In [138]:
```python
# Anthony Galczak
# CS238 - Midterm 3
```

In [139]:
```python
#Q1
import itertools

class POMDP:

    def __init__(self, horizon, num_exams, model_param, r_false_fail,
                 r_false_pass, p_false_fail, p_false_pass, p_qual):

        self.horizon = horizon
        self.num_exams = num_exams
        self.model_param = model_param
        self.r_false_fail = r_false_fail
        self.r_false_pass = r_false_pass
        self.p_false_fail = p_false_fail
        self.p_false_pass = p_false_pass
        self.p_qual = p_qual

pom = POMDP(2, 3, 0.3, -1, -1, 0.15, 0.2, 0.9)
```

In [140]:
```python
#Q2
def get_states(pom):
    n = pom.num_exams
    possible_states = [0,1]
    return [i for i in itertools.product(possible_states, repeat = n)]
def get_actions(pom):
    n = pom.num_exams
    possible_actions = [0,1]
    return [i for i in itertools.product(possible_actions, repeat = n)]
def get_observations(pom):
    n = pom.num_exams
    possible_observ = [-1,0,1]
    return [i for i in itertools.product(possible_observ, repeat = n)]

print(len(get_observations(pom)))
```

```
27
```

In [141]:

```python
#Q3
def transition(pom, state, action, next_state):
    # Edge case, if we chose to take no action.
    if sum(action) == 0:
        if state == next_state:
            return 1
        else:
            return 0
    # Probability to pass any one exam.
    prob_pass = pom.model_param / sum(action)
    transition_prob = 1
    for i in range(len(action)):
        if action[i] == 1: # We are taking this exam.
            if state[i] == 0 and next_state[i] == 0: # We fail
                transition_prob *= (1 - prob_pass)
            elif state[i] == 0 and next_state[i] == 1: # We pass
                transition_prob *= prob_pass
            # We don't have to handle the case where it's 1,1 since
            # we can't "unqualify"
    return transition_prob

print(transition(pom, [0,1,0], [1,1,0], [1,1,0]))
# Testing 2 exam passes at once.
print(transition(pom, [0,0,0], [1,1,0], [1,1,0]))
print(transition(pom, [0,1,0], [0,0,0], [0,1,0]))
```

```
0.15
0.0225
1
```

In [142]:

```python
#Q4
def reward(pom, state, action, t):
    if sum(state) == 3 and sum(action) > 0 and t == 3: # False fail
        return pom.r_false_fail
    elif sum(state) < 3 and sum(action) == 0: # False pass
        return pom.r_false_pass
    else:
        return 0

print(reward(pom, [1,0,0], [0,0,0], 3))
```

```
-1
```

In [143]:
```
# Q5
def observation(pom, action, next_state, observ):
    prob = 1
    for i in range(len(observ)):
        if action[i] == 1:
            # If we didn't take that test, return 0.
            if observ[i] == -1:
                return 0
            # Student fails, and is unqualified
            elif observ[i] == 0 and next_state[i] == 0:
                prob *= (1 - pom.p_false_fail)
            # Student fails, but is qualified
            elif observ[i] == 0 and next_state[i] == 1:
                prob *= pom.p_false_pass
            # Student passes, but is unqualified
            elif observ[i] == 1 and next_state[i] == 0:
                prob *= pom.p_false_fail
            # Student passes, and is qualified
            elif observ[i] == 1 and next_state[i] == 1:
                prob *= (1 - pom.p_false_pass)
        else:
            prob /= 3

    return prob

print(sum(observation(pom, [1,1,1], [1,1,1], o) for o in get_observations
# Testing non-trivial observation.
print(sum(observation(pom, [1, 0, 0], [1, 1, 1], o) for o in get_observat
```

```
1.0000000000000002
1.0
```

In [144]:
```
# Q6
def initbelief(pom):
    b = {}
    p_qual = pom.p_qual

    for state in get_states(pom):
        zeros = 3 - sum(state)
        ones = sum(state)
        b[state] = ((1 - p_qual) ** zeros) * (p_qual ** ones)

    return b

beliefs = initbelief(pom)
print(beliefs) # Showing numerical values
print(len(beliefs))
```

```
{(0, 0, 0): 0.0009999999999999994, (0, 0, 1): 0.008999999999999996,
(0, 1, 0): 0.008999999999999996, (0, 1, 1): 0.08099999999999999, (1,
0, 0): 0.008999999999999996, (1, 0, 1): 0.08099999999999999, (1, 1,
0): 0.08099999999999999, (1, 1, 1): 0.7290000000000001}
8
```

In [145]:
```
# Q7
def update_belief(pom, b, action, observ):
    new_belief = {}
    for state in get_states(pom):
        new_prob = observation(pom, action, state, observ) * b[state]
        new_belief[state] = new_prob

    total = sum(new_belief.values())
    for state in get_states(pom):
        new_belief[state] = (new_belief[state] * (1/total))

    return new_belief

print(update_belief(pom, beliefs, [1,1,1], [1,1,1]))
# Showing that [1,1,0] increases in belief.
print(update_belief(pom, beliefs, [1,1,1], [1,1,0]))
```

```
{(0, 0, 0): 8.499859752314077e-06, (0, 0, 1): 0.00040799326811107576,
(0, 1, 0): 0.00040799326811107576, (0, 1, 1): 0.019583676869331643,
(1, 0, 0): 0.00040799326811107576, (1, 0, 1): 0.019583676869331643,
(1, 1, 0): 0.019583676869331646, (1, 1, 1): 0.9400164897279194}
{(0, 0, 0): 0.00013359213535240805, (0, 0, 1): 0.0002829009925109818,
(0, 1, 0): 0.006412422496915588, (0, 1, 1): 0.013579247640527132, (1,
0, 0): 0.006412422496915588, (1, 0, 1): 0.013579247640527132, (1, 1,
0): 0.30779627985194835, (1, 1, 1): 0.6518038867453027}
```

In [146]:
```
# Q8
for action in get_actions(pom):
    expected_reward = 0
    for belief in beliefs:
        expected_reward += (reward(pom, belief, action, 1) / 8)
    print("Action: {0} Expected Reward: {1}".format(action, expected_rewa
```

```
Action: (0, 0, 0) Expected Reward: -0.875
Action: (0, 0, 1) Expected Reward: 0.0
Action: (0, 1, 0) Expected Reward: 0.0
Action: (0, 1, 1) Expected Reward: 0.0
Action: (1, 0, 0) Expected Reward: 0.0
Action: (1, 0, 1) Expected Reward: 0.0
Action: (1, 1, 0) Expected Reward: 0.0
Action: (1, 1, 1) Expected Reward: 0.0
```

In [147]:

```python
# Q9
def lastreward(pom, state, observ):
    found_false_pass = False
    actual_fail = False
    for i in range(len(observ)):
        # If we fail them, but they were qualified
        if observ[i] == 0 and state[i] == 1:
            return pom.r_false_fail
        # If we pass them, but they weren't qualified
        elif observ[i] == 1 and state[i] == 0:
            found_false_pass = True
        elif observ[i] == 0 and state[i] == 0:
            actual_fail = True

    # If they passed something they shouldn't have
    # and didn't fail another subject
    if found_false_pass == True and actual_fail == False:
        return pom.r_false_pass

    return 0

print(lastreward(pom, [1,1,1], [1,1,-1]))
```

```
0
```

In [148]:

```python
#Q10
def expected_utility(pom, state, observ):

    # If they failed an exam, we are going to have them re-take all
    # and try again.
    if 0 in observ:
        action = [1,1,1]
        utility = 0

        for next_state in get_states(pom):
            # Odds we transition to the next state multiplied by
            # observation probability
            prob = transition(pom, state, action, next_state) * \
                observation(pom, action, next_state, observ)
            utility += prob * lastreward(pom, next_state, observ)

        return utility

    # If they didn't fail an exam, there is a deterministic evaluation
    # that follows. i.e. Either their state is [1,1,1] and we return
    # 0 or it isn't and we return -1.
    else:
        return lastreward(pom, state, observ)


print(expected_utility(pom, [1,1,1], [1,1,0]))
# They aren't qualified, but they magically passed everything
print(expected_utility(pom, [0,0,0], [1,1,1]))
# Hail mary, study and try to pass everything
print(expected_utility(pom, [0,0,0], [0,0,0]))
```

```
-0.18050000000000005
-1
-0.0360395
```

In [149]:
```python
#Q11
def compute_alpha(state, action, t, val):

    if t == 3:
        return lastreward(pom, state, action)

    r = reward(pom, state, action, t)

    for next_state in get_states(pom):

        # If they failed an exam, make them take everything again.
        if 0 in next_state:
            action = [1,1,1]
        else:
            action = [0,0,0]

        t_prob = transition(pom, state, action, next_state)

        for observ in get_observations(pom):
            val += t_prob*observation(pom, action, next_state, observ) \
                * compute_alpha(next_state, action, t + 1, val)

    return val

alpha = [compute_alpha(state, [1,1,1], 1, 0) for \
         state in get_states(pom)]
print(alpha)

# Getting initial belief, then pulling it out of the map to dot product
# with our alpha vector.
initial_b = initbelief(pom)
expected_utility_vector = []
i = 0
states = get_states(pom)
for state in get_states(pom):
    new_utility = initial_b[state] * alpha[i]
    expected_utility_vector.append(new_utility)
    i += 1
print(expected_utility_vector)
```

```
[-1.8350423836494987, -7.264989126417368, -7.264989126417369, -46.1825
1633007208, -7.2706779360190605, -46.46311917971479, -46.463119179714
8, -1699.1588239345579]
[-0.0018350423836494974, -0.06538490213775629, -0.06538490213775629, -
3.740783822735838, -0.06543610142417151, -3.7635126535568975, -3.76351
26535568983, -1238.6867826482928]
```

In [150]:
```python
#Q12
def compute_alpha2(state, action, t, val):

    if t == 3:
        return lastreward(pom, state, action)

    r = reward(pom, state, action, t)

    for next_state in get_states(pom):

        # Get new action based on policy of:
        # "Retake the exams they fail at each timestep"
        action = []
        for i in range(len(state)):
            if state[i] == 0:
                action.append(1)
            else:
                action.append(0)

        t_prob = transition(pom, state, action, next_state)

        for observ in get_observations(pom):
            val += t_prob*observation(pom, action, next_state, observ) * \
                compute_alpha2(next_state, action, t + 1, val)

    return val

alpha2 = [compute_alpha2(state, [1,1,1], 1, 0) for \
          state in get_states(pom)]
print(alpha2)

# Getting initial belief, then pulling it out of the map to dot product
# with our alpha vector.
initial_b = initbelief(pom)
expected_utility_vector2 = []
i = 0
states = get_states(pom)
for state in get_states(pom):
    new_utility = initial_b[state] * alpha2[i]
    expected_utility_vector2.append(new_utility)
    i += 1
print(expected_utility_vector2)
```

```
[-1.5379721925327552, -6.42190958625785, -6.421909586257849, -57.38326
36842079, -6.437116576458768, -58.15067294770366, -58.15067294770365,
-1.6695939778125723]
[-0.0015379721925327541, -0.05779718627632062, -0.05779718627632061, -
4.648044358420839, -0.05793404918812889, -4.710204508763996, -4.710204
5087639945, -1.2171340098253653]
```

In [ ]: