CMPSC/DS 442: Homework 3

TO PREPARE AND SUBMIT HOMEWORK

Follow these steps exactly, so the Gradescope autograder can grade your homework. Failure to do so will result in a zero grade:

1. Download the homework template file `homework3_starter.py` from Canvas. This is a python file with the correct function names for autograding, where you fill in the definitions.

2. RENAME the template file by replacing the string ' `starter` ' with your PSU web access id. For example, if your PSU email id were abc123, you would rename your file as `homework3_abc123.py`. The autograder will not find your file if you don't use the correct naming convention, and you will get a zero grade.

3. You *must* download the zip file with the puzzles to practice on, sudoku.zip.

4. Enter your solutions in your renamed template file, using any python version from 3.7 to 3.11. Make sure your file can import into a python console before you submit. If the autograder cannot import your file, you will get a zero.

5. Submit your solution file to Gradescope by its due date. If you fail to make the submission deadline, your homework will be counted late, and you might get a zero grade.

6. Please ask questions if you have any via piazza or during office hours.

Instructions

In this assignment, you will work on two kinds of algorithms: PL and CSPs. Question 1-2 is PL, and Question 3 is CSPs. For PL, you will implement a collection of basic algorithms for working with logical expressions, then will apply them to solve textual puzzles expressible using propositional logic. For CSPs, you will implement three inference algorithms for the popular puzzle game Sudoku.

A skeleton *.py file `homework3_starter.py` containing empty definitions for each question has been provided (see above). Do not modify any names or function signatures in the provided template file, or the autograder cannot test your functions. However, you are free to introduce additional variables or functions if needed.

You may import definitions from any standard Python library, and are encouraged to do so in case you find yourself reinventing the wheel. If you are unsure where to start, consider taking a look at the data structures and functions defined in the `collections`, `itertools`, `copy`, `Queue`, and `random` modules. You cannot use any libraries (like numpy) that are not standard; see the homework FAQ on Canvas

You will find that in addition to a problem specification, most programming questions also include a pair of examples from the Python interpreter. These are meant to illustrate typical use cases, and should not be taken as comprehensive test suites.

1. Propositional Logic [50 points]

In this section, you will implement a collection of classes corresponding to logical expressions, a simple knowledge base, and algorithms for propositional logic inference via truth table enumeration and resolution. All expressions will be subclasses of the provided base Expr class. The fragment of logic we will be working with can be defined recursively as follows:

- Atom: Atom(name) is an expression.
- Negation: If e is an expression, then Not(e) is an expression.
- Conjunction: If $e_1,e_2,\ldots,e_n$ are expressions, then And($e_1,e_2,\ldots,e_n$) is an expression.
- Disjunction: If $e_1,e_2,\ldots,e_n$ are expressions, then Or($e_1,e_2,\ldots,e_n$) is an expression.
- Implication: If e_1 and e_2 are expressions, then Implies(e_1, e_2) is an expression.
- Biconditional: If e_1 and e_2 are expressions, then Iff(e_1, e_2) is an expression.

You will find that a generic hash function has been defined for the Expr class, and that the initialization methods for each of its subclasses have been provided. These should not be altered. Combined with the methods for equality checking to be implemented below, this will ensure that expressions behave as expected when used as elements of sets.

1. **[3 points]** First read through and understand the provided `__init__` and `__hash__` methods for the `Atom`, `Not`, `And`, `Or`, `Implies`, and `Iff` expression subclasses, keeping in mind that expressions will be considered immutable for our purposes. In particular, review the `*args` syntax (link) and `frozenset` class (link) if needed. The former allows for convenient n-ary conjunctions and disjunctions, and the latter is used to ensure immutability.

   We use sets rather than lists for conjunctions and disjunctions to guarantee that, e.g., all 24 permutations of the conjuncts in the expression $A \land B \land C \land D$ will be equivalent. Moreover, you may find that certain set functions such as union and set difference will prove useful later on.

   As a first exercise, implement the `__eq__(self, other)` methods in each subclass, which will be called when expressions are compared using the `==` operator. You should check for syntactic equality only, in the sense that two expressions should be considered equal only if they are of the same class and have the same internal structure. No simplification should be performed. As a special case, `Iff(a, b)` should be equal to `Iff(b, a)`. *Hint: Each of these can be implemented in a single line.*

```
>>> Atom("a") == Atom("a")
True
>>> Atom("a") == Atom("b")
False
```

```
>>> And(Atom("a"), Not(Atom("b"))) == \
... And(Not(Atom("b")), Atom("a"))
True
```

2. **[2 points]** Implement the `_repr_(self)` method in each expression subclass, which should return a string representation of the given expression. Any reasonable choice will suffice for this exercise, as this is primarily intended for debugging purposes.

```
>>> a, b, c = map(Atom, "abc")
>>> Implies(a, Iff(b, c))
Implies(Atom(a), Iff(Atom(b), Atom(c)))
```

```
>>> a, b, c = map(Atom, "abc")
>>> And(a, Or(Not(b), c))
And(Atom(a), Or(Not(Atom(b)), Atom(c)))
```

3. **[5 points]** Implement the `atom_names(self)` method in each expression subclass, which should return the set of names that occur in atoms contained within the given expression.

```
>>> Atom("a").atom_names()
set(['a'])
>>> Not(Atom("a")).atom_names()
set(['a'])
```

```
>>> a, b, c = map(Atom, "abc")
>>> expr = And(a, Implies(b, Iff(a, c)))
>>> expr.atom_names()
set(['a', 'c', 'b'])
```

4. **[5 points]** Implement the `evaluate(self, assignment)` method in each expression subclass, which should return the truth value of the given formula under the provided assignment from atom names to Boolean values. You may assume that the assignment dictionary contains the necessary entries to fully evaluate the expression.

```
>>> e = Implies(Atom("a"), Atom("b"))
>>> e.evaluate({"a": False, "b": True})
True
>>> e.evaluate({"a": True, "b": False})
False
```

```
>>> a, b, c = map(Atom, "abc")
>>> e = And(Not(a), Or(b, c))
>>> e.evaluate({"a": False, "b": False,
...   "c": True}
True
```

5. **[10 points]** Write a `satisfying_assignments(expr)` function that generates all assignments from atom names to truth values under which the input expression is true. The assignments may be generated in any order, as long as all satisfying assignments are produced.

```
>>> e = Implies(Atom("a"), Atom("b"))
>>> a = satisfying_assignments(e)
>>> next(a)
{'a': False, 'b': False}
>>> next(a)
{'a': False, 'b': True}
>>> next(a)
{'a': True, 'b': True}
```

```
>>> e = Iff(Iff(Atom("a"), Atom("b")),
...   Atom("c"))
>>> list(satisfying_assignments(e))
[{'a': False, 'c': False, 'b': True},
 {'a': False, 'c': True, 'b': False},
 {'a': True, 'c': False, 'b': False},
 {'a': True, 'c': True, 'b': True}]
```

6. **[10 points]** Implement the `to_cnf(self)` method in each expression subclass, which should return an expression in conjunctive normal form that is logically equivalent to the input. Specifically, the output of this method should be a literal (i.e. an atom or a negated atom), a disjunction of literals, or a conjunction consisting of literals and/or disjunctions of literals.

```
>>> Atom("a").to_cnf()
Atom(a)
>>> a, b, c = map(Atom, "abc")
>>> Iff(a, Or(b, c)).to_cnf()
And(Or(Atom(b), Atom(c), Not(Atom(a))),
    Or(Not(Atom(b)), Atom(a)),
    Or(Not(Atom(c)), Atom(a)))
```

```
>>> Or(Atom("a"), Atom("b")).to_cnf()
Or(Atom(b), Atom(a))
>>> a, b, c, d = map(Atom, "abcd")
>>> Or(And(a, b), And(c, d)).to_cnf()
And(Or(Atom(d), Atom(a)),
    Or(Atom(a), Atom(c)),
    Or(Atom(b), Atom(c)),
    Or(Atom(b), Atom(d)))
```

7. **[15 points]** In this question, we will consider a knowledge base to be an object which stores a collection of facts and supports entailment queries based on those facts.

   First write the `__init__(self)` and `get_facts(self)` methods in the `KnowledgeBase` class, which initialize and return an internal fact set, respectively. Next write the `tell(self, expr)` method, which converts the input expression to conjunctive normal form and adds the resulting conjuncts to the internal fact set. Lastly write the `ask(self, expr)` method, which returns a Boolean value indicating whether the facts in the knowledge base entail the input expression. You should determine entailment using the resolution algorithm discussed in class.

   Hint: Making helper functions for the `ask` method is recommended.

```
>>> a, b, c = map(Atom, "abc")
>>> kb = KnowledgeBase()
>>> kb.tell(a)
>>> kb.tell(Implies(a, b))
>>> kb.get_facts()
set([Or(Atom(b), Not(Atom(a))),
    Atom(a)])
>>> [kb.ask(x) for x in (a, b, c)]
[True, True, False]
```

```
>>> a, b, c = map(Atom, "abc")
>>> kb = KnowledgeBase()
>>> kb.tell(Iff(a, Or(b, c)))
>>> kb.tell(Not(a))
>>> [kb.ask(x) for x in (a, Not(a))]
[False, True]
>>> [kb.ask(x) for x in (b, Not(b))]
[False, True]
>>> [kb.ask(x) for x in (c, Not(c))]
[False, True]
```

2. Logic Puzzles [0 points]

This is an optional question. In this section, you will encode some simple logic puzzles in propositional logic and use the functions written in the previous section to solve them. You will be graded both on the correctness of your results and on the correctness of your formulations.

1. Consider the following set of facts. If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If the unicorn is either immortal or a mammal, then it is horned. The unicorn is magical if it is horned.
   From these, can it be inferred that the unicorn is mythical? That it is magical? That it is horned? Using the atomic expressions `Atom("mythical")`, `Atom("mortal")`, `Atom("mammal")`, `Atom("horned")`, and `Atom("magical")`, populate the indicated knowledge base with the appropriate facts, and assign the appropriate queries to the indicated variables. Then, use these to answer the above questions, and record your answers in the corresponding variables.

2. You would like to throw a party subject to the following conditions: John will come if Mary or Ann come, Ann will come if Mary does not come, and if Ann comes, then John will not come.

   Letting the atomic expressions `Atom("a")`, `Atom("j")`, and `Atom("m")` denote that Ann, John, and Mary come, respectively, encode the given conditions as a single conjunction, and use the previously-defined `satisfying_assignments(expr)` function to compute all valid scenarios. In what way(s) can the guests attend without violating the constraints?

3. A game show contestant is to choose between two rooms, each of which may either contain a prize or be empty. The sign on the first door states: "This room contains a prize, and the other room is empty." The sign on the second door states: "At least one room contains a prize, and at least one room is empty." The contestant is told that exactly one sign is true.

   Let the atomic expression `Atom("p1")` denote that the first room contains a prize, and let the atomic expression `Atom("e1")` denote that the first room is empty. Similarly define `Atom("p2")` and `Atom("e2")` for the second room. Also let the atomic expressions `Atom("s1")` and `Atom("s2")` denote that the first sign and second sign are true, respectively.

   Using these, populate the indicated knowledge base with the appropriate facts. Then make the necessary queries to determine what the contestant can deduce about the contents of each room and the truth of the signs. What is the correct state of affairs?

4. There are three suspects for a murder: Adams, Brown, and Clark. Adams says “I didn’t do it. The victim was old acquaintance of Brown’s. But Clark never knew him.” Brown states “I didn’t do it. I didn’t know the guy.” Clark says “I didn’t do it. I saw both Adams and Brown downtown with the victim that day; one of them must have done it.” Assume that the two innocent men are telling the truth, but that the guilty man is not.

   Let the atomic expressions `Atom("ia")`, `Atom("ib")`, and `Atom("ic")` denote that Adams, Brown, and Clark are innocent, respectively, and let `Atom("ka")`, `Atom("kb")`, and `Atom("kc")` denote that Adams, Brown, and Clark knew with the victim. Populate the indicated knowledge base with the given information, and query it as necessary to determine which suspect is guilty. Record your answer and the corresponding query in the provided variables.

3. Sudoku [50 points]

In the game of Sudoku, you are given a partially-filled 9 x 9 grid, grouped into a 3 x 3 grid of 3 x 3 blocks. The objective is to fill each square with a digit from 1 to 9, subject to the requirement that each row, column, and block must contain each digit exactly once.

In this section, you will implement the AC-3 constraint satisfaction algorithm for Sudoku, along with two extensions that will combine to form a complete and efficient solver.

A number of puzzles have been made available in the sudoku.zip file,

including: • An easy-difficulty puzzle: `hw3-easy.txt` .

• Four medium-difficulty puzzles: `hw3-medium1.txt` , `hw3-medium2.txt` , `hw3-medium3.txt` , and `hw3-medium4.txt` .

• Two hard-difficulty puzzles: `hw3-hard1.txt` and `hw3-hard2.txt` .

The examples in this section assume that these puzzle files have been placed in a folder named `sudoku` located in the same directory as the homework file.

An example puzzle from the Daily Pennsylvanian, available as `hw3-medium1.txt` , is depicted below.

```
* 1 5 * 2 * * * 9
* 4 * * * * 7 * *
* 2 7 * * 8 * * *
9 5 * * * 3 2 * *
7 * * * * * * * 6
* * 6 2 * * * 1 5
* * * 6 * * 9 2 *
* * 4 * * * * 8 *
  * * * 3 * 6 5 *
```

Textual Representation

	1	5		2				9
	4					7		
	2	7			8			
9	5				3	2		
7								6
		6	2				1	5
			6			9	2	
		4					8	
2				3		6	5	

Initial Configuration

6	1	5	3	2	7	8	4	9
8	4	9	5	1	6	7	3	2
3	2	7	9	4	8	5	6	1
9	5	1	4	6	3	2	7	8
7	3	2	8	5	1	4	9	6
4	8	6	2	7	9	3	1	5
1	7	3	6	8	5	9	2	4
5	6	4	7	9	2	1	8	3
2	9	8	1	3	4	6	5	7

Solved Configuration

1. **[3 points]** In this section, we will view a Sudoku puzzle not from the perspective of its grid layout, but more abstractly as a collection of cells. Accordingly, we will represent it internally as a dictionary mapping from cells, i.e. (row, column) pairs, to sets of possible values.

   In the `Sudoku` class, write an initialization method `__init__(self, board)` that stores such a mapping for future use. Also write a method `get_values(self, cell)` that returns the set of values currently available at a particular cell.

In addition, write a function `read_board(path)` that reads the board specified by the file at the given path and returns it as a dictionary. Sudoku puzzles will be represented textually as 9 lines of 9 characters each, corresponding to the rows of the board, where a digit between `"1"` and `"9"` denotes a cell containing a fixed value, and an asterisk `"*"` denotes a blank cell that could contain any digit.

```
>>> b = read_board("sudoku/hw3- medium1.txt")
>>> Sudoku(b).get_values((0, 0))
set([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> b = read_board("sudoku/hw3- medium1.txt")
>>> Sudoku(b).get_values((0, 1))
set([1])
```

2. **[2 points]** Write a function `sudoku_cells()` that returns the list of all cells in a Sudoku puzzle as (row, column) pairs. The line `CELLS = sudoku_cells()` in the `Sudoku` class then creates a class-level constant `Sudoku.CELLS` that can be used wherever the full list of cells is needed. Although the function `sudoku_cells()` could still be called each time in its place, that approach results in a large amount of repeated computation and is therefore highly inefficient. The ordering of the cells within the list is not important, as long as they are all present.

```
>>> sudoku_cells()
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), ..., (8, 5), (8, 6), (8, 7), (8, 8)]
```

3. **[5 points]** Write a function `sudoku_arcs()` that returns the list of all arcs between cells in a Sudoku puzzle corresponding to inequality constraints. In other words, each arc should be a pair of cells whose values cannot be equal in a solved puzzle. The arcs should be represented a two-tuples of cells, where cells themselves are (row, column) pairs. The line `ARCS = sudoku_arcs()` in the `Sudoku` class then creates a class-level constant `Sudoku.ARCS` that can be used wherever the full list of arcs is needed. The ordering of the arcs within the list is not important, as long as they are all present.

```
>>> ((0, 0), (0, 8)) in sudoku_arcs()
True
>>> ((0, 0), (8, 0)) in sudoku_arcs()
True
>>> ((0, 8), (0, 0)) in sudoku_arcs()
True
```

```
>>> ((0, 0), (2, 1)) in sudoku_arcs()
True
>>> ((2, 2), (0, 0)) in sudoku_arcs()
True
>>> ((2, 3), (0, 0)) in sudoku_arcs()
False
```

4. **[10 points]** In the `Sudoku` class, write a method `remove_inconsistent_values(self, cell1, cell2)` that removes any value in the set of possibilities for `cell1` for which there are no values in the set of possibilities for `cell2` satisfying the corresponding inequality constraint. Each cell argument will be a (row, column)

pair. If any values were removed, return `True`; otherwise, return `False`.

Hint: Think carefully about what this exercise is asking you to implement. How many values can be removed during a single invocation of the function?

```
>>> sudoku = Sudoku(read_board("sudoku/hw3-easy.txt")) # See below for a picture.
>>> sudoku.get_values((0, 3))
set([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> for col in [0, 1, 4]:
...     removed = sudoku.remove_inconsistent_values((0, 3), (0, col))
...     print removed, sudoku.get_values((0, 3))
...
True set([1, 2, 3, 4, 5, 6, 7, 9])
True set([1, 3, 4, 5, 6, 7, 9])
False set([1, 3, 4, 5, 6, 7, 9])
```

5. **[10 points]** In the `Sudoku` class, write a method `infer_ac3(self)` that runs the AC-3 algorithm on the current board to narrow down each cell's set of values as much as possible. Although this will not be powerful enough to solve all Sudoku problems, it will produce a solution for easy-difficulty puzzles such as the one shown below. By "solution", we mean that there will be exactly one element in each cell's set of possible values, and that no inequality constraints will be violated.

8	2	1						7
			8				6	
	6		9	3				5
		8	2		1	6		
			7			2	8	4
2	4		6		3	7		
6		5				1		3
	7			5				
9	1	2						6

Initial Configuration

hw3-easy.txt

8	2	1	5	6	4	3	9	7
5	9	3	8	1	7	4	6	2
4	6	7	9	3	2	8	1	5
7	5	8	2	4	1	6	3	9
1	3	6	7	9	5	2	8	4
2	4	9	6	8	3	7	5	1
6	8	5	4	2	9	1	7	3
3	7	4	1	5	6	9	2	8
9	1	2	3	7	8	5	4	6

Result of Running AC-3

6. **[20 points]** Consider the outcome of running AC-3 on the medium-difficulty puzzle shown below. Although it is able to determine the values of some cells, it is unable to make significant headway on the rest.

				8	3	4		

				8	3	4	7	

3					4	8	2	1
7								
		9	4		1		8	3
4	6		5		7	1		
								7
1	2	5	3					9
		7	2	4				

Initial Configuration

hw3-medium2.txt

3					4	8	2	1
7								
		9	4		1		8	3
4	6		5		7	1		
								7
1	2	5	3					9
		7	2	4				

Inference Beyond AC-3

However, if we consider the possible placements of the digit 7 in the upper-right block, we observe that the 7 in the third row and the 7 in the final column rule out all but one square, meaning we can safely place a 7 in the indicated cell despite AC-3 being unable to make such an inference.

In the `Sudoku` class, write a method `infer_improved(self)` that runs this improved version of AC-3, using `infer_ac3(self)` as a subroutine (perhaps multiple times). You should consider what deductions can be made about a specific cell by examining the possible values for other cells in the same row, column, or block. Using this technique, you should be able to solve all of the medium-difficulty puzzles.

7. **[0 points]** Although the previous inference algorithm is an improvement over the ordinary AC-3 algorithm, it is still not powerful enough to solve all Sudoku puzzles. In the `Sudoku` class, write a method `infer_with_guessing(self)` that calls `infer_improved(self)` as a subroutine, picks an arbitrary value for a cell with multiple possibilities if one remains, and repeats. You should implement a backtracking search which reverts erroneous decisions if they result in unsolvable puzzles. For efficiency, the improved inference algorithm should be called once after each guess is made. This method should be able to solve all of the hard- difficulty puzzles, such as the one shown below.

	9		7			8	6	
	3	1			5		2	
8		6						
		7		5				6
			3		7			
5				1		7		
						1		9
	2		6			3	5	
	5	4			8		7	

Initial Configuration

hw3-hard1.txt

2	9	5	7	4	3	8	6	1
4	3	1	8	6	5	9	2	7
8	7	6	1	9	2	5	4	3
3	8	7	4	5	9	2	1	6
6	1	2	3	8	7	4	9	5
5	4	9	2	1	6	7	3	8
7	6	3	5	2	4	1	8	9
9	2	8	6	7	1	3	5	4
1	5	4	9	3	8	6	7	2

Result of Inference with Guessing