HW 0\1 (AIMA Chapter 2)
Artificial Intelligence
Spring 2012

Instructions:

Due at **10:30 a.m. on Thursday, Jan 26th**. All submissions will be via Blackboard. There will be two parts to this homework. Part 1 will be short answer questions that are to be written up **individually (pdf/txt/doc are all acceptable formats)**. Part 2 will be python programming that may be completed in **pairs** (please no more than 2 per group) as we are trying to support a pair-programming model for this course. Each person must submit their own Part 1's, but only one person per group needs to submit Part 2. If you have any questions, please post to Piazza.

Part 1 – Individual Short-Answer Questions

Agents:

1. For each of the following assertions about rational agents, say whether it is true or false. Briefly justify your answer, giving a short examples or counterexample where appropriate. If you do not know what something is, please ask a TA or Google.

    a. It is possible that a rational agent has no measure for performance.

    Answer: False, the definition of rational requires there be some metric of performance

    b. A perfectly rational agent knows everything about its environment.

    Answer: False, it just responds in the optimal way to what limited information it does know about its environment, it is NOT omniscient

    c. It is not possible for a perfectly rational agent to lose a simple board game such as checkers or tic-tac-toe.

    Answer: False, if both agents are perfectly rational, one must lose eventually (though perhaps a tie is possible…)

    d. The iPhone/Android keyboards are perfectly rational agents.

    Answer: False, even though they predict what you type, they may not make the optimal suggestion given your previous history

2. For each of the following activities, give a PEAS description on the task environment and characterize it in terms of the properties listed in Section 2.3.2

    a. An autonomous thermostat (www.nest.com) [Temp, Room, Heater, Thermometer]
    b. A search engine [Speed, Websites, HTTP, Analytics]

3. Classify the following examples of agents as either pure reflex, model-based (has state), goal-based, utility-based, or learning and explain why (use the most sophisticated that is applicable):
    a. An ant that walks in a straight line, it stops if there is water in front of it [reflex]
    b. An ant that encounters new creatures frequently and must adapt based on previous encounters [learning]
    c. An ant that walks around randomly, but avoids water and revisiting locations [model-based]
    d. An ant that can choose what food to eat that is varying distances away from it [utility]

Python:

1. Does the following line of code return an error? If so, what error and what does it mean? How would you correct it?
    ```
    { [1,2]: "Bob" }
    ```

    Answer: unhashable type: “list”, type must be immutable, {(1,2):”Bob”}

2. What is the resulting L2 for the following two programs? Why are they different?
    ```
    L1 = [1,2,3]          L1 = [1,2,3]
    L2 = L1[:]            L2 = L1
    L1.reverse()          L1.reverse()
    ```

    Answer: [1,2,3] and [3,2,1], the first is making a copy, the second sets L2 to reference L1

3. What does the second line from the following code do? What is the result of y? Why?
    ```
    x= [1,2,3]
    y = x.reverse()
    ```

    Answer: x = [3,2,1] but y is nothing since x.reverse() does not return anything

4. What is a list comprehension in python? What single line of code would you use to create a list of 2-tuples where each tuple is a pair (x,x+1) where x is from 0 to 9 using a list comprehension.

   Answer: [(x,x+1) for x in range(10)]

Part 2 – Pair Programming Assignment (No more than 2 people)

The code should be in a single file called sudoku.py for this assignment.

Later in the course we will be testing several different algorithms for solving Sudoku puzzles. Sudoku is a simple puzzle game: given a partially filled in 9x9 grid, grouped into a 3x3 grid of 3x3 blocks, the goal is to fill each square with a digit in the range 1 to 9, subject to the requirement that each row, column, and block must contain each digit exactly once. (For more details, see http://sudoku.org.uk.) In this homework, we'll set up some of the basic data structures and methods that will be useful for dealing with sudoku boards in later homeworks.

1. Make a file called sudoku.py and in it create a class called *SudokuBoard*. Give the class an initialization function that takes as input the name of a file that contains a sudoku board. (You don't have to do anything inside the initialization function yet.)

2. Add the function *parseBoard* to the *SudokuBoard* class. This function should take as input the name of a file containing a *sudoku* board and return a list of lists of integers, where each internal list corresponds to one row of the board. See the sample file sudoku-board.txt for an example input board. Your function should transform all "*" characters to the integer 0 for ease of later processing. *Hint: You should be able to write this function in a single line using a nested list comprehension.*

3. Add the public variable *self.board* to the initialization function of the *SudokuBoard* class, initializing it by calling *parseBoard*.

4. Add the public function *printBoard* to the *SudokuBoard* class. This function should print a representation of *self.board* to *stdout*. For example, given the board in sudoku-board.txt, this function should produce exactly the following output:

```
* 1 * | 4 2 * | * * 5
* * 2 | * 7 1 | * 3 9
* * * | * * * | * 4 *
-------+---------+-------
2 * 7 | 1 * * | * * 6
* * * | * 4 * | * * *
6 * * | * * 7 | 4 * 3
```

```
-------+---------+-------
* 7 * | * * * | * * *
1 2 * | 7 3 * | 5 * *
3 * * | * 8 2 | * 7 *
```

Hint: You should be able to write this function using two for loops and some if statements.

5. The rules of Sudoku mandate that the 9 board locations comprising any row, column, or 3x3 block must each have a different number from 1-9 in them. So, we can represent a game constraint as the set of board locations that comprise a row, column, or block (with the semantics that those 9 locations must each have a different number from 1-9 in them). For example, the constraint for the upper left block would be set([(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)]).

   Add the private function *computeConstraintSets* to the *SudokuBoard* class. This function should return a list of sets of 2-tuples. Each tuple should indicate a board location, for example (0, 1) for the first row, second column location. Each set should contain nine such tuples, comprising a row, column, or 3x3 block. The overall constraints list should then contain 27 sets. *Hint: You should be able to write this function using a nested list comprehension for the row constraints, another for the column constraints, and a third (more complicated) comprehension for the block constraints. Note that you can create a set from a list by wrapping the list with set().*

6. Add the private variable *self. constraints* to the initialization function of the *SudokuBoard* class, initializing it by calling *computeConstraintSets*.
7. Add the private function *computePointDict* to the *SudokuBoard* class. This function should return a dictionary mapping each board location to a list of the constraints it's involved in. For example, the location (0, 1) should be mapped to a list containing references to the three sets in the *self.__constraints* variable that describe the first row, the second column, and the upper left block. *Hint: You should be able to write this function using two for loops and a list comprehension.*

8. Add the private variable *self. pointDict* to the initialization function of the *SudokuBoard* class, initializing it by calling *computePointDict*.

9. Add the public function *getConstraintSets* to the *SudokuBoard* class. This function should take as input a board location such as (0, 1) and return the *self. pointDict* entry associated with that location.

10. Add the public function *computeUnusedNums* to the *SudokuBoard* class. This function should take as input a constraint (a set of board locations) and return a set of integers. The integers should be all those numbers 1-9 that don't occur in any of the constraint's board locations. For example, in the case of the upper left block from sudoku-board.txt, this would be set([3, 4, 5, 6, 7, 8, 9]). *Hint: You should be able to write this function using a for loop and some set operations.*

11. Add the public function *isSolved* to the *SudokuBoard* class. This function should return true if the board is valid and complete, and false otherwise. That is, if each constraint region contains all the numbers 1-9, it should return true. Else, it should return false. Hint: You should be able to write this function using two for loops, some if statements, and some set operations.

12. We will run a set of automated tests on your code to grade it. To ensure that your work will be compatible with the testing format, place the test-hw1.py script in the folder where your sudoku.py file is and try running it: python test-hw1.py. Note that this script requires that sudoku-board.txt is in the same folder as your code. You can assume we will provide this board file when running the tests for grading, so you don't need to turn it in with your code. (Just submit the sudoku.py file.) Also note that the tests in test-hw1.py are not comprehensive; we will run more tests for grading, so you should perform additional testing of your code prior to submitting it.