Stanford CME 241 (Winter 2021) - Assignment 2

Stock Price MRP

Chih-Hsuan 'Carolyn' Kao (chkao831@stanford.edu)

Feb 3rd, 2021

Extend one of the Stock Price examples in Chapter 1 to be a Markov Reward Process by defining a Reward R_t that is a function f of the Stock price X_t at each time t. Program it as a class that implements the interface of the @abstractclass MarkovRewardProcess and allow flexibility in specifying your own function f. This is an infinite-states, non-terminating MRP.

In [1]:

```
import sys
sys.path.append('/Users/chih-hsuankao/Desktop/CME241/RL-book/')
```

In [2]:

```
from dataclasses import dataclass
from typing import Callable, Mapping, Optional, Tuple
import numpy as np
import itertools
from rl.distribution import Categorical, Distribution, Choose
from rl.markov_process import MarkovProcess
from rl.markov_process import MarkovRewardProcess
from rl.gen_utils.common_funcs import get_logistic_func, get_unit_sigmoid_func
from rl.chapter2.stock_price_simulations import\
    plot_single_trace_all_processes
from rl.chapter2.stock_price_simulations import\
    plot_distribution_at_time_all_processes
```

Reference: Git code base "https://github.com/TikhonJelvis/RL-book" of textbook (Foundations of Reinforcement Learning with Applications in Finance by Ashwin Rao and Tikhon Jelvis), at rl/chapter2/stock_price_mp.py

In [3]:

```
@dataclass(frozen=True)
class StateMP2:
    price: int
    is_prev_move_up: Optional[bool]
```

In [4]:

```
handy_map: Mapping[Optional[bool], int] = {True: -1, False: 1, None: 0}
```

In [5]:

```
@dataclass
class StockPriceMRP2(MarkovRewardProcess[StateMP2]):

    alpha2: float = 0.75
    reward_func: Callable[[StateMP2], float] = lambda _: 0.0

    def up_prob(self, state: StateMP2) -> float:
        return 0.5 * (1 + self.alpha2 * handy_map[state.is_prev_move_up])

    def transition_reward(self, state: StateMP2)\
            -> Optional[Distribution[Tuple[StateMP2, float]]]:

        up_p = self.up_prob(state)

        return Categorical({
            (StateMP2(state.price + 1, True), self.reward_func(state)): up_p,
            (StateMP2(state.price - 1, False), self.reward_func(state)): 1 - up_p
        })
```

In [6]:

```
def process2_price_traces(
    start_price: int,
    alpha2: float,
    time_steps: int,
    num_traces: int,
    reward_func: Callable
) -> np.ndarray:

    mp = StockPriceMP2(alpha2 = alpha2, reward_func = reward_func)

    start_state_distribution = Constant(
        StateMP2(price=start_price, is_prev_move_up=None)
    )
    return np.vstack([
        np.fromiter((s.price for s in itertools.islice(
            mp.simulate(start_state_distribution),
            time_steps + 1
        )), float) for _ in range(num_traces)])
```

In [7]:

```
def some_function(state: StateMP2):
    return state.price - 5

stock_mrp = StockPriceMRP2(0.75, some_function)
```