COMS 4281 - Intro to Quantum Computing

Problem Set 5: Qubit Tug-of-War

Due date: December 18, 11:59pm.

For the second part of Problem Set 5, you should get into teams of at most 3 (**this will be strictly enforced**), and as a team you will come up with strategies for a quantum video game we call **Qubit Tug-of-War**.

Please register your team with by filling out https://forms.gle/z5FvHubfoZvsPqxk6

Rules of the game

There are two teams in this game, Team $|0\rangle$ and Team $|1\rangle$. They take turns getting control of a rotating qubit $|\psi\rangle$. In between the teams' turns, the qubit rotates by either θ or $-\theta$ depending on the current **direction of rotation**. At the beginning, the qubit starts rotating in the direction of θ.

During their turn, each team can choose to perform an **action** from their **hand** (think of an action as a "card"). Here are possible actions that a team can have:

1. *Measure*: this measures $|\psi\rangle$ in the standard basis.
2. *X*: apply Pauli X gate to $|\psi\rangle$.
3. *Z*: apply Pauli Z gate to $|\psi\rangle$.
4. *H*: apply the Hadamard H gate to $|\psi\rangle$.
5. *R*: reverse the direction of the qubit rotation (i.e. qubit gets rotated by $-\theta$ instead of θ and vice versa).

A team can also choose to pass (do nothing). If they perform an action, it gets removed from their hand.

Each team starts with an empty hand. At random intervals, each team gets a random action, unless they already have 5 actions in a hand, or have already received the maximum of $M = 20$ actions throughout the game.

At the start of their turn, each team learns:

1. What actions were performed in the previous round (by Team $|0\rangle$ and Team $|1\rangle$).
2. If there were measurements, what the outcome were.

Important: each team is responsible for calculating the state of the qubit based on the history of actions and measurement results.

There are a total of $T = 100$ turns for each team. At the end, the qubit $|\psi\rangle$ is measured, and if the outcome is $|b\rangle$, then Team $|b\rangle$ wins.

Game Parameters

We've specified the parameters with default values as follows:

- Number of rounds: $T = 100$
- Action budget (this maximum number of actions a team can receive): 20
- Hand size (maximum number of actions they can have at one time): 5
- θ (angle of rotation): $2\pi/100$
- The relative frequency of the actions are as follows:
  - Measure: 5%
  - X, Y, H : 25%
  - Reverse: 20%
- Initial state of qubit: $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.

Programming your QToW Bot

Instead of playing this game in person, your team will program a **bot** to play for you. Your bot will be pitted against other teams' bots.

We will provide a few basic bots to help you develop your own strategy. One of the bots is called the `RandomBot`, which chooses actions randomly.

```
In [ ]: %run GamePlayer.py

class RandomBot(GameBot):
    def play_action(self,
                    team: int,
                    round_number: int,
                    hand: List[GameAction],
                    prev_turn: List) -> Optional[GameAction]:

        #this is the probability that it chooses to play an action
        p = 0.2

        #if the hand is non-empty and we flip a coin and it lands heads with pr
        #choose a random action
        if len(hand) > 0 and np.random.random() < p:
            action = random.choice(hand)
            return action

        #otherwise, don't play an action
        return None
```

All bots are a Python class that inherits `GameBot`. There is just one function to implement, called `play_action`, which (aside from `self`), has the following arguments:

1. `team`: this indicates whether the bot is team 0 or team 1.
2. `round_number`: this indicates which round (between 0 and $T = 200$) the bot is currently playing
3. `hand`: this is a list of `GameAction`s that is available for the bot to play. There are several GameActions that can be played:
   - `GameAction.MEASURE`
   - `GameAction.PAULIX`
   - `GameAction.PAULIZ`
   - `GameAction.HADAMARD`
   - `GameAction.REVERSE`

So an example of a `hand` could be the following list:
`[GameAction.MEASURE,GameAction.PAULIZ,GameAction.PAULIZ,GameAction.REVERS`
Meaning that the bot can measure, apply Pauli Z (twice), or reverse the direction of the qubit rotation.

Every few rounds, a random action will be added to the bot's hand, unless (a) the hand is full, or (b) $M = 20$ actions have already been added.

4. `prev_turn`: this is a Python dictionary that specifies what happened in the previous turns. The keys of this dictionary are: `team0_action`, `team1_action`, `team0_measurement`, `team1_measurement`. The `team0/1_action` entries store the `GameAction` that was performed by the team, or `None` if they didn't perform an action. If a team performed a measurement action, then the `team0/1_measurement` entries indicate the result of a measurement(either `[1,0]` or `[0,1]` to indicate collapsing to $|0\rangle$ or $|1\rangle$). If the team didn't perform a measurement action, then this entry will be set to `None`.

For example, suppose in the previous turn Team $|0\rangle$ used a HADAMARD action and Team $|1\rangle$ measured. Then the dictionary would look like this:

`prev_turn = {'team0_action': GameAction.HADAMARD, 'team1_action': GameAction.MEASURE, 'team0_measurement': None, 'team1_measurement': = [1,0]}`

You may find it helpful to design a bot, for example, that chooses its actions based on the history of yours and the other team's past actions.

Here's another example of a bot. It's not very intelligent, but should illustrate some things you can do.

In [ ]:
```
class WeirdBot(GameBot):
    def play_action(self,
                    team: int,
                    round_number: int,
                    hand: List[GameAction],
                    prev_turn: List) -> Optional[GameAction]:


        #if the round is even, check if there's a PauliX operation, and play it
        if GameAction.PAULIX in hand and team % 2 == 0:
            return GameAction.PAULIX

        #otherwise if round is odd, try to measure
        if GameAction.MEASURE in hand and team % 2 == 1:
            return GameAction.MEASURE


        return None
```

Let's pit `RandomBot` versus `WeirdBot` against each other! When creating the bots, you need to pass the name of the bot as an argument.

In [ ]:
```
#create the bots
weirdbot = WeirdBot("The Weirdos")
randombot = RandomBot("The Randos")

#create the gameplayer with the weirdbot as team |0> and randombot as team |1>
gp = GamePlayer(weirdbot,randombot)

#play the game, get the winning state
winning_state = gp.play_rounds()

if winning_state[0] == 1:
    print("===Weird bot beats Random bot!===")
else:
    print("===Random bot beats Weird bot!===")
```

To see a transcript of the game, you can run the following code. It shows you the state of the qubit at each round, the actions that were performed, and which actions were added to the bots' hands. If you're running this in IBM Quantum Lab, you probably want to right click on the output cell and select "Enable scrolling for outputs".

In [ ]:
```
#get the event log
log = gp.get_event_log()
print(log)
```

Now try creating your own bot by writing code below.

In [ ]:
```
'''
Insert high-level explanation of your strategy here. Why did you design this st
When should it work well, and when should it have trouble?
'''
class MyStrategy(GameBot):

    '''
        Initialize your bot here. The init function must take in a bot_name.
        You can use this to initialize any variables or data structures
        to keep track of things in the game
    '''
    def __init__(self,bot_name):
        self.bot_name = bot_name        #do not remove this

    def play_action(self,
                    team: int,
                    round_number: int,
                    hand: List[GameAction],
                    prev_turn: List) -> Optional[GameAction]:


        ##### IMPLEMENT AWESOME STRATEGY HERE ##################


        #######################################################
        return None
```

Submitting your bots

Once you have a bot ready to play against others, you should copy your Bot code (e.g. the `MyStrategy` class) into a separate python file (see the example file `example_bot.py`) and also include the following includes:

`from typing import List, Optional import numpy as np import random from GamePlayer import *`

Then, visit the Qubit Tug-of-War website at https://henryyuen.net/fall2022/qtugofwar and click "Upload your bot". You will need your Team ID. If you don't have one, register your team at https://forms.gle/z5FvHubfoZvsPqxk6 and e-mail Prof. Yuen to get a Team ID.

A tournament between all the bots will be played daily, and the Leaderboard will be updated to show the current performance. You can browse the statistics and examine sample transcripts of games.

Some constraints

- Your bot python file cannot exceed 30kb.
- If your bot crashes or throws an error, it automatically loses.
- If your bot runs for too much time, it automatically loses.
- Do not include any custom python packages, because it is unlikely we will have them.

The Leaderboard will be active from **December 1** to **December 15**. It will not be updated after December 16 so there will be a couple days for teams to optimize their bots in secret!

Your bot is due **December 18, 11:59pm**. A final Tournament will be played on **December 19** to determine the winning teams.

Late bots are not accepted!

Grading

Every team must work independently on their bot (teams should not share code). Your bots will be graded in the following way. Points will be awarded based on a combination of:

1. Effort and creativity,
2. Whether your code includes in the beginning a discussion about your strategy and your rationale behind the design choices, and
3. Performance against other bots.

As a target, you should try to design a bot that consistently beats `RandomBot` at least 60% of the time.

In [ ]:

```
```