In [259]:
```
import math as _math
import time as _time

import numpy as _np
import scipy.sparse as _sp

# import mdptoolbox.util as _util
import numpy as np
import random
```

In [260]:
```
def _computeDimensions(transition):
    A = len(transition)
    try:
        if transition.ndim == 3:
            S = transition.shape[1]
        else:
            S = transition[0].shape[0]
    except AttributeError:
        S = transition[0].shape[0]
    return S, A
```

In [261]:
```
"""1(a) Create (in code) your state space S = {s}"""
def stateSpace_2D(L, H):
    # Initially all squares are legal and hence populated with 1's

    stateSpace = [[0,0] for s in range(H*L)]
    for r in range(H):
        for c in range(L):
            stateSpace[r*L + c] = [r, c]
    return stateSpace

def grid_2D(L, H):
    space = [[0]*L for h in range(H)]
    return space

L, H = 5, 6
stateSpace = stateSpace_2D(5, 6)
grid = grid_2D(5, 6)
```

In [262]:
```
grid[5][2] = 0
grid[3][1] = "-"
grid[3][2] = "-"
grid[1][1] = "-"
grid[1][2] = "-"
grid[2][2] = 1
grid[0][2] = 10
for h in range(H):
    grid[h][4] = -100
# stateSpace
grid
```

Out[262]:
```
[[0, 0, 10, 0, -100],
 [0, '-', '-', 0, -100],
 [0, 0, 1, 0, -100],
 [0, '-', '-', 0, -100],
 [0, 0, 0, 0, -100],
 [0, 0, 0, 0, -100]]
```

In each time step, the robot can either attempt to take one step in either of the four cardinal directions: {Up, Down, Left, Right}, or choose to Not move. If the robot does choose to move, it may instead experience an error with probability p_e, and actually take one step in any of the four cardinal directions with equal probability.

Note that sometimes the result of the error is the same as the originally chosen motion. If the robot chooses not to move, it will not experience any error. If a motion would result in moving off of the grid or into an obstacle (whether commanded or as a result of an error), the robot will instead stay where it is.

In [263]:
```
"""1(b). Create (in code) your action space A = {a}"""
# Dictionary of actions that will change index of position in state-space m
# actionSpace = {"U": [0, 1], "D": [0, -1], "L": [-1, 0], "R": [1, 0], "N":
actionSpace = {"U": [1, 0], "D": [-1, 0], "L": [0, -1], "R": [0, 1], "N": [
# actionSpace = {"U": [-1, 0], "D": [1, 0], "L": [0, -1], "R": [0, 1], "N":
```

In [264]:
```
"""1(c). Write a function that returns the probability psa(s') given inputs
(and global constant error probability P_e). Assume for now that there are
# Global constant ERROR PROBABILITY
P_e = 0.01

def probability_sas(state, action, next_state):
    if is_legal(state, action, next_state) == True:
        if action == "N":
            return 1
        else:
            return 1 - P_e
    else: return 0

"""
2(a). Update the state transition function from 1(c) to incorporate the dis
"""
H, L = 6, 5
def is_legal(state, action, next_state):
    y, x = next_state
    new_state = move_deterministic(state, action)
    # can't reach that cell with this move
    if new_state != next_state:
        return False
    # will fall off the grid or step into OBSTACLE

    if (y < 0 or x < 0) or (y >= H or x >= L):
        return False

    else:
        if (grid[y][x] == "-"):
            return False
    return True
```

In [265]:
```
def move(state, action):
    # state = [row, col]
    # returns index position in the state space grid
    a = actionSpace[action]
    if action == "N":
        new_state = state
    # IF choose to move
    if action != "N":
        z = random.uniform(0, 1)
        if z <= P_e:
            prob_different_action = random.uniform(0, 1)
            pda = prob_different_action
            if pda <= 0.25:
                new_state = move_deterministic(state, "U")
            elif pda <= 0.5:
                new_state = move_deterministic(state, "D")
            elif pda <= 0.75:
                new_state = move_deterministic(state, "L")
            elif pda > 0.75:
                new_state = move_deterministic(state, "R")
        elif z > P_e:
            new_state = [state[0]+a[0], state[1]+a[1]]

    # If a motion would result in moving into an obstacle (whether commande
    # or moving off of the grid
    if is_legal(state, action, new_state) == False:
        # the robot will instead stay where it is.
        new_state = state
    return new_state

def move_deterministic(state, action):
    if type(action) == str:
        action = actionSpace[action]
    y, x = action
    new_state = [state[0]+y, state[1]+x]
    return new_state
```

In [266]:
```
"""
2(b). A function that returns the reward r(s) given input s.
"""
def reward(state):
    # state = [row, col] = index position in the state space grid
    y, x = state
    return grid[y][x]
```

In [267]:
```
"""
2.3 Policy iteration
Assume an initial policy π0 of always taking a step to the Left. In this se
"""
discount = 0.9
P_e = 0.01
```

In [268]:

```
"""
3(a). Create and populate a matrix/array that contains the actions {a = π0(
policy π0 when indexed by state s.
"""
policy0 = grid_2D(5, 6)
for r in range(H):
    for c in range(L):
        policy0[r][c] = "L"

actions = grid_2D(5, 6)
for s in stateSpace:
    r, c = s
    actions[r][c] = policy0[r][c]


def blanket_policy(L, H, action_string):
    blanket_p = grid_2D(5, 6)
    for r in range(H):
        for c in range(L):
            blanket_p[r][c] = action_string
    return blanket_p

actions
```

Out[268]:

```
[['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L']]
```

In [269]:

```
policy0
```

Out[269]:

```
[['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L']]
```

In [270]:

```
"""
3(b). Write a function to display any input policy π, and use it to display
"""
def display_policy(policy):
    display = grid_2D(L, H)
    for s in stateSpace:
        r, c = s
        display[r][c] = "from "+ str((r, c)) + " go " + str(actions[r][c])
    return display
display_policy(policy0)
```

Out[270]:

```
[['from (0, 0) go L',
  'from (0, 1) go L',
  'from (0, 2) go L',
  'from (0, 3) go L',
  'from (0, 4) go L'],
 ['from (1, 0) go L',
  'from (1, 1) go L',
  'from (1, 2) go L',
  'from (1, 3) go L',
  'from (1, 4) go L'],
 ['from (2, 0) go L',
  'from (2, 1) go L',
  'from (2, 2) go L',
  'from (2, 3) go L',
  'from (2, 4) go L'],
 ['from (3, 0) go L',
  'from (3, 1) go L',
  'from (3, 2) go L',
  'from (3, 3) go L',
```

In [271]:
```
H, L = 6, 5

def fill_transitions(stateSpace, actionSpace):

    transition_probabilities = [[[0]*(H*L) for a in range(len(actionSpace))
    states_tuples = [[0,0] for s in range(H*L)]
    action_commands = ["U", "D", "L", "R", "N"]
    for r in range(H):
        for c in range(L):
            states_tuples[r*L + c] = [r, c]
    for s_i in range(len(states_tuples)):
        for a_i in range(len(action_commands)):
            for new_s_i in range(len(states_tuples)):
                state = states_tuples[s_i]
                command = action_commands[a_i]
                action = actionSpace[command]
                next_state = states_tuples[new_s_i]

                transition_probabilities[s_i][a_i][new_s_i] = probability_s

    return transition_probabilities

transition_probabilities = fill_transitions(stateSpace, actionSpace)

def fill_rewards(stateSpace):
    return grid
```

In [272]:
```
grid_2D(L, H)
```

Out[272]:
```
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0]]
```

In [273]:
```
"""
3(c). Write a function to compute the policy evaluation of a policy π.
That is, this function should return the
matrix/array of values {v = V_π(s)}, V_π(s) ∈ R when indexed by state s.
The input will be a matrix/array storing π as above (and will use global co
"""
H, L = 6, 5
max_iter = 300
# stopping criterion
epsilon = 0.00001
def policy_evaluation(policy, vocal=True):
    # set the initial values to zero
    value_scores = grid_2D(L, H)
    prev_value_scores = grid_2D(L, H)
    num_iters, gain = 0, epsilon
    while num_iters <= max_iter and gain >= epsilon:
        gain = 0
        for state in stateSpace:

            reward_state = reward(state)
            if reward_state == "-":
                value_scores[r][c] = 0.0
                continue
                # print("skip obstacle")
            else:
                r, c = state
                action = policy[r][c]
                a_i = 0
                action_commands = ["U", "D", "L", "R", "N"]
                while action_commands[a_i] != action:
                    a_i += 1
                expected_sum_of_rewards = 0
                for possible_next_state_i in range(len(states_tuples)):
                    trans_prob = transition_probabilities[5*r+c][a_i][possi
                    next_state = states_tuples[possible_next_state_i] #move
                    y_next, x_next = next_state
                    expected_sum_of_rewards += trans_prob*value_scores[y_ne
                # print("Reward(state): ", reward(state), " (discount**num_
                value_scores[r][c] = reward(state) + (discount**num_iters)*

        for r in range(H):
            for c in range(L):
                gain += abs(value_scores[r][c] - prev_value_scores[r][c])
        if vocal == True:
            print("ITER # ", num_iters, " Gain: ", gain)
            print(" Value_scores: ", value_scores)
        num_iters += 1
        for r in range(H):
            for c in range(L):
                prev_value_scores[r][c] = value_scores[r][c]
    return value_scores
```

In [274]:
```
states_tuples = [[0,0] for s in range(30)]
for r in range(H):
    for c in range(L):
        states_tuples[r*L + c] = [r, c]
```

In [113]:
```
p_1 = policy_evaluation(policy0)
```

```
('ITER # ', 0, ' Gain: ', 611.1089)
(' Value_scores: ', [[0.0, 0.0, 10.0, 9.9, -90.199], [0.0, 0, 0, 0.0, -10
0.0], [0.0, 0.0, 1.0, 0.99, -99.0199], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.
0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0]])
('ITER # ', 1, ' Gain: ', 3.137408999999993)
(' Value_scores: ', [[0.0, 0.0, 10.0, 8.91, -92.06119], [0.0, 0, 0, 0.0,
-100.0], [0.0, 0.0, 1.0, 0.891, -99.206119], [0.0, 0, 0, 0.0, -100.0],
[0.0, 0.0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0]])
('ITER # ', 2, ' Gain: ', 2.639311290000002)
(' Value_scores: ', [[0.0, 0.0, 10.0, 8.019, -93.5695639], [0.0, 0, 0, 0.
0, -100.0], [0.0, 0.0, 1.0, 0.8019000000000001, -99.35695639], [0.0, 0,
0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.
0]])
('ITER # ', 3, ' Gain: ', 2.2260511448999916)
(' Value_scores: ', [[0.0, 0.0, 10.0, 7.217100000000001, -94.791346758999
99], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.0, 1.0, 0.7217100000000001, -99.47
91346759], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0], [0.0,
0.0, 0.0, 0.0, -100.0]])
('ITER # ', 4, ' Gain: ', 1.8824895273690192)
```

In [114]:
```
p_1
```

Out[114]:
```
[[0.0, 0.0, 10.0, 7.426706340347218e-05, -99.99999999944843],
 [0.0, 0, 0, 0.0, -100.0],
 [0.0, 0.0, 1.0, 7.4267063403472175e-06, -99.99999999994485],
 [0.0, 0, 0, 0.0, -100.0],
 [0.0, 0.0, 0.0, 0.0, -100.0],
 [0.0, 0.0, 0.0, 0.0, -100.0]]
```

In [275]:
```
"""
3(d). Write a function that returns a matrix/array π
giving the optimal policy under a one-step lookahead (Bellman backup)
when given an input value function V.

Display the policy that results from a one-step improvement on π0.
"""
V = policy_evaluation
# BELLMAN BACKUP
def optimal_policy(V):
    #  takes a value function V as input
    #  returns a new value function after a Bellman backup
    policy = grid_2D(L, H)
    for r in range(H):
            for c in range(L):
                policy[r][c] = policy0[r][c]

    for s_i in range(len(states_tuples)):
        s = states_tuples[s_i]
        r, c = s

        policy = best_action(policy, transition_probabilities, stateSpace[s

    return policy


def best_action(policy, transition_probabilities, state):
    action_commands = ["U", "D", "L", "R", "N"]
    best_action_i = 0
    best_expected_reward = 0

    for a_i in range(len(action_commands)):
        expected_sum_of_rewards = 0
        for possible_next_state_i in range(len(states_tuples)):
            trans_prob = transition_probabilities[5*r+c][a_i][possible_next
            action = action_commands[a_i]
#             print(action)
            next_state = move(state, action)#states_tuples[possible_next_st
            y_next, x_next = next_state
            value_scores = V(policy0)
            expected_sum_of_rewards += trans_prob*value_scores[y_next][x_ne

        if expected_sum_of_rewards > best_expected_reward:
            best_expected_reward = expected_sum_of_rewards
            best_action_i = action_commands[a_i]
            policy[r][c] = best_action_i
    return policy
```

In [152]:

```
"""
3(d). Display the policy that results from a one-step improvement on π0.
"""
policy1 = optimal_policy(V)
```

```
('ITER # ', 0, ' Gain: ', 611.1089)
(' Value_scores: ', [[0.0, 0.0, 10.0, 9.9, -90.199], [0.0, 0, 0, 0.0, -10
0.0], [0.0, 0.0, 1.0, 0.99, -99.0199], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.
0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0]])
('ITER # ', 1, ' Gain: ', 21.56112188999999)
(' Value_scores: ', [[0.0, 0.0, 10.0, 0.099, -99.9990199], [0.0, 0, 0, 0.
0, -100.0], [0.0, 0.0, 1.0, 0.0099, -99.99990199], [0.0, 0, 0, 0.0, -100.
0], [0.0, 0.0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0]])
('ITER # ', 2, ' Gain: ', 0.10888900218900423)
(' Value_scores: ', [[0.0, 0.0, 10.0, 0.00099, -99.99999990199], [0.0, 0,
0, 0.0, -100.0], [0.0, 0.0, 1.0, 9.900000000000001e-05, -99.99999999019
9], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.
0, 0.0, -100.0]])
('ITER # ', 3, ' Gain: ', 0.0010782178002185295)
(' Value_scores: ', [[0.0, 0.0, 10.0, 9.900000000000002e-06, -99.99999999
99902], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.0, 1.0, 9.9e-07, -99.9999999999
9902], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0], [0.0, 0.0,
0.0, 0.0, -100.0]])
('ITER # ', 4, ' Gain: ', 1.078111078603873e-05)
```

In [21]:

```
"""
3(d). Display the policy that results from a one-step improvement on π0.
"""
policy1
```

Out[21]:

```
[['L', 'R', 'N', 'L', 'L'],
 ['L', 'L', 'D', 'D', 'L'],
 ['L', 'R', 'N', 'L', 'L'],
 ['L', 'L', 'D', 'D', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L']]
```

In [276]:
```
"""
3(d). Write a function that returns a matrix/array π
giving the optimal policy under a one-step lookahead (Bellman backup)
when given an input value function V.

Display the policy that results from a one-step improvement on π0.
"""
def copy_grid(policy, initial_policy):
    for r in range(H):
            for c in range(L):
                policy[r][c] = initial_policy[r][c]
    return policy

V = policy_evaluation
# BELLMAN BACKUP
def optimal_policy(V, initial_policy=policy0):
    #  takes a value function V as input
    #  returns a new value function after a Bellman backup
    policy = actions #grid_2D(H, L)
    policy = copy_grid(initial_policy, policy)
    print(policy)

    old_values = policy_evaluation(policy)

    action_commands = ["U", "D", "L", "R", "N"]
    for a_i in range(5):
        temp_policy = blanket_policy(L, H, action_commands[a_i])
        values_a_i = policy_evaluation(temp_policy)
        for r in range(6):
            for c in range(5):
                if values_a_i[r][c] > old_values[r][c]:
                    policy[r][c] = temp_policy[r][c]
    return policy
```

In [41]:

```
"""
3(d). Display the policy that results from a one-step improvement on π0.
"""
policy1 = optimal_policy(policy_evaluation)
```

```
[['L', 'L', 'L', 'L', 'L'], ['L', 'L', 'L', 'L', 'L'], ['L', 'L', 'L',
'L', 'L'], ['L', 'L', 'L', 'L', 'L'], ['L', 'L', 'L', 'L', 'L'], ['L',
'L', 'L', 'L', 'L']]
('ITER # ', 0, ' Gain: ', 611.1089)
(' Value_scores: ', [[0.0, 0.0, 10.0, 9.9, -90.199], [0.0, 0, 0, 0.0, -10
0.0], [0.0, 0.0, 1.0, 0.99, -99.0199], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.
0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0]])
('ITER # ', 1, ' Gain: ', 3.137408999999993)
(' Value_scores: ', [[0.0, 0.0, 10.0, 8.91, -92.06119], [0.0, 0, 0, 0.0,
-100.0], [0.0, 0.0, 1.0, 0.891, -99.206119], [0.0, 0, 0, 0.0, -100.0],
[0.0, 0.0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0]])
('ITER # ', 2, ' Gain: ', 2.639311290000002)
(' Value_scores: ', [[0.0, 0.0, 10.0, 8.019, -93.5695639], [0.0, 0, 0, 0.
0, -100.0], [0.0, 0.0, 1.0, 0.8019000000000001, -99.35695639], [0.0, 0,
0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.
0]])
('ITER # ', 3, ' Gain: ', 2.2260511448999916)
(' Value_scores: ', [[0.0, 0.0, 10.0, 7.217100000000001, -94.791346758999
99], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.0, 1.0, 0.7217100000000001, -99.47
```

In [66]:

```
# policy1 = optimal_policy(V)
policy1
```

Out[66]:

```
[['R', 'R', 'N', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['R', 'R', 'N', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L']]
```

In [277]:
```
"""
3(e). Combine your functions above to create a new function that computes p
returning optimal policy π* with optimal value V*.
Display π*
"""
discount = 0.01
def no_policy_change(old_p, new_p):
    for s_i in range(H*L):
        r, c = stateSpace[s_i]
        if old_p[r][c] != new_p[r][c]:
            return False
    return True

def policy_iteration(p_init=blanket_policy(L, H, "D")):

    p_updated = optimal_policy(policy_evaluation)
    print("P_updated: ", p_updated)

    if no_policy_change(p_init, p_updated) == True:
        return (p_updated, policy_evaluation(p_updated))
    else:
        return policy_iteration(p_updated)
```

In [75]:
```
p_optimal, v_optimal = policy_iteration(blanket_policy(L, H, "D"))
```

```
[['L', 'L', 'L', 'L', 'L'], ['L', 'L', 'L', 'L', 'L'], ['L', 'L', 'L',
'L', 'L'], ['L', 'L', 'L', 'L', 'L'], ['L', 'L', 'L', 'L', 'L'], ['L',
'L', 'L', 'L', 'L']]
('ITER # ', 0, ' Gain: ', 611.1089)
(' Value_scores: ', [[0.0, 0.0, 10.0, 9.9, -90.199], [0.0, 0, 0, 0.0, -10
0.0], [0.0, 0.0, 1.0, 0.99, -99.0199], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.
0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0]])
('ITER # ', 1, ' Gain: ', 21.56112188999999)
(' Value_scores: ', [[0.0, 0.0, 10.0, 0.099, -99.9990199], [0.0, 0, 0, 0.
0, -100.0], [0.0, 0.0, 1.0, 0.0099, -99.99990199], [0.0, 0, 0, 0.0, -100.
0], [0.0, 0.0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0]])
('ITER # ', 2, ' Gain: ', 0.10888900218900423)
(' Value_scores: ', [[0.0, 0.0, 10.0, 0.00099, -99.99999990199], [0.0, 0,
0, 0.0, -100.0], [0.0, 0.0, 1.0, 9.900000000000001e-05, -99.99999999019
9], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.
0, 0.0, -100.0]])
('ITER # ', 3, ' Gain: ', 0.0010782178002185295)
(' Value_scores: ', [[0.0, 0.0, 10.0, 9.900000000000002e-06, -99.99999999
99902], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.0, 1.0, 9.9e-07, -99.9999999999
```

In [117]:
```
"""3(e). Display π*"""
p_optimal
```

Out[117]:
```
[['R', 'R', 'N', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['R', 'R', 'N', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L'],
 ['L', 'L', 'L', 'L', 'L']]
```

In [120]:
```
v_optimal
```

Out[120]:
```
[[9.801009703960107e-18,
  9.9000000980101e-10,
  10.00000000099,
  9.900000000980102e-10,
  -100.0],
 [0.0, 0, 0, 0.0, -100.0],
 [9.801009703960108e-19,
  9.900000098010097e-11,
  1.000000000099,
  9.900000000980101e-11,
  -100.0],
 [0.0, 0, 0, 0.0, -100.0],
 [0.0, 0.0, 0.0, 0.0, -100.0],
 [0.0, 0.0, 0.0, 0.0, -100.0]]
```

It looks like because the discount factor is so large, when the starting square is is at 5th row it is not worth it for the algorithm to try to move to the pellet.

But if we start close enough to the reward the values propagate nicely.

I should take about $O(S^2 * A)$

In [154]:
```
"""
3(f). How much compute time did it take to generate your optimal policy in
You may want to use your programming language’s built-in runtime analysis t
"""
import cProfile

cProfile.run(policy_iteration(policy0), False)
```

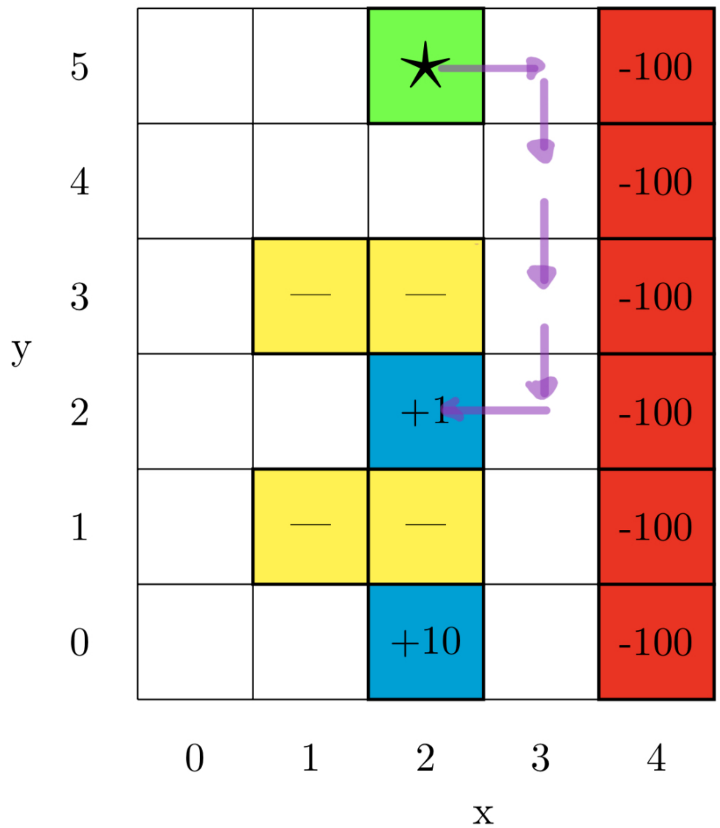
```
('ITER # ', 0, ' Gain: ', 611.1089)
(' Value_scores: ', [[0.0, 0.0, 10.0, 9.9, -90.199], [0.0, 0, 0, 0.0, -10
0.0], [0.0, 0.0, 1.0, 0.99, -99.0199], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.
0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0]])
('ITER # ', 1, ' Gain: ', 21.56112188999999)
(' Value_scores: ', [[0.0, 0.0, 10.0, 0.099, -99.9990199], [0.0, 0, 0, 0.
0, -100.0], [0.0, 0.0, 1.0, 0.0099, -99.99990199], [0.0, 0, 0, 0.0, -100.
0], [0.0, 0.0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0]])
('ITER # ', 2, ' Gain: ', 0.10888900218900423)
(' Value_scores: ', [[0.0, 0.0, 10.0, 0.00099, -99.99999990199], [0.0, 0,
0, 0.0, -100.0], [0.0, 0.0, 1.0, 9.900000000000001e-05, -99.99999999019
9], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0], [0.0, 0.0, 0.
0, 0.0, -100.0]])
('ITER # ', 3, ' Gain: ', 0.0010782178002185295)
(' Value_scores: ', [[0.0, 0.0, 10.0, 9.900000000000002e-06, -99.99999999
99902], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.0, 1.0, 9.9e-07, -99.9999999999
9902], [0.0, 0, 0, 0.0, -100.0], [0.0, 0.0, 0.0, 0.0, -100.0], [0.0, 0.0,
0.0, 0.0, -100.0]])
('ITER # ', 4, ' Gain: ', 1.078111078603873e-05)
```

In [148]:
```
"""3(g). Starting in the initial state as drawn above, plot a trajectory un
What is the total discounted reward of that trajectory?
What is the expected discounted sum-of-rewards for a robot starting in that
num_steps = 5
rew = +1
total_discounted_reward = discount**num_steps*rew
total_discounted_reward = 0.9**5*1
total_discounted_reward
```

Out[148]:
```
0.5904900000000001
```

In [151]:
```
from IPython.display import Image
Image(filename="3(g)OptimalPath.jpeg", width=400, height=300)
# ![title](img/Q3(g)OptimalPath.jpeg)
```

Out[151]:



In [319]:
```
"""
2.4 Value iteration
4(a). Using an initial condition V (s) = 0 ∀s ∈ S, write a function (and an
from the functions above) to compute value iteration, again returning optim
* with optimal value V*
"""
V_vec = grid_2D(L, H)
transition_probabilities = fill_transitions(stateSpace, actionSpace)
actions = grid_2D(L, H)

"""V = not policy_evaluation function, but an estimator matrix"""
def one_step_lookahead(s, V_vec):

    num_states = len(transition_probabilities)
    action_commands = ["U", "D", "L", "R", "N"]
    for r in range(6):
        for c in range(5):
            for state_y in range(6):
                for state_x in range(5):
                    max_rew = -999
                    for action in range(len(actionSpace)):
                        state = [state_y, state_x]
                        s_i = 5*state_y+state_x
                        act_str = action_commands[action]
                        next_state = move(state, act_str)
                        n_y, n_x = next_state
                        n_i = 5*n_y+n_x
                        rew = reward([state_y, state_x])
                        if rew == "-":
                            rew = 0
                        if rew > max_rew:
                            max_rew = rew
                            actions[state_y][state_x] = action_commands[act

                        expected_values[r][c] += transition_probabilities[s
    return expected_values, actions

def value_iteration(epsilon=0.0001, discount=0.9):
    expected_values = grid_2D(L, H)
    expected_values = copy_grid(expected_values, v_optimal)
    old_expected_values = grid_2D(L, H)
    old_expected_values = copy_grid(old_expected_values, expected_values)
    while True:
        # stopping condition
        gain = 0
        # Update in each state
        for y in range(6):
            for x in range(5):
                state = [y, x]
                # Do a one-step lookahead to find the best action
                old_expected_values = copy_grid(old_expected_values, expect
                expected_values, best_acts = one_step_lookahead(state, V_ve

                max_act = 0
                best_action_value = 0
                for r in range(6):
```

```
                    for c in range(5):
                        if expected_values[r][c] > max_act:
                            best_action_value = expected_values[r][c]
                            policy[r][c] = best_acts[r][c]
                # Calculate delta across all states seen so far
#                 delta = max(delta, np.abs(best_action_value - V_vec[y][x]
                for r in range(H):
                    for c in range(L):
                        gain += abs(old_expected_values[r][c] - expected_va
                # Update the value function
                V_vec[y][x] = best_action_value
        # Check if we can stop
        if gain < epsilon:
            print("gain < epsilon", gain, epsilon)
            break



    # Create a deterministic policy using the optimal value function
    policy = blanket_policy(L, H, "D")
    for nr in range(6):
        for nc in range(5):
            # One step lookahead to find the best action for this state
            expected_values, best_acts = one_step_lookahead([nr, nc], V_vec
            best_reward = 0
            for r in range(6):
                for c in range(5):
                    if expected_values[nr][nc] > best_reward:
                        policy[nr][nc] = best_acts[nr][nc]

    return policy, V_vec
```

In [ ]:

```
"""
4(a). Using an initial condition V (s) = 0 ∀s ∈ S, write a function (and ar
from the functions above) to compute value iteration, again returning optim
* with optimal value V
"""
opt_policy, opt_vals = value_iteration(0.0001, 0.9)
```

In [ ]:

```
"""
4(b). Run this function to recompute and display the optimal policy π
*
. Also generate a trajectory for a robot and
compute its reward as described in 3(g). Compare these results with those y
opt_policy
```

In [ ]:

```
Image(filename="3(g)OptimalPath.jpeg", width=400, height=300)
```

As expected, the optimal path is the same for both Policy and Value Iteration algorithms.

In [286]:
```
"""
4(c). How much compute time did it take to generate your results from 4(b)?
3(f).
"""
cProfile.run(value_iteration(0.0001, 0.9))
```

Out[286]:
```
[[9.801009703960107e-18,
  9.9000000980101e-10,
  10.00000000099,
  9.900000000980102e-10,
  -100.0],
 [0.0, 0, 0, 0.0, -100.0],
 [9.801009703960108e-19,
  9.900000098010097e-11,
  1.000000000099,
  9.900000000980101e-11,
  -100.0],
 [0.0, 0, 0, 0.0, -100.0],
 [0.0, 0.0, 0.0, 0.0, -100.0],
 [0.0, 0.0, 0.0, 0.0, -100.0]]
```

```
"""2.5 Additional scenarios
5(a). Explore different values of γ, pe to find different optimal
policies, and characterize / explain your observations."""
```

We find that small discount factor (i.e. bigger decay per step rate of reward) makes algorithm prioritize shorter paths, while discount of 1 is the same as no discount and the algorithm will wonder infinitely on an infinite board in that scenario. Bigger error rate makes algorithm prioritize staying away from negative rewards and taking longer path to minimize the chance of accidentally taking a step into negative reward cell. While error rate closer to 0 will make algorithm take shorter paths even if they are right next to large negative rewards. Small error rate with small discount factor makes algorithm choose a path next to fire pits (negative reward cells) to a small reward cell, because by the time it would get to a bigger reward, that bigger reward will decay and be smaller than the close-by small reward.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: