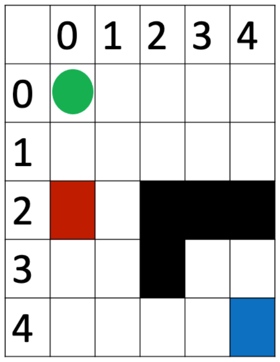
Homework #3

1) Given the following GridWorld, where R[blue] = 100, R[red] = -100, R[white] = 0, with $\gamma = 0.9$, compute value function using policy iteration with $\pi$ defined as:



```
policy = {
    (0,0): 'R', (0,1): 'D', (0,2): 'L', (0,3): 'L', (0,4): 'L',
    (1,0): 'R', (1,1): 'D', (1,2): 'L', (1,3): 'L', (1,4): 'L',
    (2,0): 'R', (2,1): 'D',
    (3,0): 'D', (3,1): 'D',             (3,3): 'R', (3,4): 'D',
    (4,0): 'R', (4,1): 'R', (4,2): 'R', (4,3): 'R', (4,4): 'D',
}
```

You can use the supplied auxiliary class GridWorld that builds a gridworld for you.

To create this gridworld, you need to call it like:

```
rewards = {
    (2,0): -100,
    (4,4):  100
}
walls = [(2,2), (2,3), (2,4), (3,2)]
g = GridWorld(5, 5, start_position=(0, 0),
        pass_through_reward=0, rewards=rewards, walls=walls)
```

2) We will modify the gridworld to accept probabilistic transitions, so that we can specify p(s',r|s,a). We will still assume r(s') to be a function of the next state, so that we can separate the probability into p(s'|s,a).

This is the initial specification of the GridWorld class.

```
class GridWorld:
    def __init__(self, rows, columns, start_position=None,
            pass_through_reward=0, rewards={}, probs={}):

        self.rows = rows
        self.columns = columns
        self.world = pass_through_reward * np.ones((rows, columns))

        if isinstance(start_position, type(None)):
            r = np.random.choice(rows)
            c = np.random.choice(columns)
            start_position = (r, c)

        self.start_position = start_position
        self.rewards = rewards

        for position in rewards:
          self.world[position] = rewards[position]

        # probs is a data structure like:
        # {
        #    (pr,pc) : { 'U': { (nr1,nc1): p1, (nr2,nc2): p2, ... }, 'D': ...}
        # }
        # with p1 + p2 + ... = 1

        self.probs = probs

        self.reset()
```

You need to define the following functions:

```
    def set_state(self, position):
    def get_state(self):
    def next_state(self, s, a):
        # returns dictionary containing distribution p(s’|s,a)
        # like { ns1’: 0.1, ns2’: 0.2, ns3’: 0.5, ns4’: 0.2 }
    def reset(self):
    def actions(self, position=None):
        # given a position (state), returns all possible actions
        # or if no position is passed, returns possible actions
        # from current state.
    def move(self, action):
        # moves from current state and action. As this is a
        # distribution, you need to pick one next state randomly
         # based on the probability of the transition.
    def all_states(self):
    def game_over(self):
```

Note that to test this you should invoke the test routine presented.  For the policy defined in problem 1, compute the value function.

3)  Compute the optimal policy using policy iteration algorithm for question #2.

4) Compute the optimal policy using value iteration algorithm for question #2.