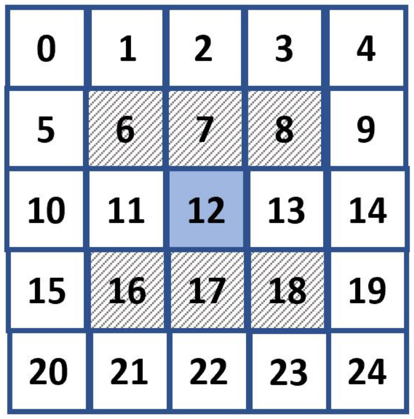
Homework 6

CSCI E-82A

In the lesson we constructed a representation of a simple grid world. Dynamic programming (DP) was used to find optimal plans for a robot to navigate from any starting location on the grid to the goal. This problem is an analog for more complex real-world robot navigation problems.

In this homework you will use DP to solve a slightly more complex robotic navigation problem in a grid world. This grid world is a simple version of the problem a material transport robot might encounter in a warehouse. The situation is illustrated in the figure below.



Grid World for Factory Navigation Example

Reference:

- [Gridworld - Stanford (https://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_dp.html)]
- [Grid - Python (https://www.hackerearth.com/fr/practice/notes/dynamic-programming-problems-involving-grids/)]
- [Python Optimization (https://medium.com/@annishared/searching-for-optimal-policies-in-python-an-intro-to-optimization-7182d6fe4dba)]

The goal is for the robot to deliver some material to position (state) 12, shown in blue. Since there is a goal state or **terminal state** this an **episodic task**.

There are some barriers comprised of the states $\{6, 7, 8\}$ and $\{16, 17, 18\}$, shown with hash marks. In a real warehouse, these positions might be occupied by shelving or equipment. We do not want the robot to hit these barriers. Thus, we say that transitioning to these barrier states is **taboo**.

As before, we do not want the robot to hit the edges of the grid world, which represent the outer walls of the warehouse.

Representation

As with many such problems, the starting place is creating the **representation**. In the cell below encode your representation for the possible action-state transitions. From each state there are 4 possible actions:

- up, u
- down, d,
- left, l
- right, r

There are a few special cases you need to consider:

- Any action transitioning state off the grid or into a barrier should keep the state unchanged.
- Any action in the goal state keeps the state unchanged.
- Any transition within the taboo (barrier) states can keep the state unchanged. If you experiment, you will see that other encodings work as well since the value of a barrier states are always zero and there are no actions transitioning into these states.

> **Hint:** It may help you create a pencil and paper sketch of the transitions, rewards, and probabilities or policy. This can help you to keep the bookkeeping correct.

Entrée [1]:
```
import numpy as np

policy = {0:{'u':0, 'd':5, 'l':0, 'r':1},
          1:{'u':1, 'd':1, 'l':0, 'r':2},
          2:{'u':2, 'd':2, 'l':1, 'r':3},
          3:{'u':3, 'd':3, 'l':2, 'r':4},
          4:{'u':4, 'd':9, 'l':3, 'r':4},
          5:{'u':0, 'd':10, 'l':5, 'r':5},
          6:{'u':6, 'd':6, 'l':6, 'r':6},
          7:{'u':7, 'd':7, 'l':7, 'r':7},
          8:{'u':8, 'd':8, 'l':8, 'r':8},
          9:{'u':4, 'd':14, 'l':9, 'r':9},
          10:{'u':5, 'd':15, 'l':10, 'r':11},
          11:{'u':11, 'd':11, 'l':10, 'r':12},
          12:{'u':12, 'd':12, 'l':12, 'r':12},
          13:{'u':13, 'd':13, 'l':12, 'r':14},
          14:{'u':9, 'd':19, 'l':13, 'r':14},
          15:{'u':10, 'd':20, 'l':15, 'r':15},
          16:{'u':16, 'd':16, 'l':16, 'r':16},
          17:{'u':17, 'd':17, 'l':17, 'r':17},
          18:{'u':18, 'd':18, 'l':18, 'r':18},
          19:{'u':14, 'd':24, 'l':19, 'r':19},
          20:{'u':15, 'd':20, 'l':20, 'r':21},
          21:{'u':21, 'd':21, 'l':20, 'r':22},
          22:{'u':22, 'd':22, 'l':21, 'r':23},
          23:{'u':23, 'd':23, 'l':22, 'r':24},
          24:{'u':19, 'd':24, 'l':23, 'r':24}}
```

You need to define the initial transition probabilities for the Markov process. Set the probabilities for each transition as a **uniform distribution** leading to random action by the robot.

> **Note:** As these are just starting values, the exact values of the transition probabilities are not actually all that important in terms of solving the DP problem. Also, notice that it does not matter how the taboo state transitions are encoded. The point of the DP algorithm is to learn the transition policy.

Entrée [2]:
```
state_to_state_probs = {0:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      1:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      2:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      3:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      4:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      5:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      6:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      7:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      8:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      9:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      10:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      11:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      12:{'u':0.0, 'd':0.0, 'l':0.0, 'r':0.0},
                      13:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      14:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      15:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      16:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      17:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      18:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      19:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      20:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      21:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      22:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      23:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25},
                      24:{'u':0.25, 'd':0.25, 'l': 0.25, 'r':0.25}}
```

The robot receives the following rewards:

- +10 for achieving the goal.
- -1 for attempting to leave the warehouse or hitting the barriers. In other words, we penalize the robot for hitting the edges of the grid or the barriers.
- -0.1 for all other state transitions, which is the cost for the robot to move from one state to another.

In the code cell below encode your representation of this reward structure.

Entrée [3]:
```
rewards =  {0:{'u':-1, 'd':-0.1, 'l':-1, 'r':-0.1},
           1:{'u':-1, 'd':-1, 'l':-0.1, 'r':-0.1},
           2:{'u':-1, 'd':-1, 'l':-0.1, 'r':-0.1},
           3:{'u':-1, 'd':-1, 'l':-0.1, 'r':-0.1},
           4:{'u':-1, 'd':-0.1, 'l':-0.1, 'r':-1},
           5:{'u':-0.1, 'd':-0.1, 'l':-1, 'r':-1},
           6:{'u':-1, 'd':-1, 'l':-1, 'r':-1},
           7:{'u':-1, 'd':-1, 'l':-1, 'r':-1},
           8:{'u':-1, 'd':-1, 'l':-1, 'r':-1},
           9:{'u':-0.1, 'd':-0.1, 'l':-1, 'r':-1},
           10:{'u':-0.1, 'd':-0.1, 'l':-1, 'r':-0.1},
           11:{'u':-1, 'd':-1, 'l':-0.1, 'r':10},
           12:{'u':0.0, 'd':0.0, 'l':0.0, 'r':0.0},
           13:{'u':-1, 'd':-1, 'l':10, 'r':-0.1},
           14:{'u':-0.1, 'd':-0.1, 'l':-0.1, 'r':-1},
           15:{'u':-0.1, 'd':-0.1, 'l':-1, 'r':-1},
           16:{'u':-1, 'd':-1, 'l':-1, 'r':-1},
           17:{'u':-1, 'd':-1, 'l':-1, 'r':-1},
           18:{'u':-0.1, 'd':-0.1, 'l':-0.1, 'r':-0.1},
           19:{'u':-0.1, 'd':-0.1, 'l':-1, 'r':-1},
           20:{'u':-0.1, 'd':-1, 'l':-1, 'r':-0.1},
           21:{'u':-1, 'd':-1, 'l':-0.1, 'r':-0.1},
           22:{'u':-1, 'd':-1, 'l':-0.1, 'r':-0.1},
           23:{'u':-1, 'd':-1, 'l':-0.1, 'r':-0.1},
           24:{'u':-0.1, 'd':-1, 'l':-0.1, 'r':-1}}
```

You will find it useful to create a list of taboo states, which you can encode in the cell below.

Entrée [4]:
```
taboos = [6, 7, 8, 16, 17, 18]
```

Policy Evaluation

With your representation defined, you can now create and test a function for **policy evaluation**. You will need this function for your policy iteration code.

You are welcome to state with the `compute_state_value` function from the DP notebook. However, keep in mind that you must modify this code to correctly treat the taboo states. Specifically, taboo states should have 0 value.

Entrée [5]:
```
def compute_state_value(pi, probs, reward, taboos, gamma = 1.0, theta =
    '''Function for policy evaluation
    '''
    delta = theta
    values = np.zeros(len(probs)) # Initialize the value array
    while(delta >= theta):
        v = np.copy(values) ## save the values for computing the differ
        for s in probs.keys():
            temp_values = 0.0 ## Initial the sum of values for this sta
            if s not in taboos: ## Adding a condition to test for taboo
                for action in rewards[s].keys():
                    s_prime = pi[s][action]
                    temp_values = temp_values + probs[s][action] * (rew
            values[s] = temp_values

        ## Compute the difference metric to see if convergence has been
        diffs = np.sum(np.abs(np.subtract(v, values)))
        delta = min([delta, diffs])
        if(display):
            print('difference metric = ' + str(diffs))
            print(values.reshape(5,5))
    return values

compute_state_value(policy, state_to_state_probs, rewards, taboos, thet
```

Out[5]:
```
array([[-38.28926029, -41.56189427, -42.65499935, -41.56857879,
        -38.30169862],
       [-32.84169421,   0.        ,   0.        ,   0.        ,
        -32.8525447 ],
       [-25.20972897,  -8.65258033,   0.        ,  -8.65482963,
        -25.21851918],
       [-32.84716098,   0.        ,   0.        ,   0.        ,
        -32.85784594],
       [-38.3016983 , -41.5751609 , -42.66847476, -41.58164302,
        -38.31375999]])
```

Examine the state values you have computed using a random walk for the robot. Answer the following questions:

1. Are the values of the goal and taboo states zero? ANS:
2. Do the values of the states increase closer to the goal? ANS:
3. Do the goal and barrier states all have zero values? ANS:

If your answer to any of the above questions is no, you need to do some more work on your code!

1 yes 2 yes 3 yes

Policy Iteration

Now that you have your representation and a function to perform policy evaluation you have everything you need to use the policy iteration algorithm to create an optimal policy for the robot to reach the goal.

If your policy evaluation function works correctly, you should be able to use the `policy_iteration` function from the DP notebook. Make sure you print the state values as well as the policy you discovered.

Entrée [17]:
```
import copy
def policy_iteration(pi, probs, reward, taboos, gamma = 1.0, theta = 0
    delta = theta
    v = np.zeros(len(probs))
    state_values = np.zeros(len(probs))
    current_policy = copy.deepcopy(probs)
    while(delta >= theta): # Continue until convergence.
        for s in probs.keys(): # Iterate over all states
            temp_values = []
            for action in rewards[s].keys(): # Iterate over all possib
                # Compute list of values given action from the state
                s_prime = pi[s][action]
                temp_values.append(current_policy[s][action] * (reward

            ## Find the max value and update current policy
            max_index = np.where(np.array(temp_values) == max(temp_val
            prob_for_policy = 1.0/float(len(max_index))
            for i,action in enumerate(current_policy[s].keys()):
                if(i in max_index): current_policy[s][action] = prob_f
                else: current_policy[s][action] = 0.0


        ## Compute state values with new policy to determine if there
        ## Uses the compute_state_value function
        state_values = compute_state_value(pi, current_policy, rewards
        diff = np.sum(np.abs(np.subtract(v, state_values)))
        if(output):
            print('\ndiff = ' + str(diff))
            print('Current policy')
            print(current_policy)
            print('With state values')
            print(state_values.reshape(5,5))

        delta = min([delta, np.sum(np.abs(np.subtract(v, state_values)
        v = np.copy(state_values) ## copy of state values to evaluate
    return current_policy
```

```
pol_ite = policy_iteration(policy, state_to_state_probs, rewards, tabo
pol_ite
```

```
diff = 134.74001676680857
Current policy
{0: {'u': 0.0, 'd': 0.5, 'l': 0.0, 'r': 0.5}, 1: {'u': 0.0, 'd': 0.0,
'l': 0.5, 'r': 0.5}, 2: {'u': 0.0, 'd': 0.0, 'l': 0.5, 'r': 0.5}, 3: {
'u': 0.0, 'd': 0.0, 'l': 0.5, 'r': 0.5}, 4: {'u': 0.0, 'd': 0.5, 'l':
0.5, 'r': 0.0}, 5: {'u': 0.5, 'd': 0.5, 'l': 0.0, 'r': 0.0}, 6: {'u':
0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25}, 7: {'u': 0.25, 'd': 0.25, 'l':
0.25, 'r': 0.25}, 8: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25}, 9:
{'u': 0.5, 'd': 0.5, 'l': 0.0, 'r': 0.0}, 10: {'u': 0.3333333333333333
, 'd': 0.3333333333333333, 'l': 0.0, 'r': 0.3333333333333333}, 11: {'u
': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0}, 12: {'u': 0.25, 'd': 0.25, 'l':
0.25, 'r': 0.25}, 13: {'u': 0.0, 'd': 0.0, 'l': 1.0, 'r': 0.0}, 14: {'
u': 0.3333333333333333, 'd': 0.3333333333333333, 'l': 0.33333333333333
33, 'r': 0.0}, 15: {'u': 0.5, 'd': 0.5, 'l': 0.0, 'r': 0.0}, 16: {'u':
0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25}, 17: {'u': 0.25, 'd': 0.25, 'l'
: 0.25, 'r': 0.25}, 18: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
19: {'u': 0.5, 'd': 0.5, 'l': 0.0, 'r': 0.0}, 20: {'u': 0.5, 'd': 0.0,
'l': 0.0, 'r': 0.5}, 21: {'u': 0.0, 'd': 0.0, 'l': 0.5, 'r': 0.5}, 22:
{'u': 0.0, 'd': 0.0, 'l': 0.5, 'r': 0.5}, 23: {'u': 0.0, 'd': 0.0, 'l'
: 0.5, 'r': 0.5}, 24: {'u': 0.5, 'd': 0.0, 'l': 0.5, 'r': 0.0}}
With state values
[[ 7.00426586  6.69649666  6.59649666  6.70372331  7.0171673 ]
 [ 7.52540325  0.          0.          0.          7.53545571]
 [ 8.25027265 10.          0.         10.          8.25697326]
 [ 7.53062108  0.          0.          0.          7.539969  ]
 [ 7.01718399  6.71046544  6.61046544  6.7167151   7.02834205]]

diff = 40.45998323319145
Current policy
{0: {'u': 0.0, 'd': 1.0, 'l': 0.0, 'r': 0.0}, 1: {'u': 0.0, 'd': 0.0,
'l': 1.0, 'r': 0.0}, 2: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0}, 3: {
'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0}, 4: {'u': 0.0, 'd': 1.0, 'l':
0.0, 'r': 0.0}, 5: {'u': 0.0, 'd': 1.0, 'l': 0.0, 'r': 0.0}, 6: {'u':
0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25}, 7: {'u': 0.25, 'd': 0.25, 'l':
0.25, 'r': 0.25}, 8: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25}, 9:
{'u': 0.0, 'd': 1.0, 'l': 0.0, 'r': 0.0}, 10: {'u': 0.0, 'd': 0.0, 'l'
: 0.0, 'r': 1.0}, 11: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0}, 12: {'
u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25}, 13: {'u': 0.0, 'd': 0.0, '
l': 1.0, 'r': 0.0}, 14: {'u': 0.0, 'd': 0.0, 'l': 1.0, 'r': 0.0}, 15:
{'u': 1.0, 'd': 0.0, 'l': 0.0, 'r': 0.0}, 16: {'u': 0.25, 'd': 0.25, '
l': 0.25, 'r': 0.25}, 17: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25}
, 18: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25}, 19: {'u': 1.0, 'd'
: 0.0, 'l': 0.0, 'r': 0.0}, 20: {'u': 1.0, 'd': 0.0, 'l': 0.0, 'r': 0.
0}, 21: {'u': 0.0, 'd': 0.0, 'l': 1.0, 'r': 0.0}, 22: {'u': 0.0, 'd':
0.0, 'l': 0.0, 'r': 1.0}, 23: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0}
, 24: {'u': 1.0, 'd': 0.0, 'l': 0.0, 'r': 0.0}}
```

```
With state values
[[ 9.7  9.6  9.5  9.6  9.7]
 [ 9.8  0.   0.   0.   9.8]
 [ 9.9 10.   0.  10.   9.9]
 [ 9.8  0.   0.   0.   9.8]
 [ 9.7  9.6  9.5  9.6  9.7]]

diff = 0.0
Current policy
{0: {'u': 0.0, 'd': 1.0, 'l': 0.0, 'r': 0.0}, 1: {'u': 0.0, 'd': 0.0,
'l': 1.0, 'r': 0.0}, 2: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0}, 3: {
'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0}, 4: {'u': 0.0, 'd': 1.0, 'l':
0.0, 'r': 0.0}, 5: {'u': 0.0, 'd': 1.0, 'l': 0.0, 'r': 0.0}, 6: {'u':
0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25}, 7: {'u': 0.25, 'd': 0.25, 'l':
0.25, 'r': 0.25}, 8: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25}, 9:
{'u': 0.0, 'd': 1.0, 'l': 0.0, 'r': 0.0}, 10: {'u': 0.0, 'd': 0.0, 'l'
: 0.0, 'r': 1.0}, 11: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0}, 12: {'
u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25}, 13: {'u': 0.0, 'd': 0.0, '
l': 1.0, 'r': 0.0}, 14: {'u': 0.0, 'd': 0.0, 'l': 1.0, 'r': 0.0}, 15:
{'u': 1.0, 'd': 0.0, 'l': 0.0, 'r': 0.0}, 16: {'u': 0.25, 'd': 0.25, '
l': 0.25, 'r': 0.25}, 17: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25}
, 18: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25}, 19: {'u': 1.0, 'd'
: 0.0, 'l': 0.0, 'r': 0.0}, 20: {'u': 1.0, 'd': 0.0, 'l': 0.0, 'r': 0.
0}, 21: {'u': 0.0, 'd': 0.0, 'l': 1.0, 'r': 0.0}, 22: {'u': 0.0, 'd':
0.0, 'l': 0.0, 'r': 1.0}, 23: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0}
, 24: {'u': 1.0, 'd': 0.0, 'l': 0.0, 'r': 0.0}}
With state values
[[ 9.7  9.6  9.5  9.6  9.7]
 [ 9.8  0.   0.   0.   9.8]
 [ 9.9 10.   0.  10.   9.9]
 [ 9.8  0.   0.   0.   9.8]
 [ 9.7  9.6  9.5  9.6  9.7]]
```

```
Out[17]: {0: {'u': 0.0, 'd': 1.0, 'l': 0.0, 'r': 0.0},
          1: {'u': 0.0, 'd': 0.0, 'l': 1.0, 'r': 0.0},
          2: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0},
          3: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0},
          4: {'u': 0.0, 'd': 1.0, 'l': 0.0, 'r': 0.0},
          5: {'u': 0.0, 'd': 1.0, 'l': 0.0, 'r': 0.0},
          6: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
          7: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
          8: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
          9: {'u': 0.0, 'd': 1.0, 'l': 0.0, 'r': 0.0},
          10: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0},
          11: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0},
          12: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
          13: {'u': 0.0, 'd': 0.0, 'l': 1.0, 'r': 0.0},
          14: {'u': 0.0, 'd': 0.0, 'l': 1.0, 'r': 0.0},
          15: {'u': 1.0, 'd': 0.0, 'l': 0.0, 'r': 0.0},
          16: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
          17: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
          18: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
          19: {'u': 1.0, 'd': 0.0, 'l': 0.0, 'r': 0.0},
          20: {'u': 1.0, 'd': 0.0, 'l': 0.0, 'r': 0.0},
          21: {'u': 0.0, 'd': 0.0, 'l': 1.0, 'r': 0.0},
          22: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0},
          23: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0},
          24: {'u': 1.0, 'd': 0.0, 'l': 0.0, 'r': 0.0}}
```

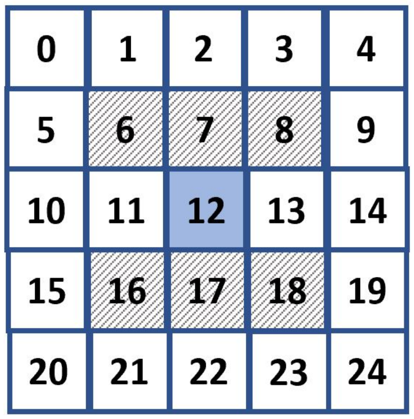
Examine your results. First look at the state values at convergence of the policy iteration algorithm and answer the following questions:

1. Are non-taboo state values closest to the goal the largest? ANS: yes
2. Are the non-taboo state values farthest from the goal the smallest? Keep in mind the robot must travel around the barrier. ANS: yes
3. Are the non-taboo state values symmetric (e.g. same) with respect to distance from the goal? ANS: Yes.
4. Do the taboo states have 0 values? ANS: Yes.

If your answer to any of the above questions is No, there is an error in your code.

Next, examine the policy you have computed. Do the follow:

1. Follow the optimal paths from the 4 corners of the grid to the goal. How does the symmetry and length of these paths make sense in terms of length and state values? ANS:
2. Imagine that the door for the warehouse is at position (state) 2. Insert an illustration showing the paths of the optimal plans below. You are welcome to start with the PowerPoint illustration in the course Github repository.



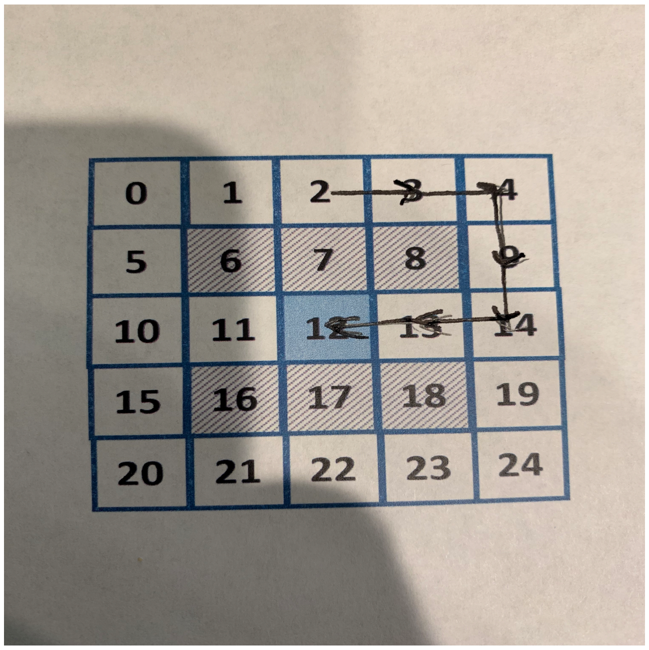
Grid world optimal plans from state 2 to the goal shown goes here

Jagriti: the above is a sample answer.

```
1  yes
2 yes
3 yes
4 yes

Policy examination:

1 yes
```



Grid World path

Value Iteration

Finally, your will use the value iteration algorithm to compute an optimal policy for the robot reaching the goal. Keep in mind that you will need to maintain a state value of 0 for the taboo states.

Entrée [8]:
```
def value_iteration(pi, probs, reward, goal, taboos, gamma = 1.0, theta
    delta = theta
    v = np.zeros(len(probs))
    state_values = np.zeros(len(probs))
    current_policy = copy.deepcopy(probs)

    while(delta >= theta):
        for s in probs.keys(): # iteratve over all states
            temp_values = []
            ## Find the values for all possible actions in the state.
            for action in rewards[s].keys():
                s_prime = pi[s][action]
                if s in taboos:
                    temp_values.append(0)
                else:
                    temp_values.append((reward[s][action] + gamma * sta
                #print('{} {}'.format(s, temp_values))

            ## Find the max value and update the value for the state
            state_values[s] = max(temp_values)
        ## Determine if convergence is achieved
        diff = np.sum(np.abs(np.subtract(v, state_values)))
        delta = min([delta, np.sum(np.abs(np.subtract(v, state_values))
        print(np.sum(np.abs(np.subtract(v, state_values))))
        v = np.copy(state_values)
        if(output):
            print('Difference = ' + str(diff))
            print(state_values.reshape(5,5))

    ## Now we need to find the policy that makes max value state transi
    for s in current_policy.keys(): # iterate over all states
        ## Find the indicies of maximum state transition values
        ## There are two cases.
        ## First, the special case of a state adjacent to the goal
        ## In this case need to ensure the only possible transition is
        ## Start by creating a list of the adjacent states.
        possible_s_prime = [pi[s][key] for key in current_policy[s].key
        ## Check if goal is adjacent, but state is not the goal.
        if(goal in possible_s_prime and s != goal):
            temp_values = []
            for s_prime in current_policy[s].keys(): # Iterate over adj
                if(pi[s][s_prime] == goal):  ## account for the special
                    temp_values.append(reward[s][s_prime])
                else: temp_values.append(0.0) ## Other transisions have
        ## The other case is rather easy requires a lookup of the value
        ## adjacent state and handled with a list comprehension.
        else: temp_values = [state_values[pi[s][s_prime]] for s_prime i

        ## Find the index for the state transistions with the largest v
        ## May be more than one.
        max_index = np.where(np.array(temp_values) == max(temp_values))
        prob_for_policy = 1.0/float(len(max_index)) ## Probabilities of
        for i, key in enumerate(current_policy[s]): ## Update policy
            if(i in max_index): current_policy[s][key] = prob_for_polic
            else: current_policy[s][key] = 0.0
    return current_policy

val_ite = value_iteration(policy, state_to_state_probs, rewards, 12, ta
val_ite
```

```
50.7
Difference = 50.7
```

```
[[-0.1 -0.1 -0.1 -0.1 -0.1]
 [-0.1  0.   0.   0.  -0.1]
 [-0.1 10.   0.  10.   9.9]
 [-0.1  0.   0.   0.   9.8]
 [-0.1 -0.1 -0.1 -0.1  9.7]]
69.2
Difference = 69.2
[[-0.2 -0.2 -0.2 -0.2 -0.2]
 [-0.2  0.   0.   0.   9.8]
 [ 9.9 10.   0.  10.   9.9]
 [ 9.8  0.   0.   0.   9.8]
 [ 9.7  9.6  9.5  9.6  9.7]]
20.299999999999997
Difference = 20.299999999999997
[[-0.3 -0.3 -0.3 -0.3  9.7]
 [ 9.8  0.   0.   0.   9.8]
 [ 9.9 10.   0.  10.   9.9]
 [ 9.8  0.   0.   0.   9.8]
 [ 9.7  9.6  9.5  9.6  9.7]]
39.60000000000001
Difference = 39.60000000000001
[[ 9.7  9.6  9.5  9.6  9.7]
 [ 9.8  0.   0.   0.   9.8]
 [ 9.9 10.   0.  10.   9.9]
 [ 9.8  0.   0.   0.   9.8]
 [ 9.7  9.6  9.5  9.6  9.7]]
0.0
Difference = 0.0
[[ 9.7  9.6  9.5  9.6  9.7]
 [ 9.8  0.   0.   0.   9.8]
 [ 9.9 10.   0.  10.   9.9]
 [ 9.8  0.   0.   0.   9.8]
 [ 9.7  9.6  9.5  9.6  9.7]]
```

```
Out[8]: {0: {'u': 0.0, 'd': 1.0, 'l': 0.0, 'r': 0.0},
         1: {'u': 0.0, 'd': 0.0, 'l': 1.0, 'r': 0.0},
         2: {'u': 0.0, 'd': 0.0, 'l': 0.5, 'r': 0.5},
         3: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0},
         4: {'u': 0.0, 'd': 1.0, 'l': 0.0, 'r': 0.0},
         5: {'u': 0.0, 'd': 1.0, 'l': 0.0, 'r': 0.0},
         6: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
         7: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
         8: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
         9: {'u': 0.0, 'd': 1.0, 'l': 0.0, 'r': 0.0},
         10: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0},
         11: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0},
         12: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
         13: {'u': 0.0, 'd': 0.0, 'l': 1.0, 'r': 0.0},
         14: {'u': 0.0, 'd': 0.0, 'l': 1.0, 'r': 0.0},
         15: {'u': 1.0, 'd': 0.0, 'l': 0.0, 'r': 0.0},
         16: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
         17: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
         18: {'u': 0.25, 'd': 0.25, 'l': 0.25, 'r': 0.25},
         19: {'u': 1.0, 'd': 0.0, 'l': 0.0, 'r': 0.0},
         20: {'u': 1.0, 'd': 0.0, 'l': 0.0, 'r': 0.0},
         21: {'u': 0.0, 'd': 0.0, 'l': 1.0, 'r': 0.0},
         22: {'u': 0.0, 'd': 0.0, 'l': 0.5, 'r': 0.5},
         23: {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0},
         24: {'u': 1.0, 'd': 0.0, 'l': 0.0, 'r': 0.0}}
```

Compare your results from the value iteration algorithm to your results from the policy iteration algorithm and answer the following questions:

1. Are the state values identical between the two methods? ANS:
2. Ignoring the taboo states, are the plans computed by the two methods identical? ANS:

Entrée [16]:
```python
for (key_p, value_p), (key_v, value_v) in zip(pol_ite.items(), val_ite
    if not value_p == value_v:
        print(('state {} \n Policy iteration {} \n Value iteration  {}
```

```
state 2
 Policy iteration {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0}
 Value iteration  {'u': 0.0, 'd': 0.0, 'l': 0.5, 'r': 0.5}

state 22
 Policy iteration {'u': 0.0, 'd': 0.0, 'l': 0.0, 'r': 1.0}
 Value iteration  {'u': 0.0, 'd': 0.0, 'l': 0.5, 'r': 0.5}
```

1 yes

2 the plan are different for state 2 and state, value_v does not preferd on specific direction

Entrée [ ]: