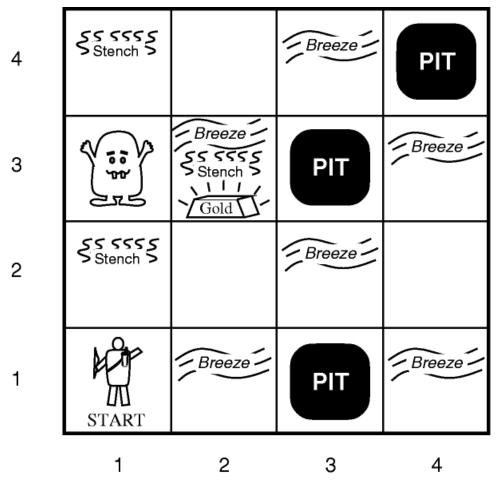
ASU CSE 571 | Fall 2022

Project 3: Logic agent

Project adapted from UoA, Intro to AI

Table of Contents



I smell a wumpus,
Time to infer my next move,
Find gold, and escape.

Introduction

A total of 24 points can be earned over the course of this project.

In this project, you will build a propositional logic (PL) agent capable of solving a variant of the Hunt The Wumpus game. This will mainly include constructing the agent's knowledge base.

The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can mostly ignore. It can be downloaded as a zip file on canvas as ***3.logicagent.zip.***

NOTE: The abbreviation **AIMA** used below refers to the Stuart Russell and Peter Norvig's book Artificial Intelligence: A Modern Approach, 3rd Edition, as well as the associated python code available from their AIMA GitHub repository.

Files you will edit and submit

`wumpus_kb.py`	Contains code to generate the wumpus knowledge base. You need to fill in most of these functions!

Files you will likely want to look at:

`wumpus.py`	The main file to execute the Hunt The Wumpus game. Contains code for generating `WumpusWorldScenario`s that combine a `WumpusEnvironment`and agent (either `Explorer` or `HybridWumpusAgent`) to play the game. Includes agent programs to drive game play manually (with and without a knowledge base). Also includes `__main__` command-line interface.
`wumpus_environment.py`	Implements main classes of the wumpus domain. The `Explorer` agent is a simple wumpus hunter agent that does not have a knowledge base. The `WumpusEnvironment` implements the physics and game play of the Hunt The Wumpus game.
`wumpus_agent.py`	Defines the `HybridWumpusAgent` (extends `Explorer`). This agent includes a knowledge base. The `agent_program` implements a hierarchy of reflexes that are executed depending on the percepts and knowledge state. This includes calls to `plan_route` and `plan_shot` that you will implement in `wumpus_planners.py`.
`logic.py`	AIMA code implementing propositional (PL) and first-order (FOL) logic language components and various inference algorithms. You will want to look at the relevant PL implementation.

`search.py`	AIMA code setting up basic search facilities; includes `Problem` class that you will implement in `wumpus_planners.py` for `plan_route` and `plan_shot`.

Files you likely don't need to look at:

`minisat.py`	Implements the interface to MiniSat, including translating AIMA PL clauses into DIMACS CNF, generating the DIMACS file read by MiniSat, using python's `sys` interface to call MiniSat, and reading the MiniSat results.
`agents.py`	AIMA code for the generic Agent and Environment framework.
`utils.py`	AIMA code providing utilities used by other AIMA modules.

Files to Edit and Submit: You will fill in portions of `wumpus_kb.py` and submit it as a zip file for the assignment.

Evaluation: Your code will be autograded for technical correctness. Please *do not* change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation -- not the autograder's output -- will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

In this project, we use an autograder that is different from our previous projects, which will be explained later.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. If you can't make our office hours, let us know and we will schedule a meeting time. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask. One more piece of advice: if you don't know what a variable does or what kind of values it takes, print it out.

Notes on terminology

In the discussion and comments/docs in the code I freely switch between referring to the agent **knowledge base**, **KB**, **kb** and **agent.kb**. These are all the same thing: the collective facilities for generating and storing propositional logic sentences.

I sometimes abbreviate **propositional logic** as **PL**.

PL sentences expressed in *full* propositional logic are sentences that may include all of the standard PL
connectives: *And/Conjunction* (`'&'`), *Or/Disjunction* (`'|'`), *Not/Negation* (`'~'`), *Conditional* (`'>>'`), and *Biconditional* (`'<=>'`) (see `logic.py` for full details about the PL python implementation). In the context of providing full PL sentences to the KB as rules for updating the agent's knowledge state, I will refer to these sentences as **axioms**.

When PL sentences are added to the KB, they are immediately converted to conjunctive normal form, **CNF**. This CNF representation is stored in the KB as a *list* of the individual disjunctive clauses (the list is treated as an implicit conjunction of its member clauses), and I will refer to the clauses in the list collectively as the KB **clauses**.

Setup: Install MiniSat, download project code, and run test

This project relies on the light weight, open source, cross platform SAT solver MiniSat (http://minisat.se). There are a variety of ways to obtain MiniSat; here are methods for the popular three platforms:

- **Mac OS X**: Several options:
  (NOTE: Unfortunately minisat is currently unavailable through Homebrew)
  (a) install using port (http://macports.org)
  ```
  $ port install minisat
  ```
  or
  (b) install from source following these instructions.
- **Linux**: How to install minisat on Ubuntu
- **Windows**: The following provides instructions for installing minisat using cygwin
  If you are encountering problems getting minisat to pass all 3 tests on Windows, please try the following variations:
  1. For Windows 10 users: it has been reported that **cygwin 32 bit** may need to be installed (you may need to uninstall cygwin if you have installed the 64 bit version).
     NOTE: Some report that after installing cygwin and the minisat package, the tests fail until the following is changed in `minisat.py`, line 183: Replace
     ```
     COMMAND = 'minisat %s %s > /dev/null'
     ```
     with
     ```
     COMMAND = 'minisat %s %s > ' + os.devnull
     ```
     or alternatively:
     ```
     COMMAND = 'minisat %s %s > NUL'
     ```
  2. A next option is to try using a linux virtual machine. Download virtualbox and a pre-made ubuntu image:
     https://www.virtualbox.org/
     https://www.osboxes.org/ubuntu/
     This should be straightforward to get running, and from there setting up minisat is just:
     ```
     $ sudo apt-get install minisat
     ```
     This setup has been tested. If you want to develop on your windows machine, but run the autograder in the virtualbox, you just need to install the extension pack
     Follow these instructions.
     Now you can write in your usual development environment, save to a directory

shared with your virtualbox "guest OS" and then go inside the guest to run the autograder.

After installing MiniSat and getting the Project 3 release from GitHub Classroom, you can then test the connection to MiniSat by executing the following from the command-line:

```
python wumpus.py -t
```

The three simple tests should pass. If they do not, or you get unexpected behavior, please post errors on piazza or contact us for help.

Welcome to the Wumpus Caves!

Having verified that the connection to MiniSat works, the next step is to familiarize yourself with the Hunt The Wumpus game. Execute the following at the command-line to to play the game:

```
python wumpus.py
```

This launches the interactive command-line interface to the game, running in "Manual" mode: you control the wumpus hunting agent. Entering '`env`' at any point will display an ascii-art representation of the current Wumpus environment state. For example, when executed in the first-time tick, you will see:

```
Scores: <Explorer>=0
  0   1   2   3   4   5     time_step=0
|---|---|---|---|---|---|
| # | # | # | # | # | # |  5
|---|---|---|---|---|---|
| # |   |   |   |   | # |  4
|---|---|---|---|---|---|
| # | W | G | P |   | # |  3
|---|---|---|---|---|---|
| # |   |   |   |   | # |  2
|---|---|---|---|---|---|
| # | ^ |   | P |   | # |  1
|---|---|---|---|---|---|
| # | # | # | # | # | # |  0
|---|---|---|---|---|---|
```

At the top, the current score of the Wumpus hunting agent (in the default manual mode, represented as an <Explorer>) is displayed. The x grid (horizontal) coordinates of the Wumpus environment are displayed across the row above the grid, and the y grid (vertical) coordinates are displayed down the right-side of the grid. Each grid cell represents a wumpus cave room. '#' represents a wall, and 'W', 'P', and 'G' represent the Wumpus, a Pit, and the Gold, respectively. The position of the wumpus hunter agent is represented by one of '^', '>', 'v', and '<', each of which show the direction (heading) the hunter agent is currently facing. We will also refer to the agent heading direction by the cardinal directions: '^' is North, '>' is East, 'v' is South, and '<' is West. At the start, as depicted in the above environment display, the agent is in location (1,1) and facing North.

Enter '`?`' at any time to see a complete list of available commands.

The goal of the game is to achieve the highest score. Each move costs one point, shooting the arrow (irrespective of whether it kills the Wumpus) costs 10 points, and leaving the caves (accomplished by executing '`Climb`' in the same location that the hunter agent started in (the cave 'entrance') *with* the Gold (i.e., having previously successfully '`Grab`'bed the Gold), earns 1000 points. Dying, by entering a square with the Wumpus or a Pit, costs 1000 points.

At each time step, the current percepts are represented by a list of the percept propositions; '~' represents that the proposition is (currently) False. For example, at time 0 (indicated in the brackets on the left) the percepts for the environment depicted above are:

```
[0] You perceive: ['~Stench', '~Breeze', '~Glitter',
'~Bump', '~Scream']
```

Play several games and see how the Wumpus environment state determines the percepts. Try dying by moving into the Wumpus square or a Pit (when you die, the game ends; simply relaunch to play again!). Shoot the Wumpus: you must execute '`Shoot`' while facing the Wumpus; when successful, the Wumpus will die and the following time step you will perceive the '`Scream`'; also note that the Wumpus is no longer represented by 'W', instead replaced by an 'X'. It is now safe to move into the Wumpus square. Also solve the game by moving to the Gold, executing '`Grab`', and move back to the exit/entrance and execute '`Climb`'.

You can load different Wumpus environment layouts by specifying either the name of an existing layout in the `wumpus/layouts/` directory (currently there are only two provided), or by specifying the path to a layout, using this command-line option:

```
python wumpus.py -l wumpus_4x4_book
```

(You can optionally specify the layout extension '.lay'.) The format of a layout specification file is very simple; here's the layout for the environment displayed above:

```
.,.,.,.
W,G,P,.
.,.,.,.
A,.,P,.
```

The file format specifies the Wumpus environment grid as a series of comma-separated wumpus-room specifications, with each row representing a row of rooms in the Wumpus environment. '.' represents an empty wumpus room, 'W' places a Wumpus (the knowledge base you will create below can only accommodate presence of exactly 1 Wumpus, no more or less, but in the manual game you can have multiple Wumpi), 'P' places a Pit, 'G' places the Gold (again, the KB will only support Grabbing 1 Gold), and 'A' is the location of the Wumpus hunting Agent (The Agent's heading is specified separately in code; North, or '^', is the default). You can also add Walls, represented by '#'. By default, the specified layout will be automatically surrounded by Walls when the Wumpus environment is constructed.

Take a look at the code comments and examples in `wumpus.py` to see how to construct a WumpusWorldScenario from a layout file or by specifying directly in code by constructing objects and assigning them to Wumpus environment locations.

General comments about the code structure and command-line options

There are two main classes that together make a working version of the Hunt the Wumpus game: the `WumpusEnvironment` and an instance of an agent are combined in a `WumpusWorldScenario`, defined in `wumpus.py`.

- The `WumpusEnvironment`, defined in `wumpus_environment.py`, represents the Wumpus cave environment, the position and state of all objects, and enforces the rules of the game (the game 'physics'), such as the effects of agent actions. The `step()` method advances the game one time step and includes calling the Wumpus hunter agent's `agent_program`.
- There are two classes that define Wumpus hunter agents. The `Explorer`, also defined in `wumpus_environment.py`, provides a minimal hunter agent skeleton, while the `HybridWumpusAgent`, defined in `wumpus_agent.py`, is a full agent implementation and includes a knowledge base and set of reflexes sufficient to solve any Hunt the Wumpus game (once you provide some code).

The main action of a wumpus hunter agent happens in its `agent_program`. There are three different `agent_program`s provided:

1. The function `with_manual_program` (in `wumpus.py`) takes an agent as input and replaces its `agent_program` with a "manual" `agent_program` in which the agent waits for commands from the command-line at each step. This is the agent that is run when launching the game from the command-line with (with or without the '`-l`' layout file option):
   ```
   python wumpus.py
   ```
   This version is most useful for playing the game and verifying you understand the "physics" of the Wumpus environment (i.e., the effects of actions on states of the world and subsequent percepts).
2. The function `with_manual_kb_program` (also in `wumpus.py`) works similar to `with_manual_program` except the agent also creates a knowledge base and the `agent_program` updates the knowledge base with new percepts and the action selected (by the human user) at each step. The user can also issue any relevant query about propositions in the knowledge base. This option is most useful for developing and debugging the axioms you provide for initializing and updating the knowledge-base. You will implement these axioms in the axiom generators in `wumpus_kb.py`. Until these axiom generators are implemented, your KB will not be fully functional.

The `with_manual_kb_program` agent program can be run by executing the following from the command-line (inside the `wumpus/`directory):

```
python wumpus.py -k
```

(You can also combine this with the '`-l`' option to load a specific layout.) Once launched, enter '`?`' to get the list of commands and complete list of available knowledge base queries.

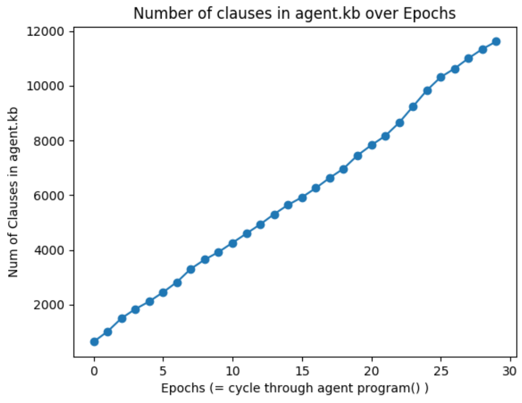
3. The `HybridWumpusAgent` (HWA) class is defined in `wumpus_agent.py` (it is a subclass of the `Explorer`). The HWA defines an `agent_program` that implements the Wumpus agent as specified in Figure 7.20, page 270 of AIMA 3rd Edition. Once you have correctly implemented the knowledge base axiom generators (in `wumpus_kb.py`), the HWA `agent_program` will be able to solve any (solvable) instance of the Hunt The Wumpus game. The HWA and its `agent_program` can be run from the command-line (inside the wumpus/ directory) with:

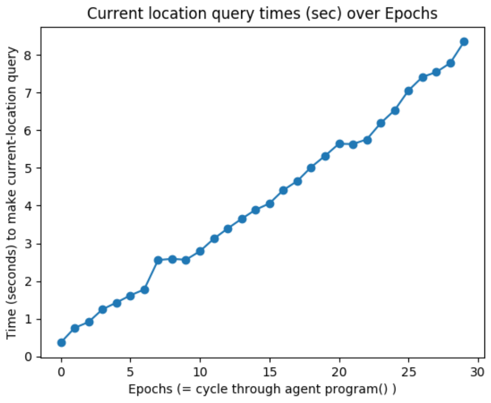
```
python wumpus.py -y
```

(Again, you can also combine this with the '`-l`' option; however, the '`-y`' option will override the '`-k`' option, if included). When run, the HWA `agent_program` will select all actions, so there is nothing for the human user to do but watch. As with the above options, the output to screen is intentionally very verbose so that you can follow along with each step in the execution.

To see what "correct" behavior should look like, I have provided the output of two runs of the fully-implemented HWA, running on the two provided layouts (in `layouts/`): HWA_lay_book_run.txt and HWA_lay_2_run.txt.

Also note that the number of clauses in the KB grows as the agent takes actions in the environment. This is expected: at each time step, new axioms are added to the knowledge base in order to represent change (based on percepts and actions). You will be implementing these axiom generators. The number of clauses created by your code should be on a par with (if not exactly the same) the number of clauses shown in HWA_lay_book_run.txt and HWA_lay_2_run.txt. . Also, your program should not be very slow. The following two figures graphically show the growth in the number of clauses and the time it takes to make one kind of query (current location) over the 30 steps of the HWA_lay_2_run.txt run.





NOTE: While you can construct layout specifications (either in layout files or in code) in any dimensions, it is recommend that you follow these general constraints (mainly due to working with the knowledge base; for general manual play without a KB, any size is fine):

- Do not create environments smaller than two rooms, as the knowledge base assumes there is 1 and only 1 Wumpus, so a 1-room cave would necessarily put the Wumpus in the same location as the entrance, leading to a contradiction!
- Do not create environments greater than 16 rooms, as the KB will grow in size too quickly and inference will bog down.)

Development

As the knowledge base and agent planner(s) developer, you have several choices as to how to proceed with development. The manual command-line interfaces to the wumpus game, defined above and implemented in `wumpus.py`, were designed to be a way by which you can directly observe the impact of code changes on knowledge base inference and agent behavior. However, you may find some or all development easier by incrementally instantiating and testing parts of the code; in this case, start with the `WumpusWorldScenario` to see how the main pieces are put together.

Propositional logic in python

The two agent programs you will be working with (found either in `with_manual_kb_program` in `wumpus.py` or the `HybridWumpusAgent` (HWA) class defined in `wumpus_agent.py`) make use of the `PropKB_SAT` knowledge base. This is defined in `wumpus_agent.py` and is a subclass the `PropKB` class defined in `logic.py`. As you'll see, these are actually quite simple, implementing the `tell()` method to add new sentences to the knowledge base, and the `ask()` method to query the knowledge base (`ask()` is an interface MiniSat). Assertions made to the knowledge base (often starting as PL syntax expressed in a string) are stored in the `clauses` field of the KB, and that is just a python list! (Robust implementations of propositional KBs have much more sophisticated storage). Representation of the assertions (sentences) themselves are built on top of the AIMA implementation of propositional logic. As you will be implementing axiom generators, it is important you understand how propositional sentences, initially expressed in strings, are turned into the underlying propositional logic representations. Take a look in `logic.py`. In particular, the following excerpt from the python prompt demonstrates some of the basic functionality.

```
In [1]: import logic

In [2]: a = '(A & B) >> ( ~(C | D) <=> E )'

In [3]: e = logic.expr(a)

In [4]: e
Out[4]: ((A & B) >> (~(C | D) <=> E))

In [5]: c = logic.to_cnf(e)

In [6]: c
Out[6]: ((~C | ~E | ~A | ~B) & (~D | ~E | ~A | ~B) & (E |
C | D | ~A | ~B))
```

```
        In [7]: logic.conjuncts(c)
        Out[7]: [(~C | ~E | ~A | ~B), (~D | ~E | ~A | ~B), (E | C
        | D | ~A | ~B)]
```

On line 2, I express a propositional sentence in a string, in the syntax of the AIMA propositional language. See the `Expr` class in `logic.py`. The function `expr()` parses that string and builds an `Expr`. The `Expr` object is designed to have a "pretty" python representation, an example of which is on line 4 above (this is accomplished by the definition of the `__repr__()` method in Expr; see this explanation). But keep in mind that this is an object! It has two main fields: the operator, `op`, and a list of arguments to the operator, `argslist`. Since the variable `e` currently references the Expr, we can inspect the `op` and `argslist` as follows:

```
        In [10]: e.op
        Out[10]: '>>'

        In [11]: e.argslist
        Out[11]: [(A & B), (~(C | D) <=> E)]
```

This shows that the Expr `e` refers to has as its operator the conditional symbol, and its argslist have two entries, the first being the antecedent to the conditional, in this case `A & B`, and the second is the consequent, `~(C | D) <=> E`. Each of these are also Exprs, with the first having the conjunction operator, `'&'`, with two args `A` and `B`. In this way, our original Expr referred to by `e` is actually the root of an Expr tree, allowing for representation of arbitrarily complex sentences.

Be sure to look at the docs for `Expr` and `expr()` in `logic.py` to understand how the PL syntax will be parsed from a string, in particular the note in the docs of `expr()` about operator precedence: `expr('P & Q ==> R & S')` will be parsed as `((P & (Q >> R)) & S)`, which may not be what you intended! To get the expected operator precedence enforced (i.e., `&` with higher precedence than `==>`), you must use `expr('(P & Q) ==> (R & S)')`. In general, it is best to use parentheses to enforce the precedence you intend!

Moving on with the original example, on line 5 I convert the full PL sentence into conjunctive normal form using the `to_cnf()` function; line 6 shows the result. This is still an `Expr` object; also, it is completely logically equivalent to the previous form expressed on lines 2 and 4. Whenever an Expr is `tell()`ed to the KB, the Expr will be converted to CNF. Then the conjuncts of the CNF will be extracted so that what is stored in the `clauses` store of the KB is a list of the individual clauses of the CNF. This is demonstrated on line 7.

Finally, a note about the use of MiniSat. MiniSat is a SAT solver, meaning that it searches for a satisfying assignment to a set of CNF clauses, return True if such an assignment is found (and in MiniSat's case, it also returns the satisfying assignment), or False if no assignment is found. This is a good building block but not by itself sufficient for propositional inference. In our case, we will not be doing full proposition inference, but instead asking whether individual propositions are entailed (True) or not (False) by the KB, or whether their truth cannot be determined. In order to determine which of these three possible outcomes is the case, the `ask()` method of `PropKB_SAT` make *two* calls to minisat, one in which the query variable (the proposition who's truth we're trying to determine) is assumed to be True, and one where it is assume to be False, and in both cases minisat determines whether that assertion conjuncted with all of the clauses in the KB is satisfiable. If the clauses + query are satisfiable in *both* cases, then that means the KB cannot determine whether the proposition is True or False. On the other hand, if one call to minisat is satisfiable, but the other not, then the proposition's truth is which of the calls was satisfiable. In general, you won't have to worry about these details, but it is important to understand how this is working!

Construct the knowledge base

OK, time to get to work! The first set of tasks is to fill in the axiom generators for the knowledge base. For this part of the project you will work in `wumpus_kb.py`, adding your code to all locations indicated by "`*** YOUR CODE HERE ***`". You will notice a pattern here, all of the methods you are implementing start with "axiom_generator_..." in their name. The doc strings to these functions describe the knowledge that you need to assert in propositional logic, with explanations of what the function arguments represent. The return values are assumed to be strings representing the PL assertions.

Section 7.7 of AIMA, starting on page 265, is a good guide for a number of the axioms you are required to implement. But beware, it is incomplete!

After the current percept sentence generator (which converts percept vectors into a sentence asserting percept propositions), there are two general classes of axiom generators you will construct: a set that generate axioms describing the initial state of knowledge, and axioms that represent changes over time (in particular, the successor-state axioms).

The assertions your generators will make will be built out of propositions. The first section of `wumpus_kb.py` defines every proposition that will appear in the KB. Because it would be very easy to add a malformed proposition symbol to the KB without knowing it, I have provided a set of proposition string builder functions, one for each type of proposition. Even though it is more verbose to use function calls like `percept_breeze_str(3)` rather than just '`Breeze3`', you'll be better off, as the KB itself won't tell you if you happened to have mistakenly asserted '`Breez3`' (that's a misspelling!) -- the KB will happily accept it and you'll be left to find your mistake through painful debugging! However, the choice is entirely up to you -- nothing about the grading will check whether you use these string-builders.

You will be working a lot with strings in this part of the project. Here are general python string functions that I found useful while building my solution:

- The plus sign is overloaded: `'a' + 'b'` results in `'ab'`
- The `join` method is very handy; the first string the join being called on will be inserted between the listed strings being joined (use an empty string to just concatenate the list of strings):
    - `''.join(['a','b','c'])` results in `'abc'`
    - `'-'.join(['a','b','c'])` results in 'a-b-c'
- The `format` method is your friend: `'string with {0}{1}'.format('stu', 'ff')` results in `'string with stuff'`

Points will be awarded, pending correct implementation, as follows (for a total of 24 points):

- `axiom_generator_percept_sentence` = 1 pt
- `axiom_generator_initial_location_assertions` = 0.5 pt
- `axiom_generator_pits_and_breezes` = 1 pt
- `axiom_generator_wumpus_and_stench` = 1 pt
- `axiom_generator_at_least_one_wumpus` = 1 pt
- `axiom_generator_at_most_one_wumpus` = 1 pt
- `axiom_generator_only_in_one_location` = 1 pt
- `axiom_generator_only_one_heading` = 1 pt
- `axiom_generator_have_arrow_and_wumpus_alive` = 0.5 pt
- `axiom_generator_location_OK` = 1 pt
- `axiom_generator_breeze_percept_and_location_property` = 1 pt
- `axiom_generator_stench_percept_and_location_property` = 1 pt
- `axiom_generator_at_location_ssa` = 4 pts
- `axiom_generator_have_arrow_ssa` = 1 pt
- `axiom_generator_wumpus_alive_ssa` = 1 pt
- `axiom_generator_heading_{north,east,south,west}_ssa` = 3 pts (for the set)
- `axiom_generator_heading_only_{north,east,south,west}` = 2 pts (for the set)
- `axiom_generator_only_one_action_axioms` = 2 pts

NOTE: While you are constructing the knowledge base generators, the KB should always be satisfiable. If it ever becomes unsatisfiable, then something you have added is leading to a contradiction! That is always a problem. To check for satisfiability of the KB, call the `minisat()` function in `wumpus_agent.py` with just the KB clauses, e.g., `minisat(kb.clauses)`; when running the wumpus.py with a KB (option `-k`) from the command-line, you can enter the '`kbsat`' command to do the same thing. Note, however, that just because the KB is satisfiable does not mean there are no other problems.

Autograder:

You may test your axioms using the provided test_axioms.py file.

- **`test_axioms.py`**
  - **`python test_axioms.py`** --> tests all functions
  - **`python test_axioms.py --fn=axioms_name`** --> tests particular axiom (use the function name for the axiom)

Once you have passed all test cases in test_axioms.py, it is time to put your code to test:

- **`python test_hwa_l1.py`** --> tests the hybrid agent with layout 1
- **`python test_hwa_l2.py`** → tests the hybrid agent with layout 2

◊ **Your code should pass these tests. Otherwise, you probably have introduced some wrong axioms. Failing each test will incur a penalty of 3 point (if you**

failed both tests, you will lose 6 points.). Also, these tests may take some time to run (but should be less than 30min), which is OK.

Submission

In order to submit your project, please upload the following files to **Project 3 on Canvas in a zip file**: `wumpus_kb.py`. Please **make sure they are in a directory named “submission”** and zip the whole directory as the autograder will not work otherwise.