*/
getEmpathy(Player,Empathy) :-
	getProp(Player,'empathy',Empathy).

/*
	Rule: setApathy
	Description:
		set the apathy value for a player; also implicitly sets the empathy value to its inverse
	Input:
		Player (String): identifier for the player
		Apathy: new value
*/
setApathy(Player,Apathy) :-
	getPrecision(Precision),
	setProp(Player,'apathy',Apathy),
	Empathy is Precision - Apathy,
	setProp(Player,'empathy',Empathy).

/*
	Rule: getApathy
	Description:
		get the apathy value for a player
	Input:
		Player (String): identifier for the player
	Output:
		Apathy: current value
*/
getApathy(Player,Apathy) :-
	getProp(Player,'apathy',Apathy).

/*
	Rule: preferRationalPD
	Description:
		Maps referRational result 0 to 'confess', and result 1 to 'refuse'
	Input:
		Player (String): identifier for the player
	Output:
		Action (String): 'confess' or 'refuse'
*/
preferRationalPD(Player,Action) :-
	deriveUtility(Player),
	preferRational(Player,0),
	Action = 'confess',!;
	preferRational(Player,1),
	Action = 'refuse',!.

/*
	Rule: deriveUtility
	Description:
		Sets a utility matrix based on empathy, apathy, and yearsInPrison matrix
	Input:
		Player (String): identifier for the player
*/
deriveUtility(Player) :-
	getEmpathy(Player,Empathy),
	getApathy(Player,Apathy),
	getYearsInPrison(Years),
	access(Years,0,A),
	access(A,0,A1),
	access(A,1,A2),
	access(Years,1,B),
	access(B,0,B1),
        access(B,1,B2),
        W is A1 * -1 * Apathy + A1 * -1 * Empathy,
        X is A2 * -1 * Apathy + B1 * -1 * Empathy,
        Y is B1 * -1 * Apathy + A2 * -1 * Empathy,
        Z is B2 * -1 * Apathy + B2 * -1 * Empathy,
        Utility =
        [
                [W,X],
                [Y,Z]
        ],
        setUtility(Player,Utility).

/*
        Rule: getResults
        Description:
                Get the result of a prisoner's dilemma between 2 players
        Input:
                Player1 (String): identifier for player 1
                Player2 (String): identifier for player 2
        Output:
                Player1Sentence (Integer): prison sentence for player 1
                Player2Sentence (Integer): prison sentence for player 2
*/
getResults(Player1,Player2,Player1Sentence,Player2Sentence) :-
        deriveUtility(Player1),
        deriveUtility(Player2),
        preferRational(Player1,Choice1),
        preferRational(Player2,Choice2),
        getYearsInPrison(YearsInPrison),
        access(YearsInPrison,Choice1,Player1Row),
        access(Player1Row,Choice2,Player1Sentence),
        access(YearsInPrison,Choice2,Player2Row),
        access(Player2Row,Choice1,Player2Sentence).

/*
        Rule: getApathyForAction
        Description:
                find all valid apathy values that would cause the specified action
        Input:
                Player (String): identifier for the player
                Action (String): the action that would result
        Output:
                Apathy (Integer): a valid apathy value
*/
getApathyForAction(Player,Action,Apathy) :-
        getApathy(Player,InitialApathy),
        findall(A,getApathyForActionHelp(Player,Action,A),Results),
        inList(Apathy,Results),
        setApathy(Player,InitialApathy).

getApathyForActionHelp(Player,Action,Apathy) :-
        getPrecision(Precision),
        restrict(0,Precision,Apathy),
        setApathy(Player,Apathy),
        deriveUtility(Player),
        preferRationalPD(Player,Action).

/*
        Rule: getEmpathyForAction
        Description:
                find all valid empathy values that would cause the specified action
        Input:
                Player (String): identifier for the player
                Action (String): the action that would result
```

```
	Output:
		Empathy (Integer): a valid empathy value
*/
getEmpathyForAction(Player,Action,Empathy) :-
	getEmpathy(Player,InitialEmpathy),
	findall(E,getEmpathyForActionHelp(Player,Action,E),Results),
	inList(Empathy,Results),
	setEmpathy(Player,InitialEmpathy).

getEmpathyForActionHelp(Player,Action,Empathy) :-
	getPrecision(Precision),
	restrict(0,Precision,Empathy),
	setEmpathy(Player,Empathy),
	deriveUtility(Player),
	preferRationalPD(Player,Action).

/*
	Rule: maxApathy
	Description:
		select the maximum apathy value for a given action
	Input:
		Player (String): identifier for the player
		Action (String): the action that would result
	Output:
		Apathy (Integer): the maximum apathy value
*/
maxApathy(Player,Action,Apathy) :-
	findall(Apathy,getApathyForAction(Player,Action,Apathy),Results), %clarify
	max(Results,Apathy).

/*
	Rule: minApathy
	Description:
		select the minimum apathy value for a given action
	Input:
		Player (String): identifier for the player
		Action (String): the action that would result
	Output:
		Apathy (Integer): the minimum apathy value
*/
minApathy(Player,Action,Apathy) :-
	nb_getval(precision,Precision),
	maxEmpathy(Player,Action,Empathy),
	Apathy is Precision - Empathy.

/*
	Rule: maxEmpathy
	Description:
		select the maximum empathy value for a given action
	Input:
		Player (String): identifier for the player
		Action (String): the action that would result
	Output:
		Empathy (Integer): the maximum empathy value
*/
maxEmpathy(Player,Action,Empathy) :-
	findall(Empathy,getEmpathyForAction(Player,Action,Empathy),Results),
	max(Results,Empathy).

/*
	Rule: minEmpathy
	Description:
		select the minimum empathy value for a given action
	Input:
```

```
                        Player (String): identifier for the player
                        Action (String): the action that would result
            Output:
                        Empathy (Integer): the minimum empathy value
*/
minEmpathy(Player,Action,Empathy) :-
            nb_getval(precision,Precision),
            maxApathy(Player,Action,Apathy),
            Empathy is Precision – Apathy.
```

```
%	file: scenario.pl

:- [prisoners].

/*
	The number of years in prison for each combination of choices made by the 2 prisoners
	First in row or col: prisoner confesses
	Second in row or col: prisoner abstains
*/
:- setYearsInPrison(
	[
		[5,0],
		[10,2]
	]
).

/*
	Arbitrary maximum for empathy/apathy values; higher number gives higher precision
*/
:- setPrecision(100).

/*
	Create some players and assign them some values
*/
:- setLikelihood('Tyler',[1,1]).

:- setEmpathy('Tyler',30).

:- setLikelihood('Ben',[1,2]).

:- setEmpathy('Ben',30).
```

Appendix B: Python Program

```
#	file: matrix.py

'''
	Class: player
	Description:
		representation of a player in a game
'''
class player:
	def __init__(self,Utility=None,Likelihood=None):
		self.Likelihood = Likelihood
		self.Utility = Utility
	def setUtility(self,Utility):
		self.Utility = Utility
	def getUtility(self):
		return self.Utility
	def setLikelihood(self,Likelihood):
		self.Likelihood = Likelihood
	def getLikelihood(self):
		return self.Likelihood

'''
	Function: preferRational
	Description:
		determine the best course of action assuming pure rationality
	Input:
		Player (player): the player
	Output:
		return (Number): index of the chosen row of the matrix
'''
def preferRational(Player):
	WeightedUtility = evalRational(Player.getUtility(),Player.getLikelihood())
	return WeightedUtility.index(max(WeightedUtility))

'''
	Function: evalRational
	Description:
		evaluate the available options, assuming pure rationality
	Input:
		Utilities (List): the utility matrix
		Likelihood (List): list of weights representing how likely the player thinks each column of the matrix will occur
	Output:
		return (List): weighted utility matrix, taking into account both original utility values and likelihood
'''
def evalRational(Utilities,Likelihood):
	WeightedUtilities = []
	for Utility in Utilities:
		WeightedUtilities.append(weightedSum(Utility,Likelihood))
	return WeightedUtilities

'''
	Function: weightedSum
	Description:
		sum a list of integers, each integer multiplied by its corresponding value in a list of weights
	Input:
		Addends (List): list of integers
		Weights (List): list of weights
	Output:
		return (Number): the weighted sum
'''
def weightedSum(Addends,Weights):
	Sum = 0
```

```
for A,Addend in enumerate(Addends):
        Sum += (Addend * Weights[A])
return Sum
```

```
#		file: prisoners.py

from matrix import *

#Declare global variables
precision = None
yearsInPrison = None

'''
		Function: setPrecision
		Description:
				set the precision value for the game
		Input:
				Precision (Number): new value
'''
def setPrecision(Precision):
		global precision
		precision = Precision

'''
		Function: getPrecision
		Description:
				get the precision value for the game
		Output:
				return (Number): current value
'''
def getPrecision(Precision):
		global precision
		return precision

'''
		Function: setYearsInPrison
		Description:
				set the yearsInPrison value for the game
		Input:
				YearsInPrison (List): new matrix
'''
def setYearsInPrison(YearsInPrison):
		global yearsInPrison
		yearsInPrison = YearsInPrison

'''
		Function: getYearsInPrison
		Description:
				get the yearsInPrison value for the game
		Output:
				return (List): current matrix
'''
def getYearsInPrison(YearsInPrison):
		global yearsInPrison
		return yearsInPrison

'''
		Class: prisoner
		Description:
				representation of a player in a game; inherits from player
'''
class prisoner(player):
		def setEmpathy(self,Empathy):
				global precision
				self.Empathy = Empathy
				self.Apathy = precision - Empathy
		def getEmpathy(self):
				return self.Empathy
```

```
	def setApathy(self,Apathy):
		global precision
		self.Apathy = Apathy
		self.Empathy = precision - Apathy
	def getApathy(self):
		return self.Apathy

'''
	Function: preferRationalPD
	Description:
		Maps referRational result 0 to 'confess', and result 1 to 'refuse'
	Input:
		Player (prisoner): the player
	Output:
		return (String): 'confess' or 'refuse'
'''
def preferRationalPD(Player):
	deriveUtility(Player)
	if preferRational(Player):
		Action = 'refuse'
	else:
		Action = 'confess'
	return Action

'''
	Function: deriveUtility
	Description:
		Sets a utility matrix based on empathy, apathy, and yearsInPrison matrix
	Input:
		Player (prisoner): the player
'''
def deriveUtility(Player):
	global yearsInPrison
	Apathy = Player.getApathy()
	Empathy = Player.getEmpathy()
	Years = yearsInPrison
	Utility = []
	Utility.append([])
	Utility.append([])
	Utility[0].append(Years[0][0] * -1 * Apathy	+	Years[0][0] * -1 * Empathy)
	Utility[0].append(Years[0][1] * -1 * Apathy	+	Years[1][0] * -1 * Empathy)
	Utility[1].append(Years[1][0] * -1 * Apathy	+	Years[0][1] * -1 * Empathy)
	Utility[1].append(Years[1][1] * -1 * Apathy	+	Years[1][1] * -1 * Empathy)
	Player.setUtility(Utility)

'''
	Function: getResults
	Description:
		Get the result of a prisoner's dilemma between 2 players
	Input:
		Player1 (prisoner): player 1
		Player2 (prisoner): player 2
	Output:
		return (Dictionary): prison sentences for player 1 and player 2
'''
def getResults(Player1,Player2):
	results = {}
	deriveUtility(Player1)
	deriveUtility(Player2)
	Choice1 = preferRational(Player1)
	Choice2 = preferRational(Player2)
	global yearsInPrison
	results['Player1'] = yearsInPrison[Choice1][Choice2]
	results['Player2'] = yearsInPrison[Choice2][Choice1]
```

```
		return results

'''
		Function: getApathyForAction
		Description:
				find all valid apathy values that would cause the specified action
		Input:
				Player (prisoner): the player
				Action (String): the action that would result
		Output:
				return (Number): a valid apathy value
'''
def getApathyForAction(Player,Action):
		global precision
		results = []
		initialApathy = Player.getApathy()
		for i in range(0,precision+1):
				Player.setApathy(i)
				deriveUtility(Player)
				if preferRationalPD(Player) == Action:
						results.append(i)
		Player.setApathy(initialApathy)
		return results

'''
		Function: getEmpathyForAction
		Description:
				find all valid empathy values that would cause the specified action
		Input:
				Player (prisoner): the player
				Action (String): the action that would result
		Output:
				return (Number): a valid empathy value
'''
def getEmpathyForAction(Player,Action):
		global precision
		results = []
		initialEmpathy = Player.getEmpathy()
		for i in range(0,precision+1):
				Player.setEmpathy(i)
				deriveUtility(Player)
				if preferRationalPD(Player) == Action:
						results.append(i)
		Player.setEmpathy(initialEmpathy)
		return results

'''
		Function: maxApathy
		Description:
				select the maximum apathy value for a given action
		Input:
				Player (prisoner): the player
				Action (String): the action that would result
		Output:
				return (Integer): the maximum apathy value
'''
def maxApathy(Player,Action):
		return max(getApathyForAction(Player,Action))

'''
		Function: minApathy
		Description:
				select the minimum apathy value for a given action
		Input:
```

```
		Player (prisoner): the player
		Action (String): the action that would result
	Output:
		return (Integer): the minimum apathy value
'''
def minApathy(Player,Action):
	return min(getApathyForAction(Player,Action))

'''
	Function: maxEmpathy
	Description:
		select the maximum empathy value for a given action
	Input:
		Player (prisoner): identifier for the player
		Action (String): the action that would result
	Output:
		return (Integer): the maximum empathy value
'''
def maxEmpathy(Player,Action):
	return max(getEmpathyForAction(Player,Action))

'''
	Function: minEmpathy
	Description:
		select the minimum empathy value for a given action
	Input:
		Player (prisoner): identifier for the player
		Action (String): the action that would result
	Output:
		return (Integer): the minimum empathy value
'''
def minEmpathy(Player,Action):
	return min(getEmpathyForAction(Player,Action))
```

```
#	file: scenario.py

from prisoners import *

'''
	The number of years in prison for each combination of choices made by the 2 prisoners
	First in row or col: prisoner confesses
	Second in row or col: prisoner abstains
'''
setYearsInPrison(
	[
		[5,0],
		[10,2]
	]
)

'''
	Arbitrary maximum for empathy/apathy values; higher number gives higher precision
'''
setPrecision(100)

'''
	Create some players and assign them some values
'''
Tyler = prisoner([],[])

Tyler.setLikelihood([1,1])

Tyler.setEmpathy(30)

Ben = prisoner([],[])

Ben.setLikelihood([1,2])

Ben.setEmpathy(30)
```