CS1571 Introduction to AI

Assignment 2

NOTES TO GRADER:

- Duplicates in my solveEquation() function cause it to **not** find solutions
    - Ex: x=2+2
    - Ex: 2x+2x=4
- I force the solution to be in the form of variables on left, and constants on right
    - This is not always most efficient
- There is an optional parameter called **debug** for all 4 functions, when set to "True" you can see all the additions to KB, parsing of the equations, and other useful info
- I created test cases for all 4 functions that can be uncommented, ready to use
- ☺

In this assignment, you will be implementing a Knowledge-Based Agent for teaching equation solving. You will implement three components of this agent:

1. A feedback system based on propositional logic that delivers different feedback to students based on a variety of criteria.
2. A planner based on first-order logic for solving the problems like a student would.
3. An assessment system based on first-order logic to assess what students know and don't know, and consequently what problems they are capable of solving.

A complete solution to this assignment should be able to A) Compare an action a student takes on a problem to a model of correct performance, B) Provide feedback to that student based on that action, C) Update a model for what students know and don't know based on the student actions, and D) Use that model to make a prediction about whether students would be able to solve a different problem.

The goals of this assignment are to have you use the course concepts involved in this section as part of a real-world application.

We have provided you with a shell Python file including the method calls you require for all parts. You are free to use the code provided as part of the textbook, but it is not required.

Task Domain

The following are examples of possible equations to be solved by the students:

x=3
-3x=6
3x-2=-6
4+3x=6x-7

As a few general rules:

- You can assume that there will be either one or two terms on each side in the initial problem state
- A term can be a variable term (e.g., 3x) or a constant term (e.g., 3)
- Variable terms can have coefficients (the coefficient of 3x is 3)
- All inputs will have an equal sign, and in all inputs the student will be solving for *x*
- Both negative or positive numbers variable and constant terms are allowed

To solve the problem, you can execute three different types of actions:

- **Add:** adding a positive or negative term to both sides
- **Divide:** dividing both sides by a number
- **Combine:** combining two variable terms or constant terms on a single side

Thus the following skills are involved in solving these problems:

- **S1:** add a positive variable term to both sides (e.g., add 3x to both sides)
- **S2:** add a negative variable term to both sides (e.g., add -3x to both sides)
- **S3:** add a positive constant term to both sides (e.g., add +3 to both sides)
- **S4:** add a negative constant term from both sides (e.g., add -3 to both sides)
- **S5:** divide both sides by a positive constant (e.g., divide by 3)
- **S6:** divide both sides by a negative constant (e.g., divide by -3)
- **S7:** combine two variable terms on a side to get a positive number (e.g., combine 3x+6x to make 9x)
- **S8:** combine two variable terms on a side to get a negative number (e.g., combine 3x-6x to make -9x)
- **S9:** combine two constant terms (e.g., combine 3-6 to make -3)

Provided Files
We are providing a file a2.py along with this assignment submission that includes stubs for all the methods you need to implement. We encourage you to download code from https://github.com/aimacode/aima-python and use it for this assignment. In particular, logic.py and planning.py will be useful. You should include anything you used in your assignment submission. If you want to use other online libraries this is fine with advance permission from the instructor.

Assignment Submission
Code should be submitted on CourseWeb as a single .zip file. Written portions of this assignment should be submitted on Gradescope.

Part A – Propositional Logic Feedback Engine (30 points)
Your first task is to build a system for giving feedback to students on their problem-solving. Implement a function called *giveFeedback.*

giveFeedback takes a String representing a propositional logic KB that specifies the student's state. The following are the possible propositional symbols in this knowledge base:

- CorrectAnswer. This specifies whether the student's most recent answer was correct.
- NewSkill. This specifies whether this skill related to the student's most recent answer is new to the student (the student has not yet seen a problem with that skill).
- MasteredSkill. This specifies whether the skill related to the student's most recent answer has been mastered by the student.
- CorrectStreak. This specifies whether the student's last three answers were correct.
- IncorrectStreak. This specifies whether the student's last three answers were incorrect.

Thus, possible inputs to *giveFeedback* might any combinations of conjunctions of these propositions. For example: "CorrectStreak & MasteredSkill", "IncorrectStreak", "NewSkill & CorrectAnswer & CorrectStreak."

Outputs of giveFeedback are one of eight feedback messages:

- Message1: “Correct. Keep up the good work!”
- Message2: “Correct. I think you’re getting it!”
- Message3: “Correct. After this problem we can switch to a new activity.”
- Message4: “Incorrect. Keep trying and I’m sure you’ll get it!”
- Message5: “Incorrect. After this problem, we can switch to a new activity.“
- Message6: “Incorrect. The following is the correct answer to the problem.”
- Message7: “Correct.”
- Message8: “Incorrect.”

The following are the rules for producing feedback messages, which should be encoded somewhere in your program:

- CorrectAnswer => (Message1 v Message2 v Message3 v Message7)
- ~CorrectAnswer => (Message4 v Message5 v Message6 v Message8)
- (MasteredSkill & IncorrectAnswer) v (MasteredSkill & CorrectStreak) => IsBored
- NewSkill v IncorrectStreak => Message6
- (IncorrectStreak & CorrectAnswer) v (NewSkill & CorrectStreak) => NeedsEncouragement
- NeedsEncouragement => Message2 v Message4
- IsBored => Message3 v Message5
- (NewSkill & CorrectAnswer) v CorrectStreak => Message1

giveFeedback should use resolution to decide which feedback message to give (which message the KB entails). The lower numbered messages have the highest priority (i.e., Message3 should be given before Message8, and thus if Message 3 is entailed, the algorithm doesn’t need to check to see if Message8 is also entailed).

Full marks will be given here if giveFeedback successfully uses a propositional logic representation and proof by resolution to arrive at the correct feedback message to give.

Part B – First-Order Logic & Planning (50 points)

Your next task is to build a planner to solve the problem like an expert student would. You need to represent each equation in First-Order Logic, represent the possible actions on the equation, and then run a planning algorithm that returns the sequence of steps representing the solution to the problem.

First, describe the design of your representation. 10 points will be given for a design that represents all possible equations in the domain and that allows you to use planning to solve the problem. In your representation, you only need to worry about representing states that are on the correct path to a solution.

1. What are your predicates?

- VarLeft(x)
  - Where x is the **coefficient** for the variable on the **left side** of the equal side
  - Ex: VarLeft(3) would look like: 3x = ?
- VarRight(x)
- ConstLeft(x)
  - Where x is the value of the constant
  - Ex: ConstLeft(7) would look like: 7 = ?
- ConstRight(x)
- I also negate these, like ~ConstLeft(7) and ~VarLeft(3)

2. How would you represent the example equations enumerated above in the task domain (this is your initial state)?

x=3
-3x=6
3x-2=-6
4+3x=6x-7

VarLeft(1) & ConstRight(3)

VarLeft(-3) & ConstRight(6)

VarLeft(3) & ConstLeft(-2) & ConstRight(-6)

ConstLeft(4) & VarLeft(3) & VarRight(6) & ConstRight(-7)

Next, describe the design of your planner. 10 points will be given to a representation that allows the most efficient solution to the problem to be found (10 points):

3. What is your goal state in first-order logic?

VarLeft(1) & ConstRight(x) OR ConstLeft(x) & VarRight(1) where x is a number -9 to +9

*The variable x having a value coefficient of 1 will ensure division successfully occurs

**The OR clause allows for you to skip some add/subtract steps, making it more efficient

4. What are your action definitions? You only need to consider actions that will lead you to a goal state.

- AddConstantsLeft(x)
  - Preconditions: ConstLeft(x)
  - Postconditions: ConstRight(x) & ~ConstLeft(x)
- AddConstantRight(x)
  - Preconditions: ConstRight(x)
  - Postconditions: ConstLeft(x) & ~ConstRight(x)
- AddVariableLeft(x)
  - Preconditions: VarLeft(x)
  - Postconditions: VarRight(x) & ~VarLeft(x)
- AddVariableRight(x)
  - Preconditions: VarRight(x)
  - Postconditions: VarLeft(x) & ~VarRight(x)
- DivideRightVariable(x)
  - Preconditions: VarRight(x) & NoLeftVariables()
  - Postconditions: VarRight(1) & ~VarRight(x)
- DivideLeftVariable(x)
  - Preconditions: VarLeft(x) & NoRightVariables()
  - Postconditions: VarLeft(1) & ~VarLeft(x)
- CombineLeftConstants(x, y)
  - Preconditions: ConstLeft(x) & ConstLeft(y)
  - Postconditions: ConstLeft(x + y) & ~ConstLeft(x) & ~ConstRight(x)
- CombineRightConstants(x, y)
  - Preconditions: ConstRight(x) & ConstRight(y)
  - Postconditions: ConstRight(x + y) & ~ConstRight(x) & ~ConstRight(y)
- CombineLeftVariables(x, y)
  - Preconditions: VarLeft(x) & VarLeft(y)
  - Postconditions: VarLeft(x + y) & ~VarLeft(x) & ~VarLeft(y)
- CombineRightVariables(x, y)
  - Preconditions: VarRight(x) & VarRight(y)
  - Postconditions: VarRight(x + y) & ~VarRight(x) & ~VarRight(y)

5. Are you going to need to use functions as part of your representation? Why or why not?

Yes, I used functions in order to generate my representation of an equation solver. I have a function that **parses the input string** to generate each of the initial statements. This would create all left/right constants & variables and made counts for them. Along with this, since you can't do math within the **action class** used from **logic.py**, I would precalculate the combine() functions for variables so the post conditions included correct computations. This way, I could print out the correct number if I was combining variables and needed to print out what number to divide by. Also this would be done for additions of variables.

I also used **astar_search()** and **ForwardPlan()** to find the set of steps I needed to take to reach the goal that was found in the search.

Then, implement a *solveEquation* function that executes the planning (30 points). *solveEquation* takes as a single parameter a string representing the equation to be solved. Your system should use your <u>first-order logic representation</u> and <u>forward planning</u>, and return a list of actions to execute based on the problem description. For example, the call solveEquation('3x-2=6') should return: ['add 2', 'combine RHS constant terms', 'divide 3']. You can assume that terms like +2 and -2 automatically cancel out, and don't require an action. Full marks will be given if the system successfully uses first-order logic and forward planning to return the most efficient solution to the problem.

Part C – Student Model (40 points)

Next, you're going to develop your student model for your system. A student model is a representation of what students know. Your goal is to use your student model to assess whether students can solve a new problem.

Each problem requires particular skills to solve efficiently (enumerated in the Task Domain section). For example, "-3x=6" only requires dividing both sides by a negative constant to solve efficiently (S6). For this part of the assignment, assume that students either know a particular skill or do not know a skill.

Write a function *predictSuccess* that takes two parameters. The first is a list of Strings representing skills that a student has. For example, if you called predict success with the list ['S1', 'S2'], this particular student has mastered S1 and S2. The second parameter is a string representing the equation the student is currently trying to solve (e.g., "-3x=6"). The function should return a list of the skills missing that students require to solve the problem – in the above example, it would return ['S6']. If the student can solve the problem, the function should return an empty list []. It can use any method you would like to do so, as long as the method relies on a first-order logic representation of student knowledge to make inferences.

Describe your implementation of predictSuccess using either words or pseudocode. You will earn 10 points for the design of a reasonable strategy that uses first-order logic.

There FOL knowledge base will use the same predicates as before (Ex: LeftConst(-3) & RightVar(2)). However, I need to construct rules to allow my system to resolve if a skill is needed. They are as follows for **constants**:

- LeftConst(a) & LeftConst(b) ➔ Skill9(a, b)
    - If there are more than 1 constants, use this skill
    - Skill 9 - combine two constant terms
- LeftVar(a) & LeftVar(b) ➔ Skill8(a, b) | RightVar(a) & RightVar(b) ➔ Skill8(a, b)
    - If there is more than 1 Variable, we will be using this skill
    - Skill 8 - combine two variable terms on a side to get a negative number
- LeftVar(a) & LeftVar(b) ➔ Skill7(a, b) | RightVar(a) & RightVar(b) ➔ Skill7(a, b)
    - If there is more than 1 Variable, we will be using this skill
    - Skill 7 - combine two variable terms on a side to get a positive number
- LeftVar(a) ➔ Skill6(a) | RightVar(a) ➔ Skill6(a)
    - If there exists a variable and it is negative
    - Skill 6 - divide both sides by a negative constant
- LeftVar(a) ➔ Skill6(a) | RightVar(a) ➔ Skill6(a)
    - If there exists a variable and it is positive
    - Skill 5 - divide both sides by a positive constant
- LeftConst(a) ➔ Skill4(a)
    - If there is a constant on the left hand side and it is positive
    - Skill 4 - add a negative constant term from both sides
- LeftConst(a) ➔ Skill3(a)
    - If there is a constant on the left hand side and it is negative
    - Skill 3 - add a positive constant term to both sides
- RightVar(a) ➔ Skill2(a)
    - If there is a variable on the right hand side and it is positive
    - Skill 2 - add a negative variable term to both sides
- RightVar(a) ➔ Skill1(a)
    - If there is a variable on the right hand side and it is negative
    - Skill 1 - add a positive variable term to both sides

30 points will be given for the implementation of a function that returns a correct answer using reasoning based on first-order logic.

Part D – Putting Your System Together (30 points)

Finally, you're going to create a *stepThroughProblem* function. The function should take three parameters: an equation, a student action (represented as described in Part B), and a list of current student skills (represented as described in Part C). The function should determine if the student action was correct (based on the results of the planner developed in Part B). It should then return a list containing two items: a feedback message to the student (determined through the rules in Part A) and a list of the skills the student now knows that is updated based on the success or failure of the action. You can assume that if the action is the correct action for the problem, the student now knows the skill related to the action. If the action is unsuccessful, the skill list should remain unchanged. In addition, you should create and update a model of the student based on the student actions anad the requirements of Part A (i.e., log correct streaks, incorrect streaks, whether a skill is new to the student).

We will use this function to simulate running the student through several problems (worth 20 points). Full marks will be given if the function returns the correct feedback and correct skill model for each student action.

Now, reflect on your implementation (10 points). Which parts of your implementation are specific to the domain of equation solving. Which parts of your implementation could be used with any domain? What were the advantages and disadvantages of first-order logic for building this type of tutoring system? Full marks will be given if you reference specific functions or aspects of your implementation and critically reflect on these issues.

- The largest portion of the domain-specific implementation was the **parse_equation()** function I created.
    - This was totally dependent on the rules defined in the premise for part A, which included having only two terms at most on either side of the equation, guaranteed constants between 1-9, and variable coefficients also 1-9.
    - This could not be applied to, for example, a different domain of the Wumpus World which had a domain of coordinates in the grid.
- The resolution portions of **pl_resolution**() and the graph searches with **astar_search()** from the **forward_plan()** which found the goal based on our actions was domain independent.
    - These parts of the implementation required shared actions and goals which are made into generic expressions, that can be solved with any domain
- Advatages of FOL
    - Had the simplest implementation as opposed to forward planning with propositional logic
- Disadvantages of FOL
    - Required that you only use definite clauses, which restricted more advanced logic
    - If you wanted to use advanced logic it required tedious logical operations to create the needed definite clauses

Bonus Points (up to 15 points)
Bonus points will be given if, along with the above assignment requirements, you make a more sophisticated knowledge model (i.e., one that takes into account forgetting, or increases a confidence level in what students know based on repeated correct performance). If you decide to extend the assignment in this way, name your functions predictSuccessBonus and stepThroughProblemBonus.

Are you submitting bonus functions: **No**