Homework 2: Murdlebot

Due September 14th at 10pm

Submit the **solver.py** file along with your question responses on Gradescope. **Please do not rename your Python file.**

Part 1: Murdlebot (80 points)

Murdle is a murder mystery-themed style of logic puzzle created by G. T. Karber.

Each Murdle puzzle provides a list of suspects, a list of weapons, and a list of locations. Each suspect has exactly one weapon in exactly one location. One suspect is the culprit— the murderer!

If you look in the provided **examples** folder, you will see that a sudden crime spree has been spreading throughout the Wellesley faculty. Can anyone be trusted?

Your goal is to create Murdlebot, an automated detective who will restore order on campus using the power of logic. Murdlebot will solve these crimes using *constraint satisfaction* and the set of clues provided in each puzzle.

Each puzzle clue rules out some possible assignments between suspects, locations, and weapons. By constraining the space of possible assignments using the puzzle's clues, your program can eliminate impossible combinations and hopefully land on a unique assignment (the identity of the guilty party!).

You should write your code in the file **solver.py**. You should add function calls to **main** to test your code as you go. Submit this file on Gradescope. **Keep the original name when you submit.**

Reading Puzzle Data (5 points)

We will represent each Murdle puzzle as a JSON file. Look at some example puzzles in the **examples** folder so that you understand how the data is structured.

Write a function called **read_puzzle**. It should take a filename string (for instance, "examples/mini_3x3.json") and return a dictionary constructed by calling **json.loads** on the text of the file.

Setting Up the Search Problem (20 points)

We will represent the search space as a list of possible states. A Murdle state is an assignment between suspects and weapons and locations: each suspect is assigned to exactly one weapon and one location.

We will represent each state as a list of tuples. The inner tuples are the assignments by suspect; there will be one tuple per suspect in the list.

For example, one possible state for a three-suspect puzzle might be:

```
[
  ("sohie", "deadly_cipher", "L180"),
  ("lyn", "poisoned_cookie", "hallway"),
  ("catherine", "syntax_error", "Leaky_Beaker")
]
```

Initially, our search space will include all possible assignments. We will gradually narrow this down by applying the clues given in the puzzle as constraints on the possible states.

Write a function called **make_state_space** that takes a puzzle dictionary as an argument. It should return a list containing every possible complete state (before applying any clues).

When constructing these states, remember the one-to-one structure of the puzzle:

- Every suspect must receive exactly one weapon and exactly one location
- Every weapon must be assigned to exactly one suspect
- Every location must be assigned to exactly one suspect

If a puzzle has three suspects, three weapons, and three locations, there are 36 possible states.

Constructing these states in an orderly manner is tricky. I would encourage you to work through an algorithm on paper for a small example before trying to implement it.

Checking Ownership (5 points)

Next, we will need a way to apply constraints to states by checking their compatibility with a given clue. It's useful to start with a helper function to check which suspect is in a particular place or has a particular weapon.

Write a function called **get_owner** that takes an weapon or location and an assignment dictionary, and returns the suspect who has that attribute.

Checking Predicates (20 points)

Now we can tackle applying clues. Each clue consists of one or more predicates. We want to eliminate states that do not match all of the predicates. Let's first handle the one-predicate and one-state case: given a predicate and a state, check if the predicate is true of that state.

To do this, write a function called **check_predicate**. It will take a predicate dictionary and a state, and return True if the state satisfies the predicate, and False otherwise.

Your solver should handle the following predicate types:

- **has**: a suspect has a particular weapon or location
- **not_has**: a suspect does not have a particular weapon or location
- **same_suspect**: two attributes belong to the same suspect
- **different_suspect**: two attributes belong to different suspects
- **either_or**: exactly one of two predicate lists is true

- **if_then**: if the first predicate list is true, the second predicate list must also be true

Hint: two of these predicate types will require *recursive calls* to the **check_predicate** function.

The **get_owner** helper function you defined above should be helpful in implementing **check_predicate**.

Applying Clues (10 points)

Now we will generalize to more complex clues and to lists of states.

First, we will write a function that tests a clue, which may contain a list of predicates. A state satisfies a clue only if it satisfies every predicate in that list.

Write a function called **check_clue**. It should take a clue dictionary and one possible state. It should return True if the state satisfies all of the predicates in the clue, and False otherwise.

Next, we will write a function to operate over lists of states, rather than a single state. Call this function **apply_clue**. It should take a list of possible states and a clue dictionary and return a new list containing only the states that satisfy that clue.

Solve (10 points)

Now you can write your main problem-solving function, **solve**. **solve** takes a puzzle dictionary and searches for a solution.

The search algorithm should work as follows:

- Construct the list of all possible complete states
- Apply each clue, one at a time, to filter the state list
- Stop if the state list becomes empty
- Stop if all clues have been applied.

If exactly one state remains after all ordinary clues have been applied, the assignment part of the puzzle has been solved. Return the status *solved*.

If no states remain, the clues contradict each other. Return a result with status *contradiction*.

If more than one state remains after all ordinary clues have been applied, the puzzle is underconstrained. Return a result with status *ambiguous*.

Naming the Culprit (10 points)

Our solver now identifies a unique state (if one exists). To find the murderer, we need to apply the *culprit clue.*

Write a function called **get_culprit**. It should take a culprit clue and a state. It should return the culprit identified by the culprit clue.

You can now modify your **solve** function to use the culprit clue to identify the culprit.

Next, write a function called **construct_result** to return a puzzle-solving result neatly.

It should take a status string and optionally, a solution state and a culprit. It should return a dictionary with the following keys:

- **status**: one of *solved*, *contradiction*, or *ambiguous*
- **assignments**: a dictionary mapping each suspect to their weapon and location
- **culprit**: the suspect identified as the culprit, if there is one

If there are no assignments or culprit, use an empty dictionary, None, or an empty list as appropriate.

Modify your **solve** function so that it uses **construct_result** to construct a solve result and returns the solve status, the assignment dictionary, and the culprit.

Question

This solver works by constructing every complete state before applying the clues. That approach is simple and useful for small puzzles, but it can become expensive very quickly.

Can you think of a way to improve the efficiency of our solver? Hint: think about how we could use the clues before constructing every complete state.

Part 2: Reading Reflections

These questions ask you to reflect on the first chapter of *AI Needs You* ("Shadow Self").

Question 1 (10 points)

Harding gives two examples of how AI is being applied to tasks that it simply isn't capable of solving. **What are they? Can you think of any more examples?**

Question 2 (10 points)

Harding argues that Amazon's solution to employee injuries was "not an idea rooted in how to adapt the job to human beings [...] but in how to adapt human beings to the job" (p. 18).

Can you think of other examples of how humans are being asked to adapt to AI rather than having AI that adapts to them?