Wampa World

This Wampa World logic-based agent homework was developed by Joseph Willem Ricci and Ben Swanson upon the original Jupyter Notebook by Lara Martin.

Gameplay

R2-D2 begins at the bottom-left location (0, 0) and must navigate the rectangular grid of unknown size. To navigate the grid, he can turn `left` or `right`, or move `forward`. R2-D2's goal is to rescue Luke by navigating to the room that contains Luke, `grab`ing him, and navigating back to (0, 0) to `climb` out of the cave.

Along the way, R2-D2 must avoid pits which he can fall into, and must avoid the Wampa, which can destroy him. In any adjacent room to a pit is a `breeze`. In any adjacent room to a Wampa is a `stench`. There can be `[0, 1]` Wampas, and there can be `[0, m*n - 2]` pits, where the grid is of size m x n. Each room can have 0 or 1 features from [`luke`, `pit`, `wall`, `wampa`].

R2-D2 is also carrying a blaster with one shot and infinite range, and can "shoot" the Wampa with a shot in its direction. If the Wampa is killed by the shot, a `scream` can be perceived in every room.

A `gasp` from Luke can be perceived by R2-D2 if they are both in the same room.

Finally, if R2-D2 moves `forward` into a wall, he perceives a `bump`. For this assignment, assume that R2 knows that the world is rectangular and that there are no internal walls. A hypothetical 1x1 grid would have walls at `{(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)}`.

Percepts: `['stench', 'breeze', 'gasp', 'bump', 'scream']`, where at some location, R2's percepts might look like `[None, 'breeze', None, 'bump', None]`

Actions: `'left', 'right', 'forward', 'grab', 'climb', 'shoot'`

To run the game, run the following command in your terminal from your homework's directory

```
python wampa_world.py <scenario>
```

File Structure

agent.py

Contains R2-D2's constructor including initial knowledge base class, KB. Familiarize yourself with the knowledge base, as the agent class will be using it for logical inference.

TODOs:

`adjacent_rooms(self, room):` - Returns a set of tuples representing all adjacent rooms to 'room'.

Input: `(2, 3)`

Expected output: `{(1, 3), (3, 3), (2, 2), (2, 4)}`

`record_percepts(self, sensed_percepts, current_location):` - Update the percepts in agent's KB with the percepts sensed in the current location, and update visited_rooms and all_rooms accordingly.

`enumerate_possible_worlds(self)` - Return the set of all possible worlds, where a possible world is a tuple of (pit_rooms, wampa_room), pit_rooms is a tuple of tuples representing possible pit rooms, and wampa_room is a tuple representing a possible wampa room. Since the goal is to combinatorially enumerate all the possible worlds (pit and wampa locations) over the set of rooms that could potentially have a pit or a wampa, we first want to find that set. To do that, subtract the set of rooms that you know cannot have a pit or wampa from the set of all rooms. For example, you know that a room with a wall cannot have a pit or wampa. Then use itertools.combinations to return the set of possible worlds, or all combinations of possible pit and wampa locations. You may find the utils.flatten(tup) method useful here for flattening wampa_room from a tuple of tuples into a tuple. The output of this function will be queried to find the model of the query, and will be checked for consistency with the KB to find the model of the KB.

From step 0, the initial position on scenario S1:

```
. . . P
W L P .
. . . .
^ . P .
```

Expected output `{(), ()}`. Since all adjacent rooms are known to be safe when there is no breeze and no stench in the current room, so all rooms in `KB.all_rooms` are known to be safe, so there are no possible worlds with pits or wampas.

From step 1, in the following position resulting from a forward action:

```
. . . P
W L P .
^ . . .
. . P .
```

Expected output, `possible_worlds =`

```
{
    ((), ()),
    ((), (-1, 1)),
    ((), (0, 2)),
    ((), (1, 1)),
    (((-1, 1),), ()),
    (((-1, 1),), (0, 2)),
    (((-1, 1),), (1, 1)),
    (((-1, 1), (0, 2)), ()),
    (((-1, 1), (0, 2)), (1, 1)),
    (((-1, 1), (1, 1)), ()),
    (((-1, 1), (1, 1)), (0, 2)),
    (((-1, 1), (1, 1), (0, 2)), ()),
    (((0, 2),), ()),
    (((0, 2),), (-1, 1)),
    (((0, 2),), (1, 1)),
    (((1, 1),), ()),
    (((1, 1),), (-1, 1)),
    (((1, 1),), (0, 2)),
```

```
    (((1, 1), (0, 2)), ()),
    (((1, 1), (0, 2)), (-1, 1))
}
```

which corresponds to every combination of pit locations and wampa location in the rooms that could potentially have a pit or wampa, `(-1, 1), (0, 2), (1, 1)`. Note, this is not doing "model checking" (checking consistency with the KB). This is simply enumerating all possible pit and wampa locations. The set of possible worlds will be used to check against our KB and to query with our queries.

`pit_room_is_consistent_with_KB(self, room)` - Return True if the room could be a pit given breeze in KB, False otherwise. A room could be a pit if all adjacent rooms that have been visited have had breeze perceived in them. A room cannot be a pit if any adjacent rooms that have been visited have not had breeze perceived in them. This will be used to find the model of the KB.

`wampa_room_is_consistent_with_KB(self, room)` - Return True if the room could be a wampa given stench in KB, False otherwise. A room could be a wampa if all adjacent rooms that have been visited have had stench perceived in them. A room cannot be a wampa if any adjacent rooms that have been visited have not had stench perceived in them. This will be used to find the model of the KB.

`find_model_of_KB(self, possible_worlds)` - Return the subset of all possible worlds consistent with KB. possible_worlds is a set of tuples (pit_rooms, wampa_room), pit_rooms is a set of tuples of possible pit rooms, and wampa_room is a tuple representing a possible wampa room. A world is consistent with the KB if wampa_room is consistent and all pit rooms are consistent with the KB.

Input: `possible_worlds`, from step 1.

Output, `model_of_KB =`

```
{
    ((), (-1, 1)),
    ((), (0, 2)),
    ((), (1, 1))
}
```

which is the subset of worlds from `possible_worlds` that "checked out" with the model of the KB. No breeze has been perceived yet, so the only worlds that are consistent with the KB are those with no pits. And with stench perceived in `(0, 1)`, rooms `{(-1, 1), (0, 2), (1, 1)}` are the only rooms that `wampa_room_is_consistent_with_KB` returns True for.

If we consider the state of S1 after taking the following actions from the initial position: `forward`, `left`, `left`, `forward`, `left`, `forward`

To end up in the following state:

```
. . . P
W L P .
. . . .
. > P .
```

After recording percepts in that position, we would expect `model_of_KB =`

```
{
    (((1, -1),), (-1, 1)),
    (((1, -1),), (0, 2)),
    (((2, 0),), (-1, 1)),
    (((2, 0),), (0, 2)),
    (((2, 0), (1, -1)), (-1, 1)),
```

```
    (((2, 0), (1, -1)), (0, 2))
}
```

which is all of the possible worlds `(pit_rooms, wampa_room)` where `pit_room_is_consistent_with_KB(room` returns True for all rooms in `pit_rooms` and `wampa_room_is_consistent_with_KB(room)` returns True for `wampa_room`.

`find_model_of_query(self, query, room, possible_worlds)` - Where query can be "pit_in_room", "wampa_in_room", "no_pit_in_room" or "no_wampa_in_room", filter the set of worlds according to the query and room.

Inputs: `"wampa_in_room", (0, 2), possible_worlds` from step 1.

Output:

```
{
    ((), (0, 2)),
    (((-1, 1),), (0, 2)),
    (((-1, 1), (1, 1)), (0, 2)),
    (((1, 1),), (0, 2))
}
```

which is the subset of `possible_worlds` that contain `(0, 2)` in the `wampa_room` index.

`inference_algorithm(self):`

-First, make some basic inferences: 1. If there is no breeze or stench in current location, infer that the adjacent rooms are safe. 2. Infer wall locations given bump percept. 3. Infer Luke's location given gasp percept. 4. Infer whether the Wampa is alive given scream percept.

-Then, infer whether each adjacent room is safe, pit or wampa by following the backward-chaining resolution algorithm: 1. Enumerate possible worlds. 2. Find the model of the KB, i.e. the subset of possible worlds consistent with the KB. 3. For each adjacent room and each query, find the model of the query. 4. If the model of the KB is a subset of the model of the query, the query is entailed by the KB. 5. Update KB.pits, KB.wampa, and KB.safe_rooms based on any newly derived knowledge.

`all_safe_next_actions(self)` - Define R2-D2's valid and safe next actions based on his current location and knowledge of the environment.

`choose_next_action(self)` - Choose next action from all safe next actions. You may want to prioritize some actions based on current state. For example, if R2-D2 knows Luke's location and is in the same room as Luke, you may want to prioritize 'grab' over all other actions. Similarly, if R2-D2 has Luke, you may want to prioritize moving toward the exit. You can implement this as basically (randomly choosing between safe actions) or as sophisticated (optimizing exploration of unvisited states, finding shortest paths, etc.) as you like.

scenarios.py

Contains six scenarios, S1, S2, S3, S4, S5 and S6 to test your program with. Feel free to write your own!

utils.py

Contains miscellaneous helper and utility functions. You may want to utilize `flatten(tup)`, `get_direction(degrees)`, `is_facing_wampa(agent)`.

visualize_world.py

Is called during gameplay to visualize the current state of the world.

wampa_world.py

Contains the WampaWorld class which defines gameplay and the main gameplay loop

Submission

Upload your file agent.py to the Gradescope assignment submission.

A (non-exhaustive) suite of test cases for each method can help you troubleshoot and ascertain whether you are along the right track.

Interpreting Autograder Results

"Items in the first set but not the second:" are items in your program's result that are not in the expected result.

"Items in the second set but not the first:" are items in the expected result that are not in your program's result.

Tips

Debugging

You can debug with command line arguments (i.e. `S1`, `S2` ...) by setting up the debug configuration to do so. In VS Code, search "debug Current File with Arguments".

You can also reproduce, inspect and write your own test cases by importing the necessary modules into a test file, and manually scripting actions and inspections. For example:

```
from scenarios import *
from wampa_world import WampaWorld
from visualize_world import visualize_world

w = WampaWorld(S1)
visualize_world(w)
w.agent.record_percepts(w.get_percepts())
w.agent.inference_algorithm()
w.take_action("forward")
visualize_world(w)
w.agent.record_percepts(w.get_percepts())
possible_worlds = w.agent.enumerate_possible_worlds()
print(w.agent.find_model_of_KB(possible_worlds))
```

FEWPHE (Frequently Encountered Wonderings, Possible Hypotheses, and Elenchi)

Instead of an "FAQ" this "FEWPHE" is modelled on the stages of the Socratic Method and is meant to gently steer your line of inquiry toward the immensely instructive and rewarding experience of *discovering* solutions (as opposed to being told the solution). First, *wonder*! E.g. "Why isn't this

working?". Second, propose a *hypothesis*. E.g. "I see expected behavior X is not working, so perhaps it has something to do with Y". Third, go through *Elenchus* with your interlocutor (TA); one or many questions in response that scrutinze and test your presumptions in order to stimulate your own critical thinking. Fourth, *revise* your hypothesis and fifth, *act accordingly*.

Usage: "FEWPHE! I'm so glad I *discovered* the answer!"

1

Wondering: "My implementation is passing S1 - S5, but failing S6. Why?

Possible Hypothesis: "Perhaps it has something to do with how R2 is not inferring (0, 2) to be safe."

Elenchus: When you inspect the initial scenario, what do you see in (0, 2)?

2

Wondering: "My implementation is passing all method test cases but timing out on the `run_game` test cases. Why?"

Possible Hypothesis: "Perhaps it has something to do with how my `take_next_action` method is choosing actions randomly."

Elenchus: That is quite possible! Try thinking about what bits of logic you could incorporate to prioritize exploration of unvisited locations, for example. What other situations are there that you could prioritize certain actions over others? It is also possible that your code could be made more efficient in certain areas. What techniques do you know of to made certain blocks of logic more efficient? Which method in particular is likely the most computationally intensive?