In [1]:

```python
import numpy as np
import matplotlib.pyplot as plt
import time
```

Q1: Implement the discrete MDP model.

In [64]:

```python
P=np.zeros((3,2,3))
R=np.zeros((3,2))
P[0,0,0]=0.45
P[2,0,0]=0.55
P[2,1,0]=1
P[2,0,1]=1
P[0,1,1]=0.5
P[1,1,1]=0.4
P[2,1,1]=0.1
P[2,0,1]=1
P[0,0,2]=0.6
P[2,0,2]=0.4
P[1,1,2]=0.9
P[2,1,2]=0.1
R[0,0]=-0.4
R[0,1]=0
R[1,0]=2
R[1,1]=0
R[2,0]=-1
R[2,1]=-0.5
```

P is the transition 3x2x3 matrix of the given MDP model, the first index indicates the next state, the second one indicates the action taken and the third one is for the previous (or the current) state.

R is the reward 3x2 matrix, its rows represent the states and its columns the actions.

Q2: Implement and run value iteration (VI) in order to identify a 0.01–optimal policy.

In [351]:

```python
def VI(P,R,discount_factor,precision):
    v_new=np.zeros((1,np.shape(P)[0]))
    v_old=v_new+1
    iteration=0
    while(abs(np.max(v_old-v_new))>precision):
        v_old=v_new
        V=np.zeros((np.shape(P)[0],np.shape(P)[1]))
        for x in range(np.shape(P)[0]):
            for a in range(np.shape(P)[1]):
                V[x,a]=R[x,a]+discount_factor*np.sum(P[:,a,x]*v_new)
        v_new=np.max(V,axis=1)
        iteration+=1
    p=np.argmax(V,axis=1)
    return np.array([p,v_new,iteration])
```

The function VI (value iteration) takes in argument the transition and reward matrices, The discount_factor (gamma) and the precision of the optimal policy. It returns an array of three elements, the first one is the optimal policy, the second one the optimal value and the third one is the number of iteration till the while condition is unsatisfied anymore.

In [94]:

```python
print('The 0.01-optimal policy is : ', VI(P,R,0.95,0.01)[0], '\nThe 0.01-optimal value is:', VI
(P,R,0.95,0.01)[1])
```

```
The 0.01-optimal policy is :  [ 1.  0.  1.]
The 0.01-optimal value is: [ 12.20945133  14.20945133  12.86174503]
```

Plot ‖vk − v∗‖∞ as a function of iteration k, knowing that d∗ = [a1, a0, a1] (i.e., implement policy evaluation).

In [354]:

```python
pi_optimal=[1,0,1]
P_pi=np.array([P[:,int(pi_optimal[0]),0],P[:,int(pi_optimal[1]),1],P[:,int(pi_optimal[2]),2]])
r_pi=np.array([R[0,int(pi_optimal[0])],R[1,int(pi_optimal[1])],R[2,int(pi_optimal[2])]])
v_optimal=np.dot(np.linalg.inv(np.eye(np.shape(P)[0])-0.95*P_pi),r_pi)
```

Calculating the optimal value given the optimal policy, using the exact formula (matrix formula: Vπ =(I−γPπ)−1rπ).

In [355]:

```python
def policy_evaluation(k,P,R,discount_factor,v_optimal):
    v_new=np.zeros((1,np.shape(P)[0]))
    for i in range(k):
        V=np.zeros((np.shape(P)[0],np.shape(P)[1]))
        for x in range(np.shape(P)[0]):
            for a in range(np.shape(P)[1]):
                V[x,a]=R[x,a]+discount_factor*np.sum(P[:,a,x]*v_new)
        v_new=np.max(V,axis=1)
    return abs(np.max(v_optimal-v_new))
```

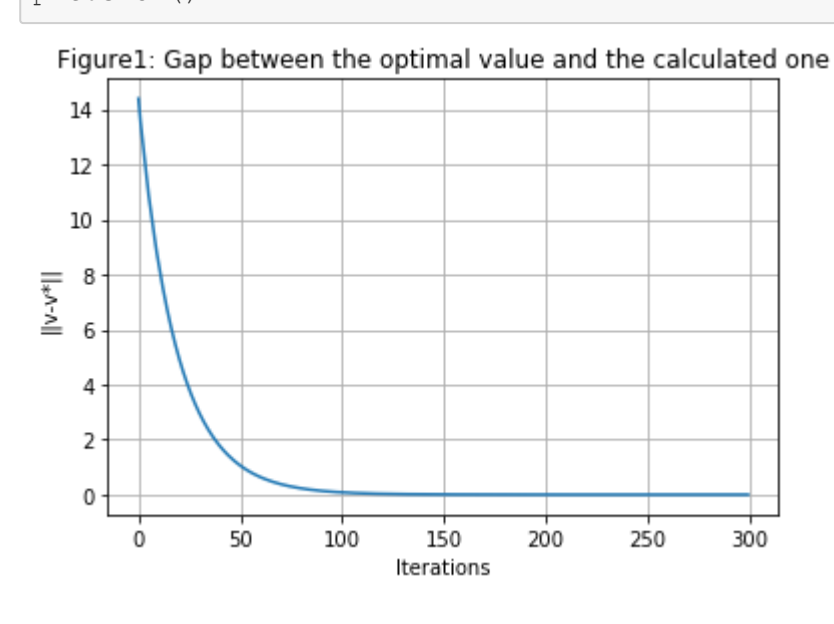
The function policy_evaluation takes in argument The number of iterations k, the transition and reward matrices, The discount_factor (gamma) and the the optimal value. It returns the maximum difference between the elementwise of the optimal value and the estimated one.

In [356]:

```python
k=[i for i in range(300)]
value_convergence=[policy_evaluation(i,P,R,0.95,v_optimal) for i in range(300)]
```

In [357]:

```python
plt.plot(k,value_convergence)
plt.xlabel('Iterations')
plt.ylabel('||v-v*||')
plt.grid()
plt.title('Figure1: Gap between the optimal value and the calculated one')
plt.show()
```



The figure tells us that the gap between the optimal value and the calculated one decreases when the number of iteration increases till it converges to 0 around 110 iterations.

Q3: implement policy iteration (PI) with arbitrary initial policy.

In [98]:

```python
def PI(P,R,discount_factor):
    p=np.zeros(np.shape(P)[0])
    v_old=np.zeros(len(P))
    is_changed=True
    iteration=0
    while(is_changed):
        P_pi=np.array([P[:,int(p[0]),0],P[:,int(p[1]),1],P[:,int(p[2]),2]])
        r_pi=np.array([R[0,int(p[0])],R[1,int(p[1])],R[2,int(p[2])]])
        v_pi_new=np.dot(np.linalg.inv(np.eye(np.shape(P)[0])-discount_factor*P_pi),r_pi)
        if((v_pi_new==v_old).all()):
            is_changed=False
        v_old=v_pi_new
        V=np.zeros((np.shape(P)[0],np.shape(P)[1]))
        for x in range(np.shape(P)[0]):
            for a in range(np.shape(P)[1]):
                V[x,a]=R[x,a]+discount_factor*np.sum(P[:,a,x]*v_pi_new)
        p=np.argmax(V,axis=1)
        iteration+=1
    return [p,v_pi_new,iteration]
```

The function PI (Policy iteration) takes in argument The number of iterations k, the transition and reward matrices, The discount_factor (gamma). It returns an array of three elements, the first one is the optimal policy, the second one the optimal value and the third one is the number of iteration till the while condition is unsatisfied anymore. For the policy evaluation step I used the matrix closed form, since the given MDP is not too complex (matrix formula: Vπ =(I−γPπ)−1rπ).

Compare the speed of convergence w.r.t. VI and discuss the relative merits of the two approaches.

In [108]:

```python
t=time.time()
value_iteration=VI(P,R,0.95,0)
elapsed=time.time()-t
```

In [109]:

```python
t=time.time()
policy_iteration=PI(P,R,0.95)
elapsed_2=time.time()-t
```

In [110]:

```python
print('The time taken by the value iteration algorithm to converge is', elapsed)
print('The number of value iterations till convergence is: ', value_iteration[2])
print('The time taken by the policy iteration algorithm to converge is', elapsed_2)
print('The number of policy iterations till convergence is: ', policy_iteration[2])
```

```
The time taken by the value iteration algorithm to converge is 0.03574800491333008
The number of value iterations till convergence is:  642
The time taken by the policy iteration algorithm to converge is 0.0011980533599853516
The number of policy iterations till convergence is:  3
```

The time taken to converge, in this case, of the VI approach is larger than the one taken by PI approach, but this is because the VI is an approximatly method and then if we are searching for exact convergences we should wait a lot of iterations unlike the PI approach which use the closed form of policy evaluation and then will converge rapidly, but if the dimension of the problem is higher (the number of action and states is very large) then the computation of the policy evaluation using the closed form will be computationnaly very expensive and then the PI will take much more time than the VI approach.

The relative merits of the approches:
-The pros of VI are: There isn't a lot of computation in each iteration (computationnally efficient).
-The cons of VI are: The convergence is asymptotic and then the algorithm may need a lot of iterations to give a good approximation of the optimal value.
-The pros of PI are: The PI approach converges to the exact optimal value unlike the VI approach in a finite number of iterations (generally not very large).
-The cons of PI are: The PI approach is computationnally expensive since it needs to do at each iteration a complete policy evaluation which need a lot of calculation, specially when the matrix closed form is used.

Q4: denote with Qn(x,a) the value function estimated using Monte-Carlo, i.e., empirical average. Build such estimator and plot Jn − J π as a function of n

In [2]:

```python
from gridworld import GridWorld1
import gridrender as gui
import numpy as np
import time

env = GridWorld1

################################################################################
# investigate the structure of the environment
# - env.n_states: the number of states
# - env.state2coord: converts state number to coordinates (row, col)
# - env.coord2state: converts coordinates (row, col) into state number
# - env.action_names: converts action number [0,3] into a named action
# - env.state_actions: for each state stores the action availables
#   For example
#       print(env.state_actions[4]) -> [1,3]
#       print(env.action_names[env.state_actions[4]]) -> ['down' 'up']
# - env.gamma: discount factor
################################################################################
print(env.state2coord)
print(env.coord2state)
print(env.state_actions[4])
print(env.action_names[env.state_actions[4]])

################################################################################
# Policy definition
# If you want to represent deterministic action you can just use the number of
# the action. Recall that in the terminal states only action 0 (right) is
# defined.
# In this case, you can use gui.renderpol to visualize the policy
################################################################################
pol = [1, 2, 0, 0, 1, 1, 0, 0, 0, 0, 3]
gui.render_policy(env, pol)

################################################################################
# Try to simulate a trajectory
# you can use env.step(s,a, render=True) to visualize the transition
################################################################################

env.render = True
state = 0
fps = 1
for i in range(5):
        action = np.random.choice(env.state_actions[state])
        nexts, reward, term = env.step(state,action)
        state = nexts
        time.sleep(1./fps)

################################################################################
# Work to do: Q4 :See in the next cell
################################################################################


# here the v-function and q-function to be used for question 4
v_q4 = [0.87691855, 0.92820033, 0.98817903, 0.00000000, 0.67106071, -0.99447514, 0.00000000, -
0.82847001, -0.87691855,
        -0.93358351, -0.99447514]


q_q4 = [[0.87691855, 0.65706417],
        [0.92820033, 0.84364237],
        [0.98817903, -0.75639924, 0.89361129],
        [0.00000000],
        [-0.62503460, 0.67106071],
        [-0.99447514, -0.70433689, 0.75620264],
        [0.00000000],
        [-0.82847001, 0.49505225],
        [-0.87691855, -0.79703229],
        [-0.93358351, -0.84424050, -0.93896668],
        [-0.89268904, -0.99447514]
        ]



################################################################################
# Work to do: Q5 see in the next cells
################################################################################
v_opt = [0.87691855, 0.92820033, 0.98817903, 0.00000000, 0.82369294, 0.92820033, 0.00000000, 0.
77818504, 0.82369294,
         0.87691855, 0.82847001]
```

```
[[0, 0], [0, 1], [0, 2], [0, 3], [1, 0], [1, 2], [1, 3], [2, 0], [2, 1], [2, 2], [2, 3]]
[[ 0  1  2  3]
 [ 4 -1  5  6]
 [ 7  8  9 10]]
[1, 3]
['down' 'up']
```

In [66]:

```python
def value_estimation(n,Tmax,gamma):
    Q=np.zeros((env.n_states,4))
    N=np.zeros((env.n_states,4),dtype=int)
    for i in range(n):
        state=env.reset()
        if(0 in env.state_actions[state]):
            action=0
        else:
            action=3
        x_0,action_0=state,action
        N[x_0,action_0]+=1
        t=0
        S=0
        term=False
        while(t<Tmax and term==False):
            nexts, reward, term = env.step(state,action)
            state = nexts
            S+=gamma**(t-1)*reward
            if(0 in env.state_actions[state]):
                action=0
            else:
                action=3
            t+=1
        Q[x_0,action_0]+=S
    return np.divide(Q,N,out=np.zeros_like(Q), where=N!=0)
```

The function value estimation takes in argument The number of iterations n, Tmax the maximal length of a trajectory and the discount_factor (gamma). It returns an array Q which is a 11x4 matrix and it has as rows the states and as columns the actions. The algo uses a deterministic action which is go right if it's possible, up otherwise.

In [3]:

```python
# here we build the function that calculate abs(Jn-J) at each iteration n
v_q4 = [0.87691855, 0.92820033, 0.98817903, 0.00000000, 0.67106071, -0.99447514, 0.00000000, -
0.82847001, -0.87691855,
        -0.93358351, -0.99447514]
def plt_estimation(n,Tmax,gamma,v):
    J=0
    Jn=0
    for state in range(11):
        J+=(1/11)*v[state]
        if(0 in env.state_actions[state]):
                action=0
        else:
                action=3
        Jn+=(1/11)*value_estimation(n,Tmax,gamma)[state,action]
    return abs(Jn-J)
```

The function plt_estimation takes four arguments, the same as value_estimation function and the fourth one the optimal value v, and it return the abs of gap between Jn and J.

In [21]:

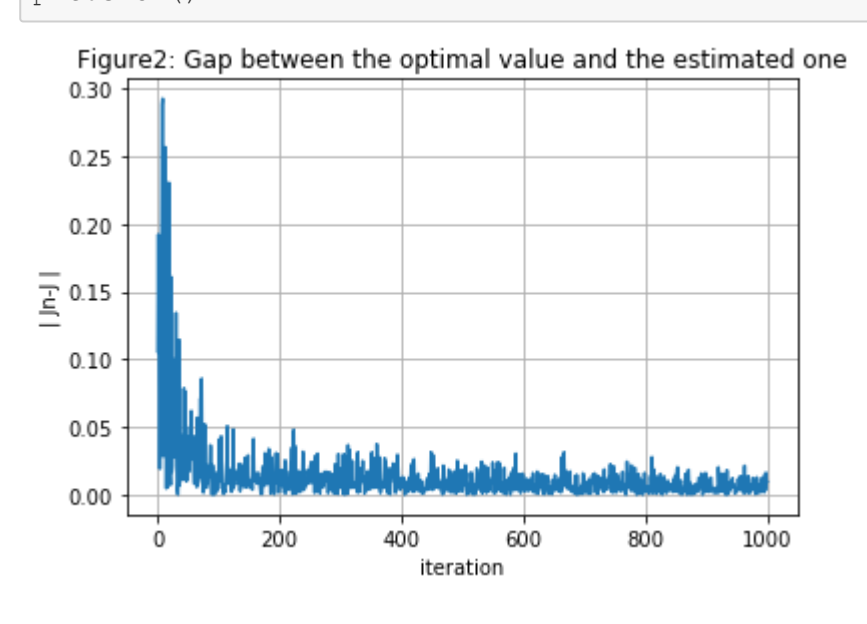
```python
print('The gap after 10 000 iteration is:', plt_estimation(10000,5,0.95,v_q4))
```

```
The gap after 10 000 iteration is: 0.00188132693416
```

In [358]:

```python
# here we plot abs(Jn-J) as a function of iteration n
echantillon=[i for i in range(1000)]
estimation=[plt_estimation(i,5,0.95,v_q4) for i in range(1000)]

plt.plot(echantillon,estimation)
plt.grid()
plt.xlabel('iteration')
plt.ylabel('| Jn-J |')
plt.title('Figure2: Gap between the optimal value and the estimated one')
plt.show()
```



We could see that the gap decreases as the number of iteration increase with some fluctuations, the plot show the gap for a range of iteration from 0 to 1000. The time taken to plot the figure is not negligible duo to the expensive calculation in the function value_estimation as the number n becomes very large. But we could see that the gap is almost equal to 0 after the 10 000 th iteration (0.001).

Q5: Describe the (parameters of the) exploration policy and learning rate chosen.

In [267]:

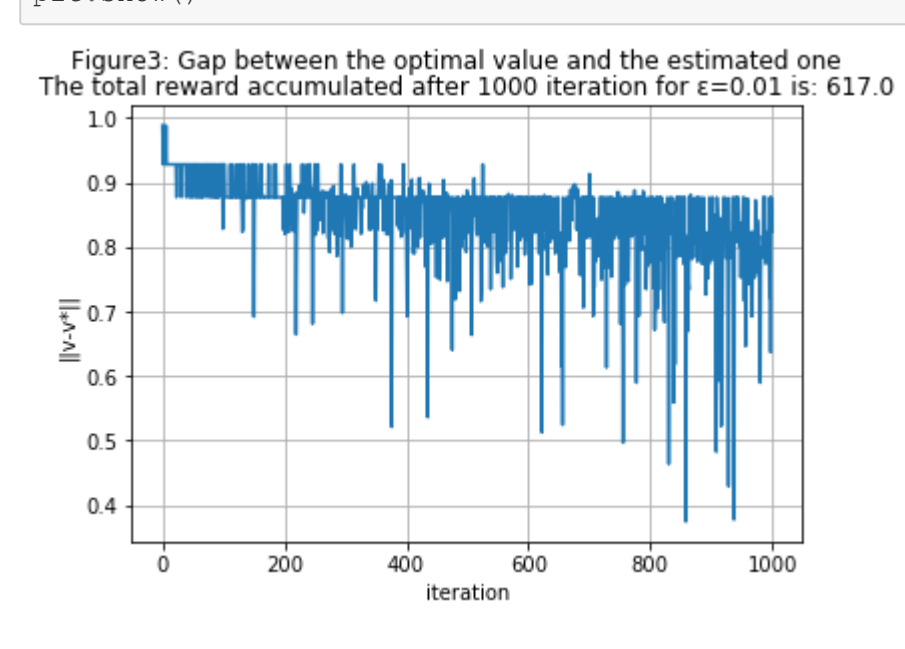
```python
def q_learning(n,Tmax,gamma,epsilon):
    Q=np.zeros((11,4))
    N=np.zeros((env.n_states,4),dtype=int)
    total_reward=0
    for i in range(n):
        state=env.reset()
        random=np.random.rand()
        if (random<=epsilon):
            action = np.random.choice(env.state_actions[state])
        else:
            z=Q[state,:][[env.state_actions[state]]]
            action=env.state_actions[state][np.argmax(z)]
        N[state,action]+=1
        t=0
        term=False
        while(t<Tmax and term==False):
            nexts, reward, term = env.step(state,action)
            total_reward+=reward
            alpha=1/N[state,action]
            q=Q[nexts,:][[env.state_actions[nexts]]]
            Q[state,action]=(1-alpha)*Q[state,action]+alpha*(reward+gamma*np.max(q))
            state=nexts
            if (random<=epsilon):
                action = np.random.choice(env.state_actions[state])
            else:
                z=Q[state,:][[env.state_actions[state]]]
                action=env.state_actions[state][np.argmax(z)]
            N[state,action]+=1
            t+=1
    return [Q,total_reward]
```

The function q_learning takes as arguments number of iteration n, the maximal length of the trajectory Tmax, gamma the discount factor and epsilon the exploration probability. It returns the 11x4 matrix of the Q values of each state*action and the total_reward accumulated over all the trajectories. The stepsizes is chosen to be harmonical (i.e α_N(xt,at)(state,action)=1/N[state,action] where N is the matrix given the number of times the (state,action) pairs are visited over all trajectories till the time t of the last trajectory. The exploration policy is chosen as parameter (argument) and will be chosen small enough to let all the state to be explorated.

illustrate the convergence of Q-Learning

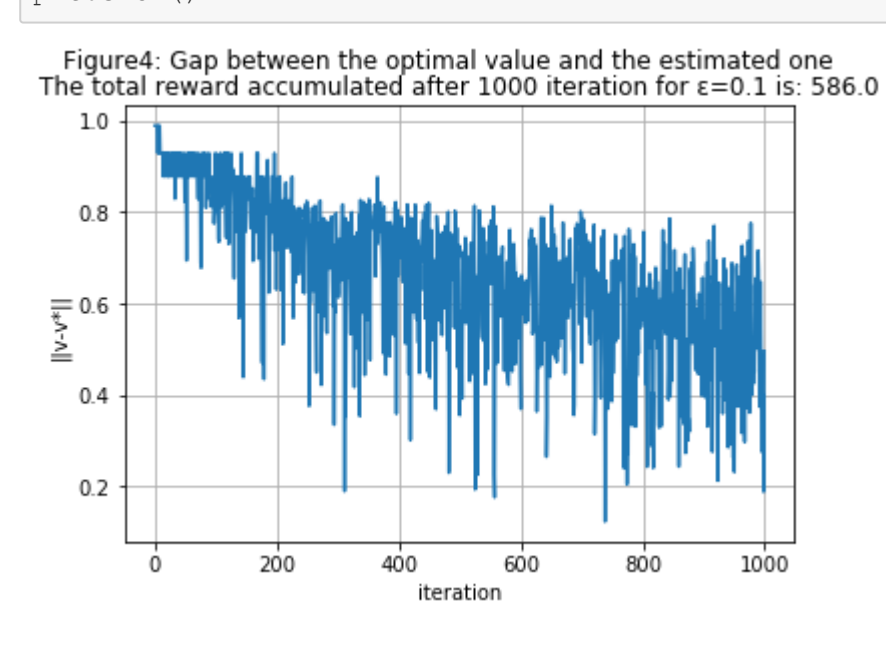
In [407]:

```python
v_optimal =[0.877,0.928,0.988,0,0.824,0.928,0,0.778,0.824,0.877,0.828]
iteration=[i for i in range(1000)]
gap=[np.max(abs(v_optimal-np.max(q_learning(i,5,0.95,0.01)[0],axis=1))) for i in range(1000)]
tot_reward=str(q_learning(1000,5,0.95,0.01)[1])
plt.plot(iteration,gap)
plt.xlabel('iteration')
plt.ylabel('||v-v*||')
plt.grid()
plt.suptitle('Figure3: Gap between the optimal value and the estimated one')
plt.title('The total reward accumulated after 1000 iteration for ε=0.01 is: ' + tot_reward)
plt.show()
```



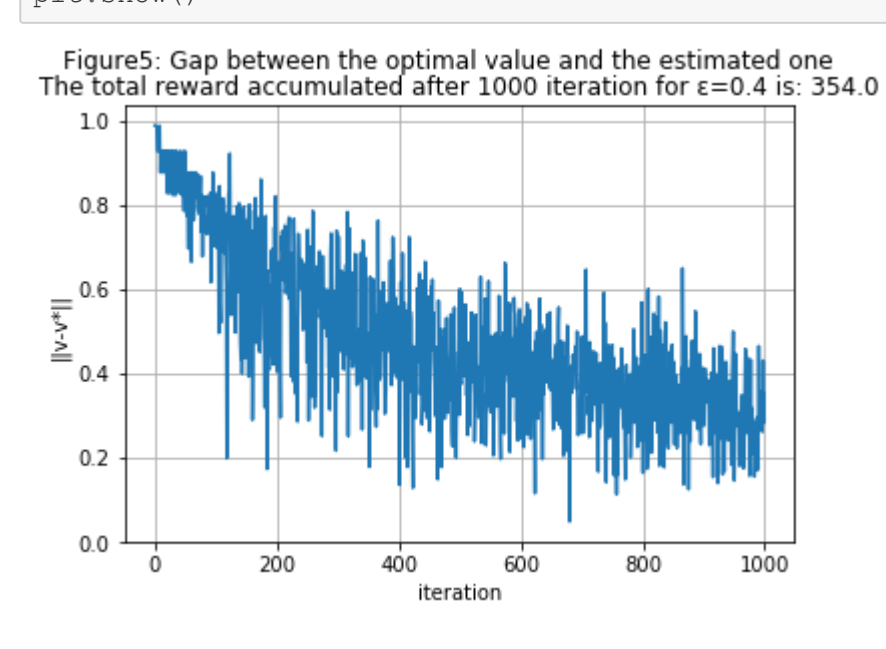
In [361]:

```python
v_optimal =[0.877,0.928,0.988,0,0.824,0.928,0,0.778,0.824,0.877,0.828]
iteration=[i for i in range(1000)]
gap=[np.max(abs(v_optimal-np.max(q_learning(i,5,0.95,0.1)[0],axis=1))) for i in range(1000)]
tot_reward=str(q_learning(1000,5,0.95,0.1)[1])
plt.plot(iteration,gap)
plt.xlabel('iteration')
plt.ylabel('||v-v*||')
plt.grid()
plt.suptitle('Figure4: Gap between the optimal value and the estimated one')
plt.title('The total reward accumulated after 1000 iteration for ε=0.1 is: ' + tot_reward)
plt.show()
```



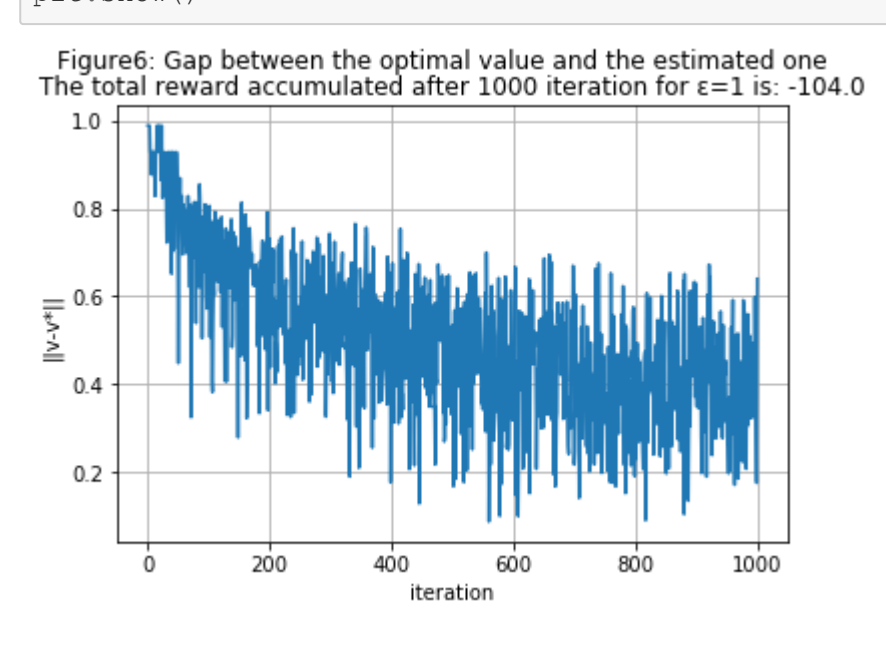
In [362]:

```python
v_optimal =[0.877,0.928,0.988,0,0.824,0.928,0,0.778,0.824,0.877,0.828]
iteration=[i for i in range(1000)]
gap=[np.max(abs(v_optimal-np.max(q_learning(i,5,0.95,0.4)[0],axis=1))) for i in range(1000)]
plt.plot(iteration,gap)
tot_reward=str(q_learning(1000,5,0.95,0.4)[1])
plt.xlabel('iteration')
plt.ylabel('||v-v*||')
plt.grid()
plt.title('The total reward accumulated after 1000 iteration for ε=0.4 is: ' + tot_reward)
plt.suptitle('Figure5: Gap between the optimal value and the estimated one')
plt.show()
```



In [363]:

```python
v_optimal =[0.877,0.928,0.988,0,0.824,0.928,0,0.778,0.824,0.877,0.828]
iteration=[i for i in range(1000)]
gap=[np.max(abs(v_optimal-np.max(q_learning(i,5,0.95,1)[0],axis=1))) for i in range(1000)]
plt.plot(iteration,gap)
tot_reward=str(q_learning(1000,5,0.95,1)[1])
plt.title('The total reward accumulated after 1000 iteration for ε=1 is: ' + tot_reward)
plt.xlabel('iteration')
plt.ylabel('||v-v*||')
plt.grid()
plt.suptitle('Figure6: Gap between the optimal value and the estimated one')
plt.show()
```



In [317]:

```python
abs(np.max(q_learning(5000,5,0.95,0.1)[0],axis=1)-v_optimal)
```

Out[317]:

```
array([ 0.03035853,  0.01294132,  0.00263904,  0.        ,  0.07065536,
        0.00520729,  0.        ,  0.12462937,  0.0515077 ,  0.00745637,
        0.0972529 ])
```

In [433]:

```python
value_es=np.max(q_learning(20000,5,0.95,0.1)[0],axis=1)
print('The optimal value estimated one after 10000 iteration for ε=0.1 is:\n',value_es)
print('\nIt\'s maximum pointwise difference from the optimal value is (ε=0.1):\n', np.max(abs(
value_es-v_optimal)))
```

```
The optimal value estimated one after 10000 iteration for ε=0.1 is:
 [ 0.87982636  0.92974869  0.98906333  0.          0.82569565  0.92360642
  0.          0.77517404  0.72571866  0.86291688  0.69479574]

It's maximum pointwise difference from the optimal value is (ε=0.1):
 0.133204263896
```

In [432]:

```python
value_es=np.max(q_learning(20000,5,0.95,0.9)[0],axis=1)
print('The optimal value estimated one after 10000 iteration for ε=0.9 is:\n',value_es)
print('\nIt\'s maximum pointwise difference from the optimal value is (ε=0.9):\n', np.max(abs(
value_es-v_optimal)))
```

```
The optimal value estimated one after 10000 iteration for ε=0.9 is:
 [ 0.86274043  0.92378071  0.98694163  0.          0.78826138  0.92817264
  0.          0.75349863  0.81891388  0.87945874  0.82865148]

It's maximum pointwise difference from the optimal value is (ε=0.9):
 0.0357386187317
```

By analyzing all those results, we could affirm that the value of ε represent the trade-off between the maximum reward and the convergence to the optimal value, by choosing ε very small, we don't allow for much exploration then the convergence to the optimal value will be very slowly w.r.t to the number of iterations but the accumulated reward will be significant (as shown in Figure3). If ε is very large then we allow for a lot of exploration, the convergence to the optimal value will come rapidly but the accumulated reward will be very small or negative (as shown in Figure6)

By construing the last results (printed results), we can see that for the same number of iterations (20 000) for a value of ε=0.1 and ε=0.9, the estimated value for the second one is more precise

Q6: Is the optimal policy of an MDP affected by the change of the initial distribution μ0?

In [78]:

```python
def non_uniform_prior(n_state,r):
    z=np.random.random()
    if z<=r:
        state=np.random.randint(0,n_state//2+1)
    else:
        state=np.random.randint(n_state//2+1,n_state)
    return state
```

The function non_uniform_prior generates a initial probability over states that is not uniform as like the function env.reset() generates (if r=0.5 it will be uniform).

In [93]:

```python
def value_estimation_bis(n,Tmax,gamma,r):
    Q=np.zeros((env.n_states,4))
    N=np.zeros((env.n_states,4),dtype=int)
    for i in range(n):
        state=non_uniform_prior(env.n_states,r)
        if(0 in env.state_actions[state]):
            action=0
        else:
            action=3
        x_0,action_0=state,action
        N[x_0,action_0]+=1
        t=0
        S=0
        term=False
        while(t<Tmax and term==False):
            nexts, reward, term = env.step(state,action)
            state = nexts
            S+=gamma**(t-1)*reward
            if(0 in env.state_actions[state]):
                action=0
            else:
                action=3
            t+=1
        Q[x_0,action_0]+=S
    return np.divide(Q,N,out=np.zeros_like(Q), where=N!=0)
```

The function value_estimation_bis is the same as the function value_estimation where we change the prior distribution over states from a uniform to a non_uniform one and we add an additionnal argument r: the parameters that control the non_uniform distribution over all states.

In [101]:

```python
Q_uniform_prior=value_estimation(10000,5,0.95)
Q_nonuniform_prior_1=value_estimation_bis(10000,5,0.95,0.3)
Q_nonuniform_prior_2=value_estimation_bis(10000,5,0.95,0.1)
Q_nonuniform_prior_3=value_estimation_bis(10000,5,0.95,0.9)
```

In [102]:

```python
optimal_policy=np.argmax(Q_uniform_prior,axis=1)
optimal_policy_1=np.argmax(Q_nonuniform_prior_1,axis=1)
optimal_policy_2=np.argmax(Q_nonuniform_prior_2,axis=1)
optimal_policy_3=np.argmax(Q_nonuniform_prior_3,axis=1)
print(optimal_policy)
print(optimal_policy_1)
print(optimal_policy_2)
print(optimal_policy_3)
```