Homework Problem 1

Formal Methods in Robotics (Fall 2019)

In this problem, we want to construct a **Gridworld** transition system (TS) for a two-player turn-based game between players `P1` and `P2` using `IGLSynth` tool. Complete the ***TODO***'s in `generate_graph` function in `gw_graph.py`.

You will need to be familiar with following objects.

- Class Graph : See `iglsynth.util.graph` module
- Class TSys: See `iglsynth.game.tsys` module
- Class Gridworld: See `iglsynth.game.gridworld` module
- Class Action: See `gw_graph.py`

Remark 1: In IGLSynth, every transition system has a `kind`, which is either `TURN-BASED` OR `CONCURRENT`. In this HW problem, implement the construction for `TURN-BASED` transition system.

Remark 2: Why is Action Class in gw_graph.py?

Because the `Action` class API is still experimental.

An `Action` is a `Callable` object that acts on a `Vertex` object to return a new `Vertex` object. See definition of `N, E, S, W, ...` actions to understand how to define and implement actions.

[Any suggestions or feedback on Action class API would be greatly appreciated.]

Remark 3 Thanks to the feedback given by the students, I have updated IGLSynth API to make it more intuitive, while maintaining its efficiency. Here's a quick primer on the changes.

Primer on IGLSynth v0.2

In IGLSynth v0.1, graph was represented as $G = \langle V, E, vp, ep, gp \rangle$, where users needed to maintain vp, ep, gp using an unintuitive interface. In IGLSynth v0.2, we now have **Vertex** and **Edge** classes associated with every graph to maintain vertex and edge properties. This means that every sub-class of Graph class will have its own structure of vertex and edge objects that carry necessary properties. See following example.

Example: TSys.Edge and Gridworld.Edge

Following is a snippet of `TSys.Edge` class. An edge of TS is defined as 3-tuple (u, v, a) where a is an action. In case of TS class, action may be any `PyObject`. When the user calls `TS.add_edge(e)`, internally the `add_edge` function checks whether input parameter `e` is an instance or a sub-class of `TSys.Edge` class. If not, then it throws an exception.

```
        class Edge(Graph.Edge):
        ACTIONS = set()

        def __hash__(self):
            return (self._source, self._target).__hash__()

        def __init__(self, u: 'TSys.Vertex', v: 'TSys.Vertex', act):
            super(TSys.Edge, self).__init__(u=u, v=v)
            self._act = act
            TSys.Edge.ACTIONS.add(act)

        def __repr__(self):
            return f"Edge(source={self._source}, target={self._target}, act=
{self._act})"

        def __eq__(self, other: 'TSys.Edge'):
            return self.source == other.source and self.target == other.target
and self.act == other.act

        @property
        def act(self):
            return self._act
```

Now, we define a `Gridworld` as a transition system where all actions are from `[N, E, S, W, NE, NW, SE, SW, STAY]`. Therefore, we define a new edge class `Gridworld.Edge` that inherits `TSys.Edge` and asserts that above condition. See the code snippet below.

```
    class Edge(TSys.Edge):
        __hash__ = TSys.Edge.__hash__

        def __init__(self, u: 'Gridworld.Vertex', v: 'Gridworld.Vertex', act):
            assert act in Gridworld.ACTIONS or all(a in Gridworld.ACTIONS for a
in act)
            super(Gridworld.Edge, self).__init__(u=u, v=v, act=act)
```

A similar argument holds for `<Graph-SubClass>.Vertex` objects. Note that `<Graph-Subclass>.add_vertex` will check for the input vertex to be of `<Graph-Subclass>.Vertex` type.

Running the code

1. If using PyCharm, just `Run` the file. PyCharm will take care of configurations for you.
2. If using terminal, then

```
PC$ docker exec -it <docker-container name> /bin/bash
Docker$ cd /home/iglsynth/
Docker$ python3 -m pytest FMR_HW/gw_graph.py
```

Check if all tests are pass or not. Ignore any warnings.