CMSC 671 (AI), Fall 2017
Group Project Description

Please read this carefully and start thinking about the steps involved in designing and implementing your player. The project requirements are subject to changes and improvements (but hopefully not too many). Good luck, and let the science begin!

The Game of New Eleusis

New Eleusis is a game of logical induction, in which the players try to work out an overarching 'rule' that defines whether a card is legal to play. It is a simulation of scientific research: The general idea is that the dealer (in the role of "God" or "Nature") thinks up a rule that governs the correct play of the cards, and the other players ("Scientists") take turns playing cards to test hypotheses, racing to see who can come up with a good theory about the rule.[1]

For the project, you are going to write a New Eleusis player that is capable of generating hypotheses, coming up with tests for those hypotheses, implementing those tests, and modifying the rule(s) until it is ready to declare success. Simplifications to the game include reducing the space of possible rules and reducing the inter-player interaction drastically (in fact, the first phase of the project will be single-player).

Project Requirements and Grading

There are four graded components of the project: a project design (20%), your implemented system in two phases (30% each), and a final writeup (20%). You will be graded on the thoughtfulness and clarity of your design and presentation, and not primarily on your algorithm's performance. This is gives you the freedom to try a risky approach that is interesting from a design perspective but might not work very well. An approach that doesn't work very well, and is *also* naïve, trivial, or not well-motivated, will not receive a good grade.

Don't just hack something together—this is an AI class! You should think about the strategies you want to implement; what AI methods would be appropriate for such strategies; and how those AI methods can be adapted for this purpose.

Project Design (20%)

You must submit a project design via Blackboard. Only one project design should be submitted per group, and all members should have input into the project design. (Remember, submitting something with someone's name on it when they did not in fact work on it is a form of academic dishonesty—and besides, you're going to be working on it; you should want input!)

---

[1] *Eleusis was invented by Martin Gardner in his Mathematical Games column in the June 1959 Scientific American, and subsequently modified by a number of people. The usual (unsimplified) version is fully described at http://matuszek.org/eleusis0.html, © David Matuszek 1994.*

Your project design should contain:

1. The team name and names of all participating team members.
2. A short (≤ 500 words) description of your strategy for Phase I, written in clear and understandable English.
   - Your player will have to look at a sequence of cards being played, form one or more hypothes(es), play cards specifically intended to test those hypotheses, refine hypothes(es), and ultimately express a rule that describes a hypothesis that it believes to be correct.
3. A short (≤ 500 words) description of your strategy for Phase II, written in clear and understandable English.
4. A discussion of your strategies and how you developed them (by playing the game, by reading articles, by talking about them, by trying things out and experimenting, …). This discussion should talk about Phase I *and* II players.
5. How your implemented system draws on ideas from the AI literature.
   - This should be focused on material and concepts that we covered in class.
   - Cite references for the AI concepts you mention (when citing the textbook; include section numbers).
   - If you like, you may also discuss methods that you would/could use but don't expect to implement within the scope of the semester.
6. Your evaluation strategy. How will you test your system to see if it's working? How will you quantify how well it works? This will be part of the final writeup.
   - Examples of possible evaluation strategies: Have your player play against itself; play against human(s); work with another team to run your players against each other; come up with an evaluation set *before you start testing* that *demonstrably* covers a lot of the search space .
7. A corresponding Python design:
   - Main functions (including required functions).
   - Expected inputs and outputs for each function.
   - Clear, readable pseudocode for the behavior of the function.
   - Think about what helper functions and computations you will need in order to implement your strategies, and have corresponding function stubs.

It is fine (and likely) to make changes to the design after you submit this document.

Rules and Simplifications

As you know, New Eleusis can be difficult, partly because there are so many possible possible rules, and partly because the scoring and game play are complicated. The version you will implement is very simplified, so please be sure to read carefully to understand what you do and don't have to do.

Game play

You *do not* need to support prophets, sudden death, playing multiple cards at once, or no-plays. Essentially, the purpose of this project is to build a good inductive solver, not to deal with game

mechanics. Your player's final score will consist of how many plays it took to figure out the rule, with penalties for playing wrong cards. (See Phase I and Phase II for scoring details.)

Legal rules

Rules must be expressed with the functions provided in new_eleusis.py.

In general, a rule decides whether a card is legal to play 'next' – that is, on the player's current turn. This might depend only on the card being played (e.g., "all red cards must be even, all black cards must be odd"), or it may depend on previous cards (e.g., "a heart must be followed by a spade"). Rules can be treated as expressions that evaluate to **True** (legal) or **False** (illegal).

Rules can only depend on the current card and, at most, the two previous cards.

- In practice, this means you are should maintain the cards **prev**, **prev2** (the card before last), and **current** (the card you are choosing to play), and rules will only refer to those cards.
- It is okay (and often necessary) to look at all previous plays for information.

Card characteristics

Rules must be expressed with the functions provided in new_eleusis.py, which means they can only depend on card characteristics found in that file.

- **Suits** (diamond ♦, heart ♥, spade ♠, club ♣).
- **Royal card** (King, Queen, Jack) or not.[2]
- **Even or odd** value.
- **Numeric** value (Ace=1, Jack=11, Queen=12, King=13).
- **Higher or lower** deck value.

A note on value. Suits in a standard playing deck are ordered, which creates a value ordering for all cards. The ordering is ♣ < ♦ < ♥ < ♠. This means, for example, that the king of clubs is lower *value* than the two of diamonds.

Phase I (30%)

Your Phase I player will *not* be graded primarily based on score (although obviously, a solver that doesn't solve problems is probably not great). We will also grade based on the clarity and elegance of your code, the correctness of your approach, and how well your strategy captures appropriate AI concepts in trying to solve the problem, as well as a short description of what you implemented.

Your submission should not include any I/O in the required functions. Helper functions that perform I/O can be included but should not be called from the required functions.

In the first phase of the project, the following simplifications will apply.

- Your player can play any card (assume an infinite hand).
- Your player is the only player—it is making all plays, legal and illegal.
- The rule returned by your scientist must describe all cards played so far, which must include a minimum of 20 plays. (See scoring, below.)

[2] Also called "face cards."

- After 200 plays, the player must return its best guess at a rule.

Functions

You must implement the following functions:

setRule(<rule-expression>): Set the current rule, using functions provided in new_eleusis.py.

rule(): Return the current (actual) rule.

boardState(): Returns the formal representation of all plays so far as a sequential list of tuples, in order of play. Each tuple will contain a card played in the main sequence (that is, played successfully), then a list of all cards played unsuccessfully after it, which may be empty.

play(<card>): Play a single card and return **True** or **False** for legal or illegal plays. This is probably a good place to update **boardState**.

scientist(): This function returns the rule your player has found. It is responsible for the inductive task, that is, figuring out the rule. When called, this function is responsible for making plays, considering the information gained, dealing with hypotheses*, and choosing when (after 20+ plays) to declare success and return a rule.

score(): Returns the score for the most recent round. (Low is better!) Calculate by adding points as follows: +1 for every successful play over 20 and under 200; +2 for every failed play; +15 for a rule that is not equivalent to the correct rule; +30 for a rule that does not describe all cards on the board.

Examples

Rules:

No royal cards, everything else is legal:
notf(is_royal(current))

A card must not be an even heart:
notf(andf(even(current), equal(suit(current), H)))

Black must be followed by red:
iff(equal(color(prev), B), equal(color(current), R), False)) [*remember 'iff' is 'if' in our parser*]

Must either change suit or go higher in the same suit:
iff(equal(suit(current), suit(prev)), greater(value(current), value(prev)), False)

Black cards cannot be odd, except royal (face) cards:
iff(orf(equal(color(current), B), orf(is_royal(current), even(current))))

Board state example:

Black must be followed by red:	← rule
10♠, 3♥, 6♣, 6♥, 7♦, 9♣, . . .	← main line
K♠, A♠	← failed plays
9♣	← failed play
[('10S', []) ('3H', []) ('6C', ['KS', '9C']), ('6H', []), ('7D',[]), ('9C', ['AS']), ...]	← board state

*A Note on Hypotheses

How you manage hypotheses will depend completely on your solving strategy. A few things to consider (and discuss in your design document):

- How many hypotheses will you maintain at a time? (One, ten, many?) When do you prune the hypothesis space?
- How do you determine the best play for testing a hypothesis?
- Will your hypotheses be fully bound (that is, 1 hypothesis = 1 complete rule), or will you have partial hypotheses?
- When and how do you update, refine, or delete hypotheses?
- How will you decide when a hypothesis is correct (or rather, when to return it for scoring) in Phase II?

In addition, for debugging purposes, it is extremely likely that you (and potentially we) will find it useful to know what hypothesis or hypotheses the system is considering. You will almost certainly want a function that returns this information in some form.

Phase II (30%)

For Phase II, the following complexities will apply:

- Your player will no longer be the only one; you will have to analyze and account for plays by other players. Other players may end the game by solving the problem, so efficient solving will be necessary for good scores.
- Your player will have a hand of 14 cards, which will limit its possible plays.
- It may be necessary to give a "best guess" rule at <u>any</u> <u>time</u>, which will be scored based on how far it diverges from the true rule. You may also be asked for some measure of how confident your player is in its hypothesis or hypotheses.

Adversaries

When entering a game, your scientist will have between 1 and 3 adversaries. Each adversary will play after you play, going cyclically (e.g., the order of play with two adversaries will be [player, *adv*1, *adv*2, player, *adv*1, *adv*2, ...]). At any point, any player (including your scientist!) may announce that they are ready to guess a rule. At this point, **the game ends,** and all players must return a best guess at the current rule. The score function will be modified as shown below.

For testing purposes, we will provide an Adversary class that plays (fairly) randomly; this will allow you to test your scientist against other players (although not very good ones). You must use objects from this class as adversaries. You are welcome to change the adversary's behavior to make the game harder for testing if you like; we will use a more sophisticated adversary during grading.

Calculating Scores

Your code will need to modify scoring as follows:

score(player): Returns the score for the selected player's most recent round. (***Low is better!***)

- **Add** points as follows:
  - +1 for every successful play over 20 cards and under 200 cards; +2 for every unsuccessful play; +15 for a rule that is not equivalent to the correct rule; +30 for a rule that does not describe all cards on the board.
- **Subtract** points as follows:
  - Each player that guesses the correct rule (see below) with few or no extra terms, receives an additional bonus of -75 points.
  - If the player that ended the game gives the correct rule, it receives an additional -25 points.

No other players receive points, that is, there is no "partial credit."

A Note on Rule Equivalence

In general, determining whether two logical statements are equivalent is an NP-hard[3] problem. That gets intractable quickly! Here are three possibilities for logical equivalence:

1. Calculate exact equivalence for rules directly by using a general theorem proving method or heuristics specific to our cards problem. This will only work for certain rules.
2. Evaluate all combinations of three cards for both rules: 52^3 (140,608) operations. Pretty feasible, especially if combined with heuristics. Because our rules cover at most the previous two cards, this is *exact*.
3. Run some large or very large number of cards (selected randomly or otherwise) against both rules and see if they perform the same. This is *approximate*; its accuracy depends on how many cards you play and asymptotically approaches exactness.

Hand Limits

Your scientist must now maintain a hand of 14 cards at all times, drawn randomly from all possible cards. In practice, this means that you must *randomly select* 14 cards (from infinite deck) at the beginning of the game. Each time your scientist plays a card, you must add a new, *randomly selected* one to the 13 still available to you. You may not discard cards or "choose" cards non-randomly, and all plays must come from the 14 currently in the hand.

Suggestions

You will probably get the best results considering the following suggestions:

---

[3] Specifically, it is in a class called Co-NP-complete, meaning it's often easy to find negative counterexamples, but for positive examples you must solve the entire problem.

- Concentrate on improving your solver, not on game theory. Since you don't know what type of adversaries you may encounter, and they aren't guaranteed to be the same from game to game, the only demonstrably good strategy—not to mention the most interesting problem—is to try to get your scientist to return the correct rule as fast as possible.
- If you aren't considering your adversaries' plays, you're losing out a significant source of information. How can you take advantage of plays that weren't selected by your scientist to further prune your search space?

Writeup (20%)

Each team must submit a 6-8 page (3000-4000 words, not counting references and citations) project report. It should cover both your Phase I and Phase II, and describe your approach, your experience in designing and implementing the approach, and the evaluated performance of your system.

1. **A discussion of your final, implemented game strategies** and how you developed them. This discussion should have a section on your Ph**deas**ase I player and a separate section on your Phase II player, and can discuss what changes you made in between.
2. **A discussion of how your system draws on i from the AI literature.** This discussion should be focused on concepts *from class.* If you use other AI sources, please be very clear about what they are and consider checking with me to make sure they are suitably AI-"ish."
    - You may want to talk about better methods that you weren't able to implement. If you ended up with something relatively simple, but had some ambitious/interesting ideas that you weren't able to get working in the scope of the semester, you should talk about it here.
3. **References for the AI concepts you mention** (it's OK to cite the textbook, but please include specific section numbers).
4. **Experimental evaluation of your system.** How did you test it to see if it works? How well did it do? This section should discuss what worked well, and not so well, about your player's strategies; and whether the player behaves as expected. Both qualitative/anecdotal and quantitative data should be included in this discussion; this is where graphs, multiple-trial experiments, etc. go.

A Note on Cooperation

Teams should not share solutions and must follow the usual rules about sharing code, e.g., you should never have copies of or be using someone else's code or ideas except for occasional help debugging is okay. In this case, that means "Someone who is not on your team" – teams can and should be working together closely.

However, you may find that you have some good ideas about utility functions (tracing your player's behavior, analyzing game state, printing hypotheses, testing rules for equivalence...). You may think that some of this code could be useful for other students. You're probably right, and that's great! Please feel free to send the TA and professor any code that you develop that might be generally useful; we will vet it against the academic integrity policy and post it into a code repository. (If you're having trouble with class participation, this is a great way to contribute.)