ML818 | Reinforcement Learning and Planning | CW1

Hugo Touchette

Started: 19 May 2022

Last updated: 28 July 2022

Python 3

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
```

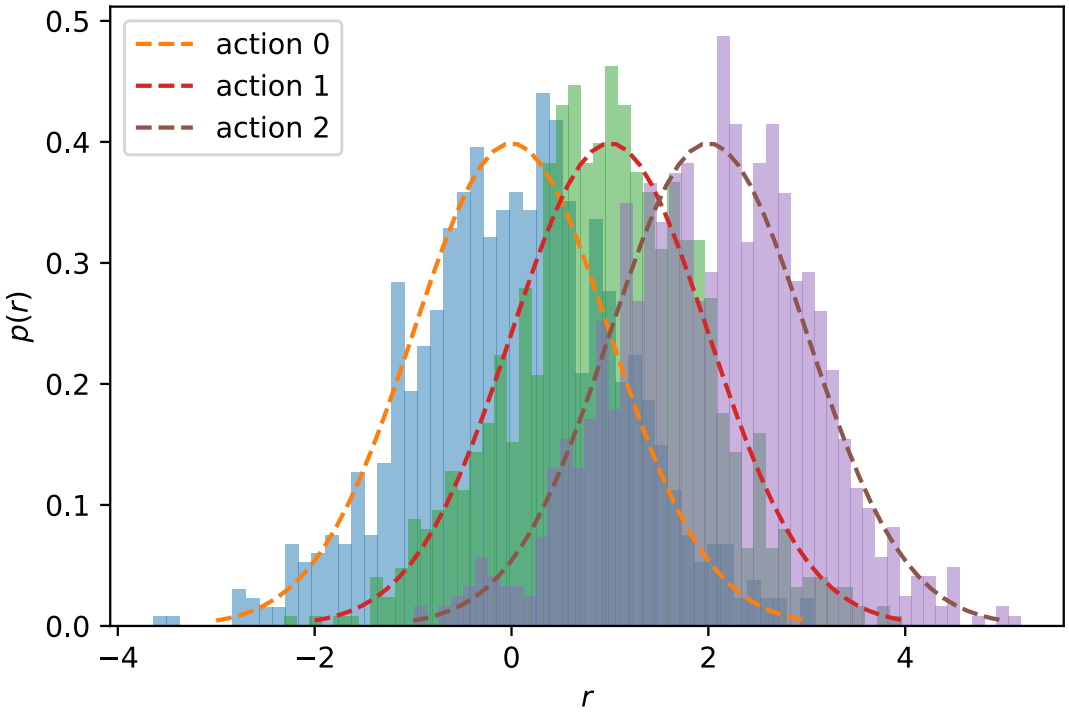
In [2]:
```
# Magic command for vectorised figures
%config InlineBackend.figure_format = 'svg'
```

Q1

(a)

In [3]:
```
def bandit_reward(a):
    return np.random.normal(loc=a, scale=1.0)
```

In [4]:
```
nsamples = 10**3
nactions = 3
rewrange = np.linspace(-3,3,50)
meanrew = np.zeros(3)

for a in range(nactions):
    reward_sample = [bandit_reward(a) for j in range(nsamples)]
    meanrew[a] = np.mean(reward_sample)

    plt.hist(reward_sample, bins=50, density=True, alpha=0.5)
    plt.plot(rewrange+a, stats.norm.pdf(rewrange+a, loc=a, scale=1), '--'

plt.legend(loc='upper left')
plt.xlabel(r'$r$')
plt.ylabel(r'$p(r)$')
plt.show()

for a in range(nactions):
    print('Mean reward for action', a, '=', meanrew[a])

print('Action with max reward is', np.argmax(meanrew))
```




```
Mean reward for action 0 = 0.05948172467820382
Mean reward for action 1 = 0.9786832303318072
Mean reward for action 2 = 2.0478934195046374
Action with max reward is 2
```

(b)

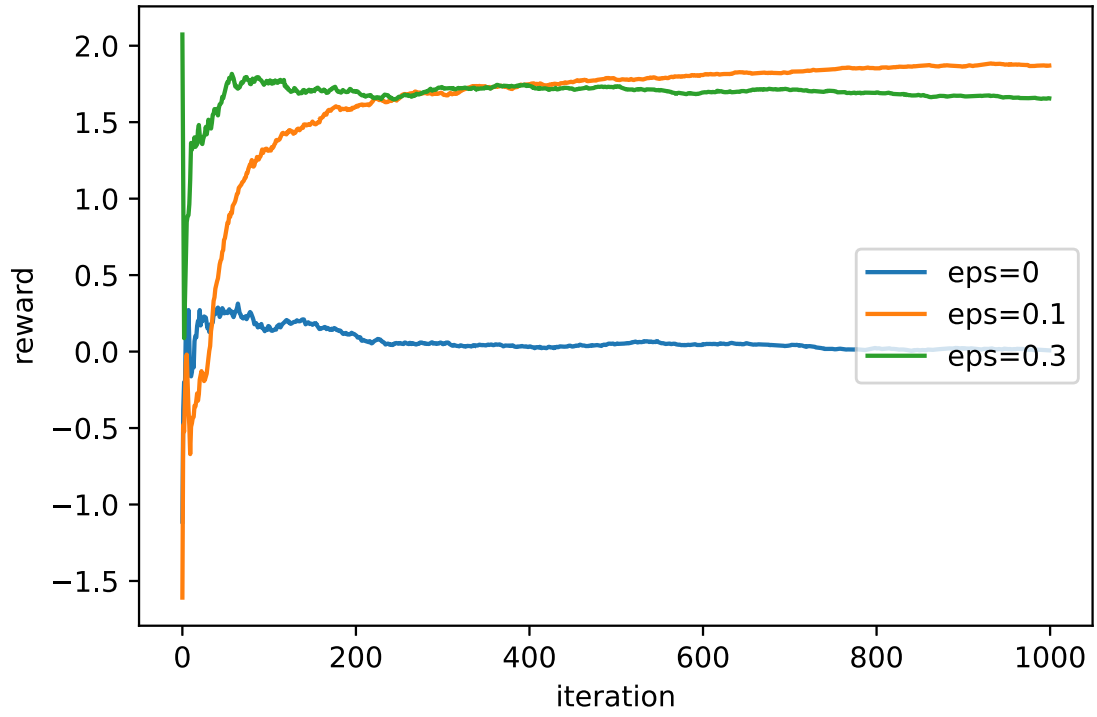
For a sequence (or trajectory) of actions and rewards, we estimate the value $Q(a)$ of action a as follows:

In [5]:
```python
def qa_estimator(actionseq, rewardseq, k):
    qa = np.zeros(k)
    numa = np.zeros(k)
    dena = np.zeros(k)

    for i, ai in enumerate(actionseq):
        numa[ai] += rewardseq[i]
        dena[ai] += 1

    for i in range(k):
        if dena[i]>0:
            qa[i] = numa[i]/dena[i]
        else:
            qa[i] = float('-inf')

    return qa
```

(c)

We apply next this estimator with a greedy action selection to a given trajectory:

In [6]:
```
nactions = 3
nsteps = 1000

for eps in [0, 0.1, 0.3]:
    a = 0
    alist = [a]
    rewlist = [bandit_reward(a)]
    estqa = np.zeros(nactions)
    totrew = 0.0
    totrewlist = []

    for k in range(nsteps):
        estqa = qa_estimator(alist, rewlist, nactions)

        if np.random.random()<eps:
            a = np.random.randint(0,3)
        else:
            a = np.argmax(estqa)

        rew = bandit_reward(a)
        totrew += rew

        alist.append(a)
        rewlist.append(rew)
        totrewlist.append(totrew/(k+1))

    plt.plot(range(nsteps), totrewlist, label='eps='+str(eps))

plt.legend(loc='center right')
plt.xlabel('iteration')
plt.ylabel('reward')
plt.show()
```

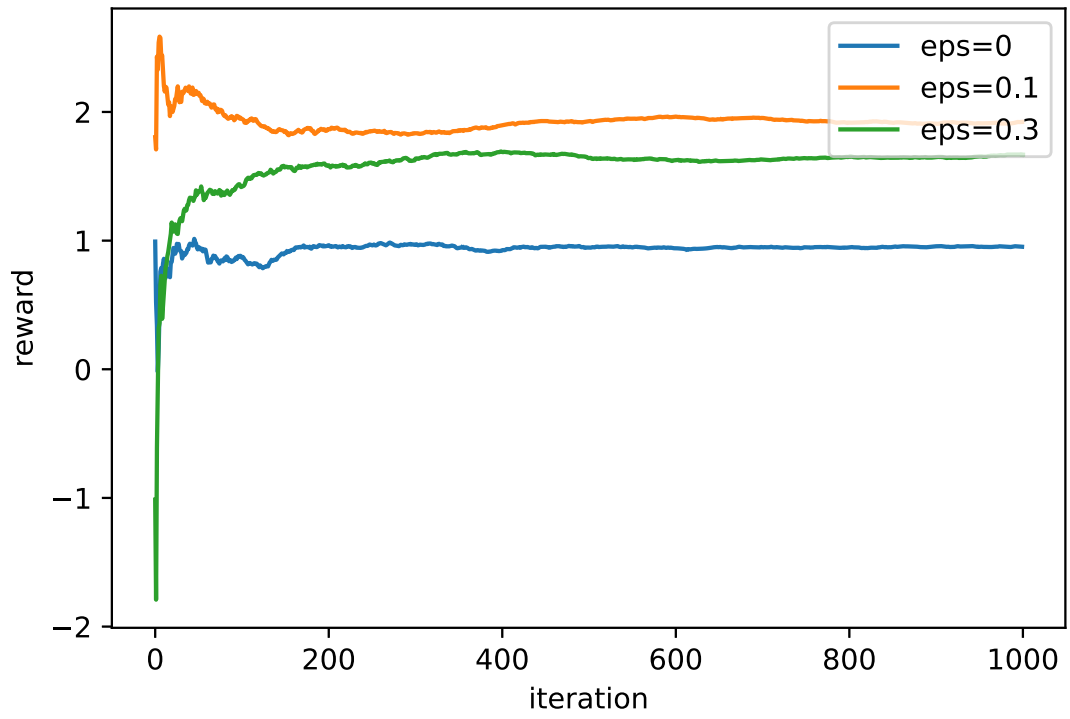


Exploration with $\varepsilon > 0$ is necessary to sample the highest reward (2), but too much exploration (with $\varepsilon = 0.3$) is sub-optimal because the actions selected are sub-optimal a 1/3 of the time.

(d)

In [33]:

```
nactions = 3
nsteps = 1000
baseline = 0.0

for eps in [0, 0.1, 0.3]:
    a = 0
    qa = np.zeros(nactions)+baseline
    totrew = 0.0
    totrewlist = []

    for n in range(nsteps):
        if np.random.random()<eps:
            a = np.random.randint(0,3)
        else:
            a = np.argmax(qa)

        rew = bandit_reward(a)
        alpha = 1/(n+1)
        qa[a] = (1-alpha)*qa[a] + alpha*rew
        totrew += rew
        totrewlist.append(totrew/(n+1))

    plt.plot(range(nsteps), totrewlist, label='eps='+str(eps))

plt.legend(loc='upper right')
plt.xlabel('iteration')
plt.ylabel('reward')
plt.show()
```



Similar results as in (c) if repeated enough times. The algorithm is now quicker because it is updated in an online way. As an improvement, we could anneal ϵ in time to decrease the amount of exploration and thus ensure that the search is progressively more optimal.

Effect of initial condition (simulations tried but not shown):

- $Q_0(a) = -10$: For a low initial baseline, any initial reward will lead future iterations in terms of exploitation. We therefore need more exploration (higher ϵ) to detect the 3rd lever as the optimal one.
- $Q_0(a) = 10$: A high baseline favors the 3rd lever, as it will take more iterations to readjust the value function of the 'lower' levers. In this case, exploration is not as important.

(e)

The ε-greedy algorithm can be seen as a learning algorithm for solving the k-armed bandit problem, but it is not a truly a reinforcement learning algorithm mainly because there is no system-agent interaction - the actions have rewards, but do not affect the state of a system. We say in this case that the actions are non-associative - they are not taken with respect to a context and don't influence any context.

Q2

(a)

The transition matrix is

In [8]:
```
Ptr = np.zeros((7,7))
for i in range(7):
    for j in range(7):
        if np.abs(i-j)==1:
            Ptr[i,j] = 0.4
    Ptr[i,i] = 0.2

Ptr[0,0] = 0.6
Ptr[6,6] = 0.6
Ptr
```

Out[8]:
```
array([[0.6, 0.4, 0. , 0. , 0. , 0. , 0. ],
       [0.4, 0.2, 0.4, 0. , 0. , 0. , 0. ],
       [0. , 0.4, 0.2, 0.4, 0. , 0. , 0. ],
       [0. , 0. , 0.4, 0.2, 0.4, 0. , 0. ],
       [0. , 0. , 0. , 0.4, 0.2, 0.4, 0. ],
       [0. , 0. , 0. , 0. , 0.4, 0.2, 0.4],
       [0. , 0. , 0. , 0. , 0. , 0.4, 0.6]])
```

This is a doubly stochastic matrix (rows and columns sum to 1), so its stationary distribution is the uniform distribution.

The formula in vector form for calculating the value function is

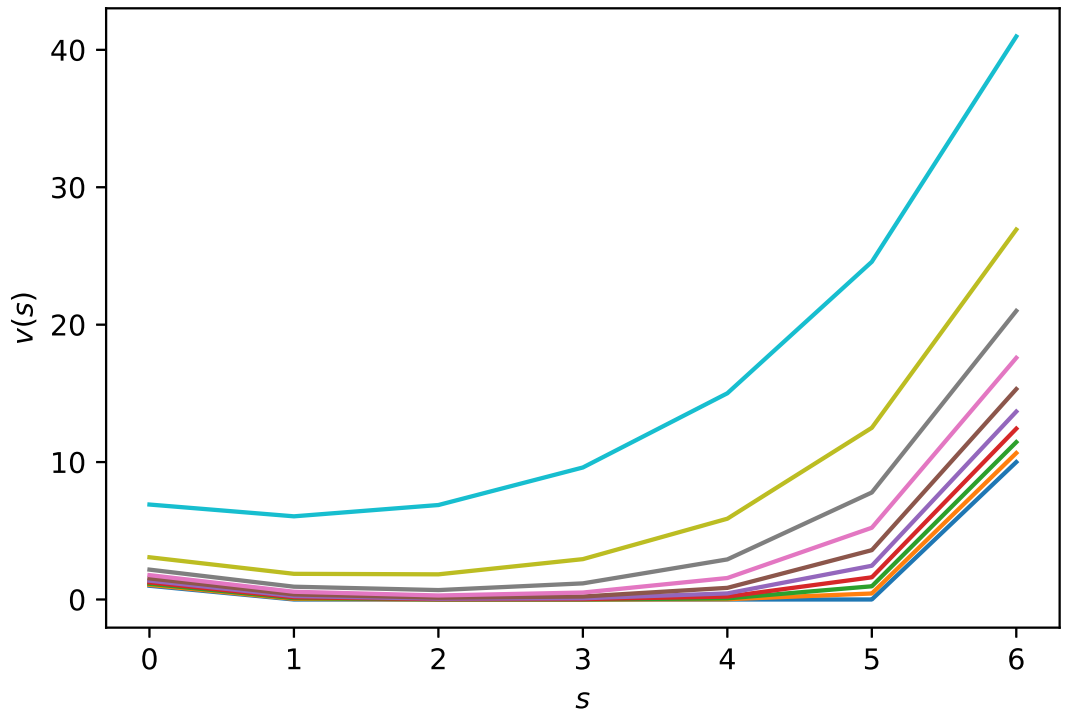
$$v = (I - \gamma P)^{-1}\rho,$$

where I is the identity matrix, P is the transition probability above, and ρ is the column vector containing the reward per state.

In [9]:
```
rho = np.array([1, 0, 0, 0, 0, 0, 10])
```

In [10]:
```
def vsol(gamma, P):
    return np.linalg.inv(np.identity(7)-gamma*P).dot(rho)
```

In [11]:
```
sval = range(7)

for gamma in np.arange(0.,1.0,0.1):
    plt.plot(sval, vsol(gamma, Ptr))

plt.xlabel(r'$s$')
plt.ylabel(r'$v(s)$')
plt.show()
```



(b)

We apply the iteration

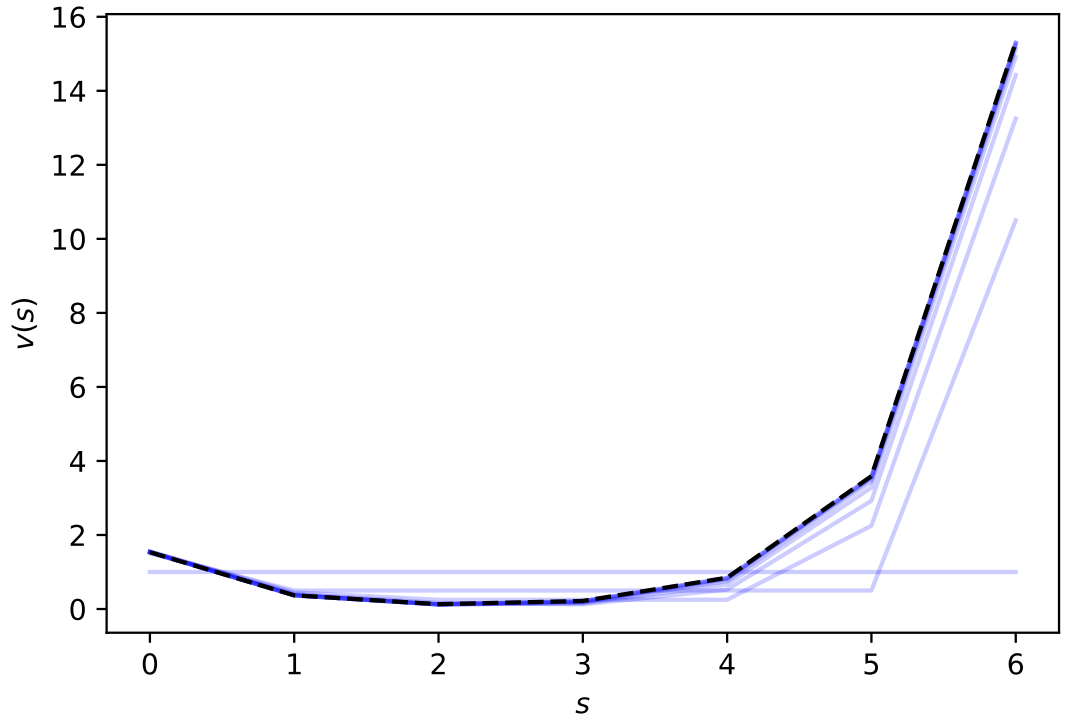
$$v_{k+1} = T(v_k)$$

using the Bellman operator $T(v) = \rho + \gamma P v$.

In [12]:
```
def TBellman(v, gamma, P):
    return rho + gamma*P.dot(v)
```

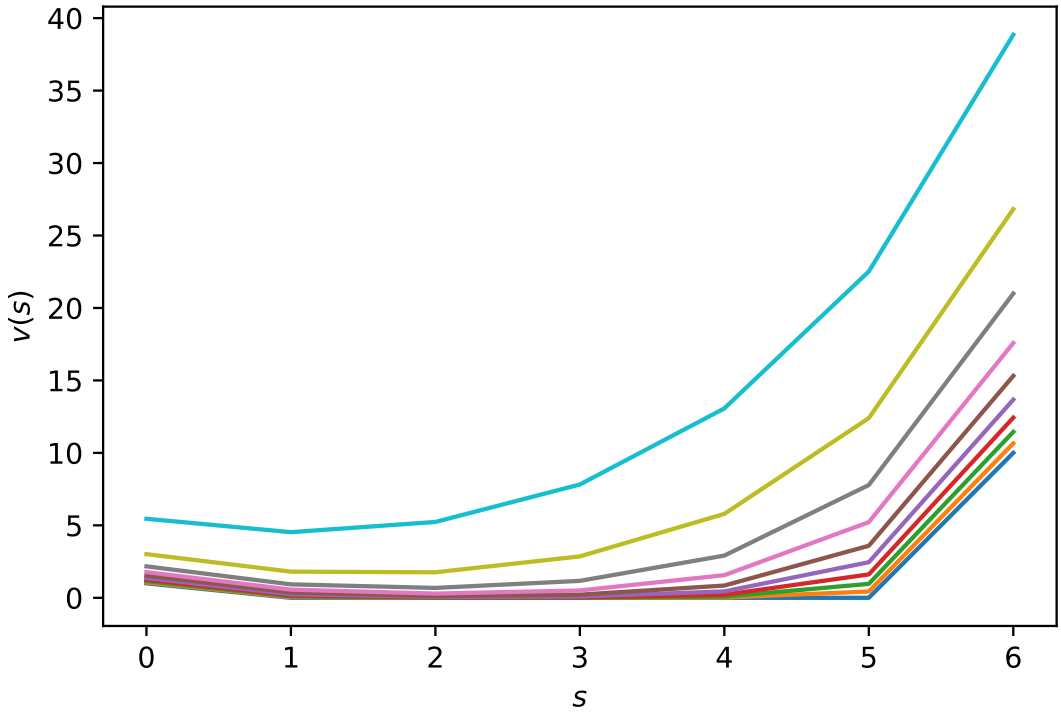
Here's the result of 10 iterations for $\gamma = 0.5$:

In [13]:
```
niter = 10
gamma = 0.5
v = np.ones(7)
for k in range(niter):
    plt.plot(sval, v, 'b-', alpha=0.2)
    v = TBellman(v, gamma, Ptr)

plt.plot(sval, vsol(gamma, Ptr), 'k--')
plt.xlabel(r'$s$')
plt.ylabel(r'$v(s)$')
plt.show()
```



The iteration converges, although larger γ values require more iterations. Repeating for different γ:

In [14]:
```
niter = 20
for gamma in np.arange(0., 1.0, 0.1):
    v = np.ones(7)
    for k in range(niter):
        v = TBellman(v, gamma, Ptr)
    plt.plot(sval, v)

plt.xlabel(r'$s$')
plt.ylabel(r'$v(s)$')
plt.show()
```

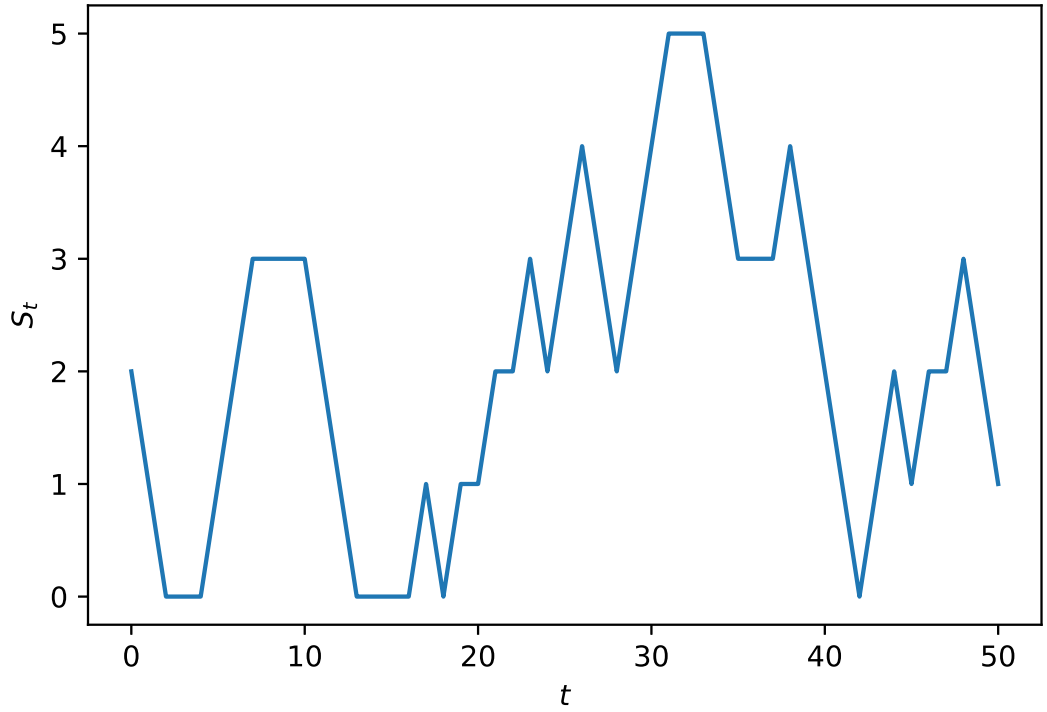


(c)

For generating trajectories, we need to generate random choices for going up or down, or, for border states, for staying in those states:

In [15]:
```
def next_state(s):
    r = np.random.random()
    if s==0:
        if r<0.4: s+=1 # Move up from 0
    elif s==6:
        if r<0.4: s-=1 # Move down from 6
    else:
        if r<0.4:
            s+=1       # Move up
        elif r<0.8:
            s-=1       # Move down
    return s
```

In [16]:
```
# Parameters
gamma = 0.5
nsteps = 50
slist = []

# Starting state and its reward
s = 2
slist.append(s)
reward = rho[s]

# Generate state sequence (trajectory)
for i in range(nsteps):
    s = next_state(s)
    slist.append(s)
    reward += gamma**(i+1) * rho[s]

plt.plot(range(nsteps+1), slist)
plt.xlabel(r'$t$')
plt.ylabel(r'$S_t$')
plt.show()
print('Final reward:', reward)
```




```
Final reward: 0.4377326965334305
```

(d)

The estimator of $v(s)$ is obtained by averaging the random reward of many simulated trajectories (remember the Monte Carlo course).

In [17]:
```
gamma = 0.5
nsteps = 1000
niter = 1000
rewardest = 0.0

for j in range(niter):
    s = 2
    reward = rho[s]
    for i in range(nsteps):
        s = next_state(s)
        reward += gamma**(i+1) * rho[s]

    rewardest += reward

print('Estimated v(s):', rewardest/niter)
print('True value:', vsol(gamma, Ptr)[2])
```

```
Estimated v(s): 0.12519526553529844
True value: 0.1304331838806863
```

Q3

(a)

The MoveLeft ($a = -1$) and MoveRight ($a = +1$) transition matrices are:

In [18]:
```
PMoveLeft = np.zeros((7,7))
for i in range(7):
    for j in range(7):
        if i-j==1:
            PMoveLeft[i,j]=1.0

PMoveLeft[0,0]=1.0
PMoveLeft
```

Out[18]:
```
array([[1., 0., 0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0.]])
```

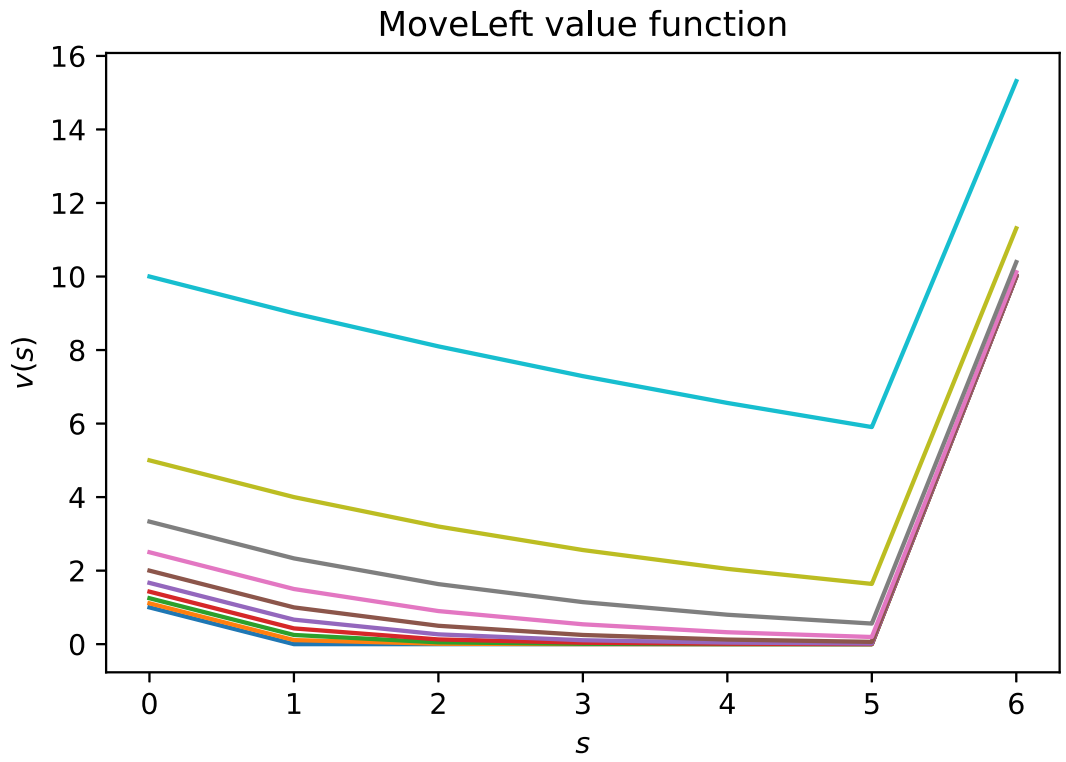
In [19]:
```
PMoveRight = np.zeros((7,7))
for i in range(7):
    for j in range(7):
        if j-i==1:
            PMoveRight[i,j]=1.0

PMoveRight[6,6]=1.0
PMoveRight
```

Out[19]:
```
array([[0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 0., 0., 1.]])
```

(b)

The value function for the full MoveLeft policy is the value function of the Markov reward process with the MoveLeft transition matrix. We use the function defined earlier to obtain the value function:

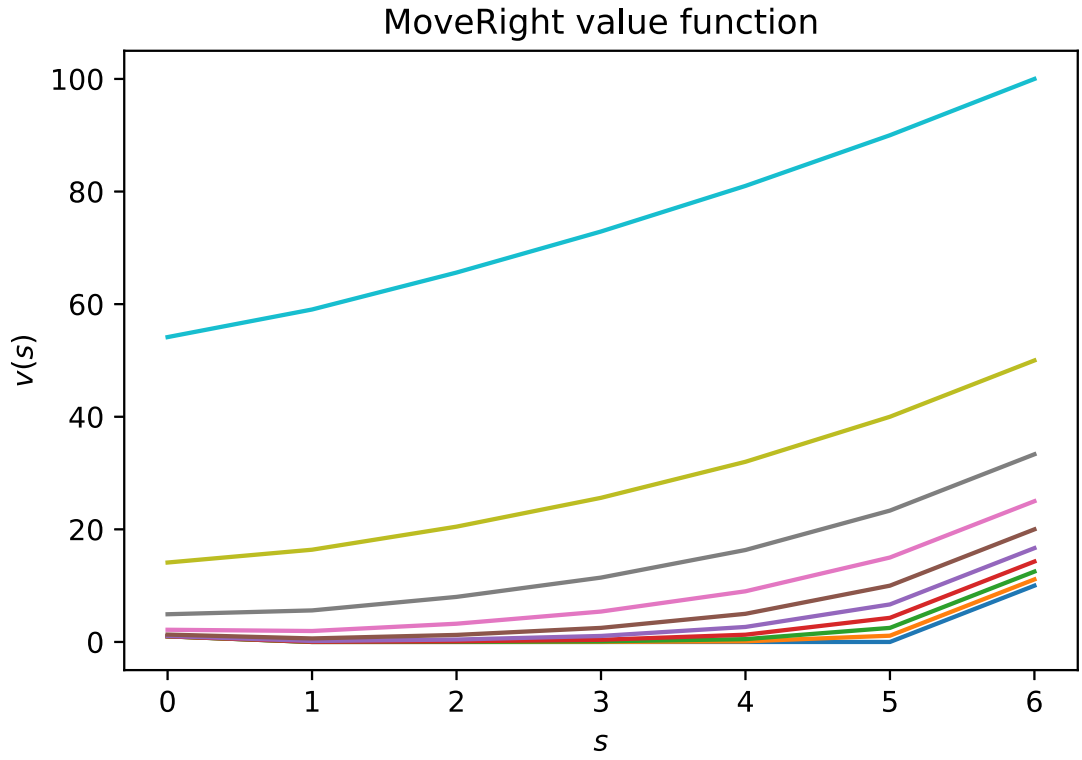
In [20]:
```
for gamma in np.arange(0.0, 1.0, 0.1):
    plt.plot(sval, vsol(gamma, PMoveLeft), label=gamma)

plt.xlabel(r'$s$')
plt.ylabel(r'$v(s)$')
plt.title('MoveLeft value function')
plt.show()
```



Analysis:

- $v(s)$ is min at $s = 5$ because this state is furthest away from the reward at $s = 0$.
- $v(s)$ increases as we go to $s = 0$ as it takes fewer steps to get to $s = 0$.
- $v(s)$ is largest at $s = 6$ where the reward is largest.
- For $\gamma = 0$, $v(s)$ is the reward: 0, 1 or 10 depending on s.
- For $\gamma > 0$, $v(0) > \rho(0) = 1$ because of the accumulated reward coming from staying at $s = 0$.

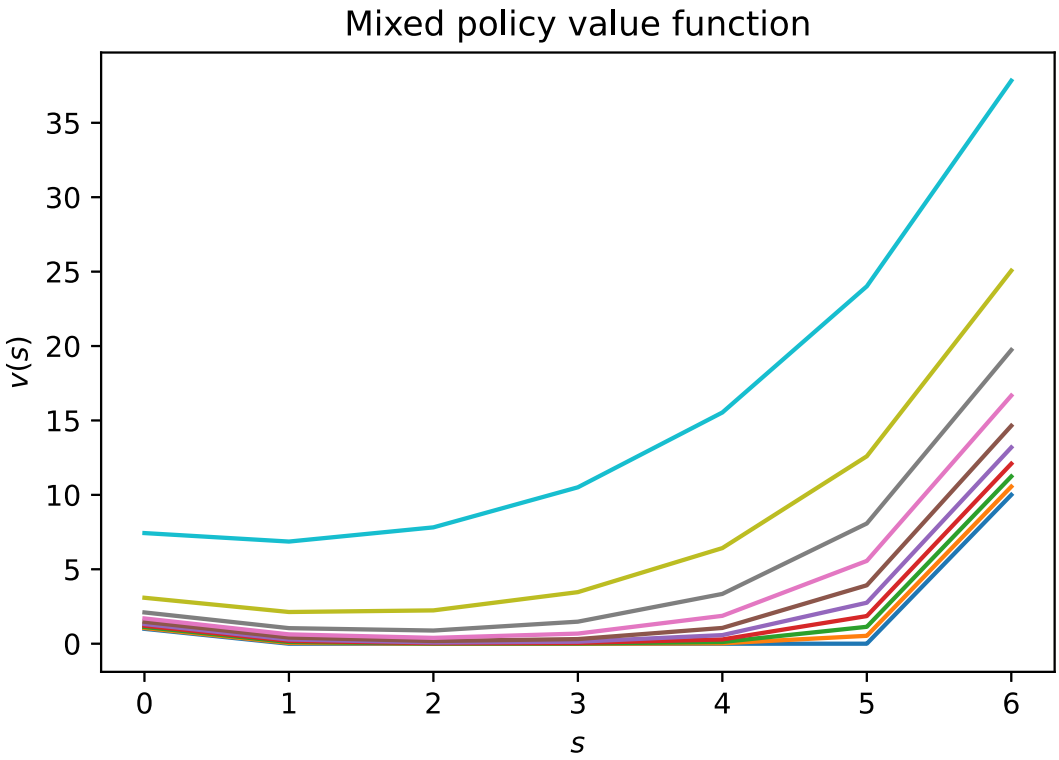
In [21]:
```
for gamma in np.arange(0.0, 1.0, 0.1):
    plt.plot(sval, vsol(gamma, PMoveRight))

plt.xlabel(r'$s$')
plt.ylabel(r'$v(s)$')
plt.title('MoveRight value function')
plt.show()
```



Analysis: For $\gamma > 0$,

- $v(s)$ is min at $s = 0$, the state of min reward.
- $v(s)$ increases as we go to $s = 6$ as the max reward can be achieved by moving right.
- $v(s)$ is largest at $s = 6$ where the reward is largest.
- $v(6) > \rho(6) = 10$ because of the accumulated reward coming from staying at $s = 10$.

In [22]:
```
Pmix = 0.5*PMoveLeft+0.5*PMoveRight
for gamma in np.arange(0.0, 1.0, 0.1):
    plt.plot(sval, vsol(gamma, Pmix))

plt.xlabel(r'$s$')
plt.ylabel(r'$v(s)$')
plt.title('Mixed policy value function')
plt.show()
```



Here we have the effect of being able to move left or right and to obtain, as a result, rewards by either reaching the state $s = 0$ with 1 reward or the state $s = 6$ with 10 reward.

(c)

The Bellman optimality equation is

$$v_*(s) = \max_a \mathbb{E}[R_{t+1} + v_*(s)|S_t = a, A_t = a].$$

For our model,

- R_{t+1} does not depend on A_t, only on S_t
- There are only two actions, moving left or right
- The transition matrix for either action is deterministic: the state can only move up or down or stay the same.

For the move left action, the expectation is thus

$$\rho(s) + v_*(s')$$

where $s' = s$ for $s = 0$ and $s' = s - 1$ otherwise. For moving right, we have instead

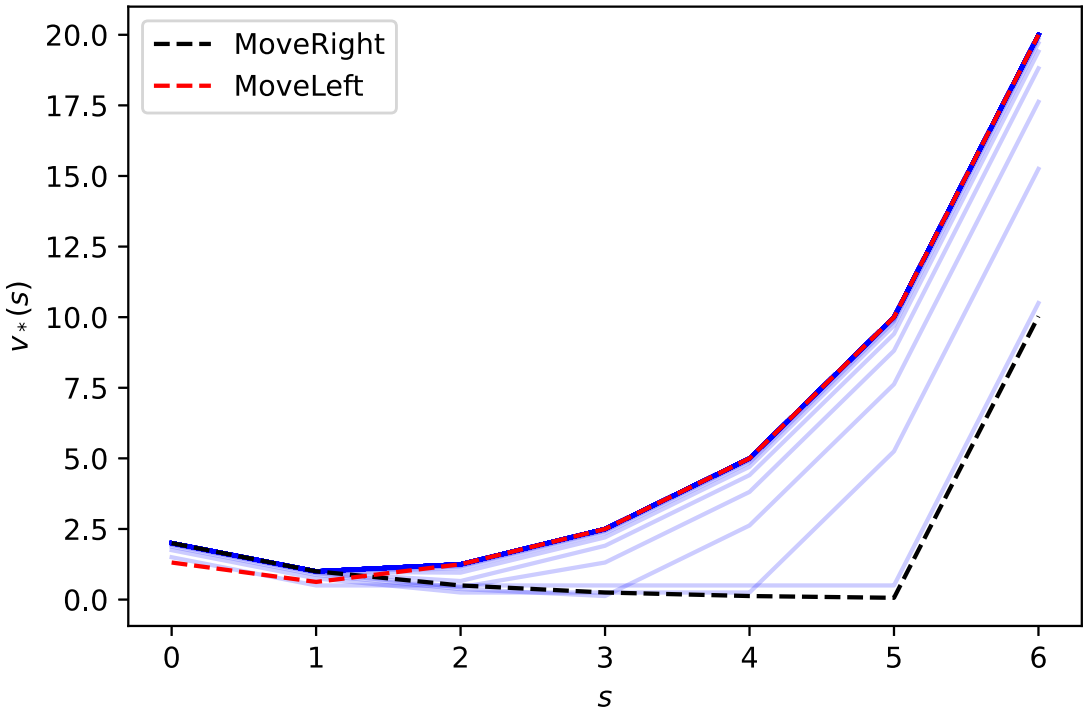
$$\rho(s) + v_*(s')$$

where $s' = s$ for $s = 6$ and $s' = s + 1$ otherwise. Taking the max between the two yields $v_*(s)$.

(d)

In [23]:

```
gamma = 0.5
niter = 100
v = np.ones(7)
policy = np.zeros(7)

for k in range(niter):
    vtemp = np.copy(v)

    # Apply Bellman equation for each state
    for s in range(7):
        if s==0:
            rewardMoveLeft = rho[s] + gamma*v[s]
        else:
            rewardMoveLeft = rho[s] + gamma*v[s-1]

        if s==6:
            rewardMoveRight = rho[s] + gamma*v[s]
        else:
            rewardMoveRight = rho[s] + gamma*v[s+1]

        # Take max of two actions
        if rewardMoveRight>rewardMoveLeft:
            vtemp[s] = rewardMoveRight
            policy[s] = 1
        else:
            vtemp[s] = rewardMoveLeft
            policy[s] = 0

    v = vtemp
    plt.plot(sval, v, 'b-', alpha=0.2)

vopt = v # For Q4

plt.plot(sval, vsol(gamma, PMoveLeft), 'k--', label='MoveRight')
plt.plot(sval, vsol(gamma, PMoveRight), 'r--', label='MoveLeft')
plt.legend(loc="upper left")
plt.xlabel(r'$s$')
plt.ylabel(r'$v_*(s)$')
plt.show()

print('Optimal value function:', v)
print('Optimal policy', policy)
print('MoveLeft value function:', vsol(gamma, PMoveLeft))
print('MoveRight value function:', vsol(gamma, PMoveRight))
```



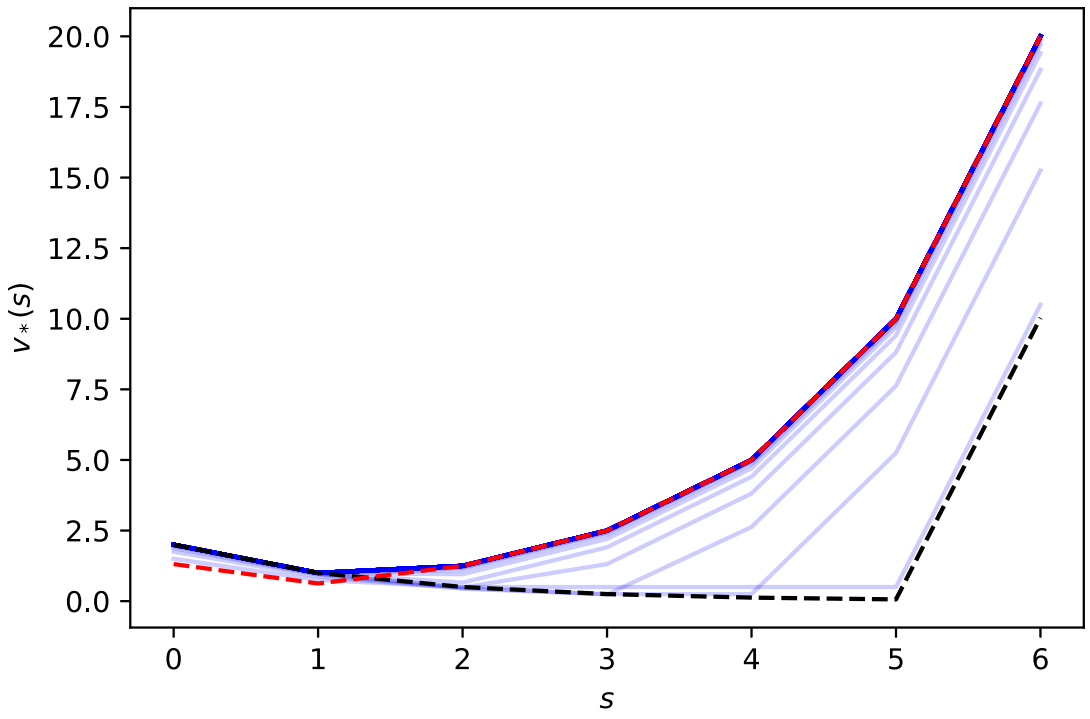

```
Optimal value function: [ 2.    1.    1.25  2.5   5.   10.   20.  ]
Optimal policy [0. 0. 1. 1. 1. 1. 1.]
MoveLeft value function: [ 2.       1.       0.5      0.25     0.125    0
.0625  10.03125]
MoveRight value function: [ 1.3125  0.625   1.25    2.5     5.     10.
20.    ]
```

- The optimal value function comes from a trade-off between the MoveLeft value function and the MoveRight value function.
- MoveLeft is more optimal than MoveRight for states 0 and 1, as shown in the optimal policy, whereas MoveRight is more optimal than MoveLeft for all the other states.
- Close to state 0, it is better to move left to that state, whereas from state 2, it is better to move right to state 6.
- The optimal policy is determined by moving left or right from s to the largest value function (this is Bellman's optimality equation with 0 reward when in states $1, \ldots, 5$).

(e)

The in-place version of the code follows by deleting the reference to vtemp:

In [24]:
```
gamma = 0.5
niter = 100
v = np.ones(7)
policy = np.zeros(7)

for k in range(niter):
    for s in range(7):
        if s==0:
            rewardMoveLeft = rho[s] + gamma*v[s]
        else:
            rewardMoveLeft = rho[s] + gamma*v[s-1]

        if s==6:
            rewardMoveRight = rho[s] + gamma*v[s]
        else:
            rewardMoveRight = rho[s] + gamma*v[s+1]

        if rewardMoveRight>rewardMoveLeft:
            v[s] = rewardMoveRight
            policy[s] = 1
        else:
            v[s] = rewardMoveLeft
            policy[s] = 0

    plt.plot(sval, v, 'b-', alpha=0.2)

plt.plot(sval, vsol(gamma, PMoveLeft), 'k--')
plt.plot(sval, vsol(gamma, PMoveRight), 'r--')
plt.xlabel(r'$s$')
plt.ylabel(r'$v_*(s)$')
plt.show()
```



The end result is the same, although the intermediate value functions might be different.

(f)

Staying at a site other than 0 or 6 only delays the reward gained by reaching those states, so adding STAY as an action will not change the optimal value function and policy as it will not be selected as a 'good' action.

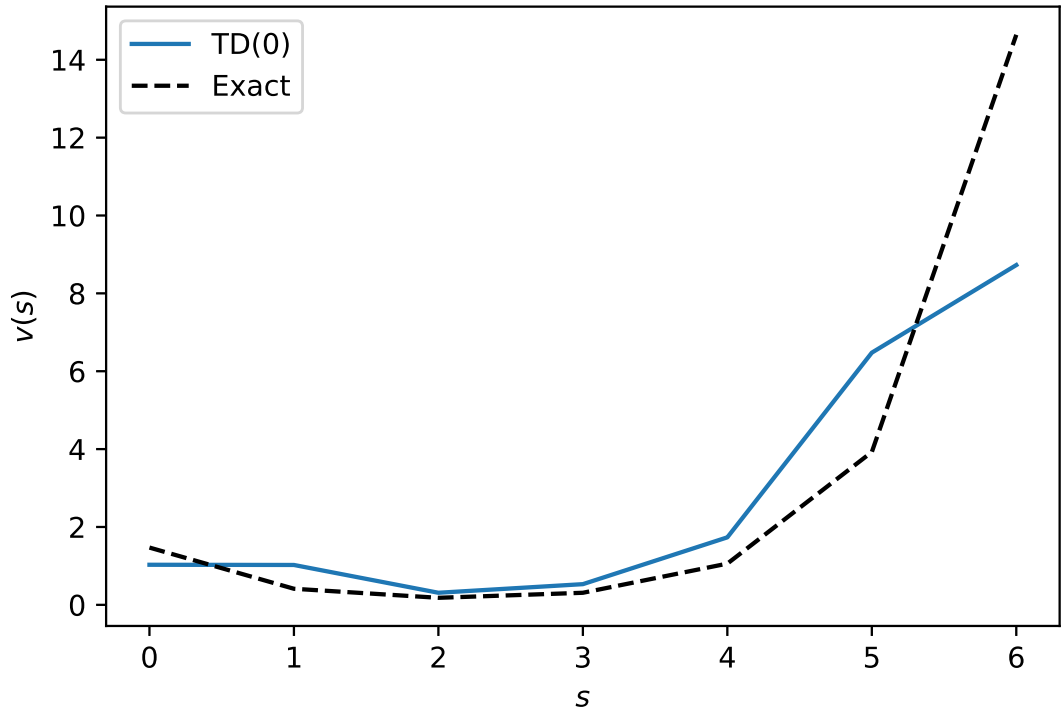
Adding an action from state 3 to 6 would change the optimal solution, though, since the reward at 6 could be reached sooner by going via 3 and taking that 'global' move.

Q4

(a)

In [25]:
```
def update_state(s, a):
    if a==0:
        if s==0:
            return 0
        else:
            return s-1 # Move left
    if a==1:
        if s==6:
            return 6
        else:
            return s+1 # Move right
```

In [26]:
```
episodes = 100
finaltime = 1000
gamma = 0.5
alpha = 0.1
v = np.zeros(7)

for j in range(episodes):
    s = np.random.randint(0,7)

    for k in range(finaltime):
        a = np.random.choice([0,1])
        sp = update_state(s, a)
        reward = rho[sp]
        v[s] = (1-alpha)*v[s] + alpha*(reward + gamma*v[sp])
        s = sp

plt.plot(sval, v, label='TD(0)')
plt.plot(sval, vsol(gamma, 0.5*PMoveLeft+0.5*PMoveRight), 'k--', label='E
plt.legend(loc='upper left')
plt.xlabel(r'$s$')
plt.ylabel(r'$v(s)$')
plt.show()
```



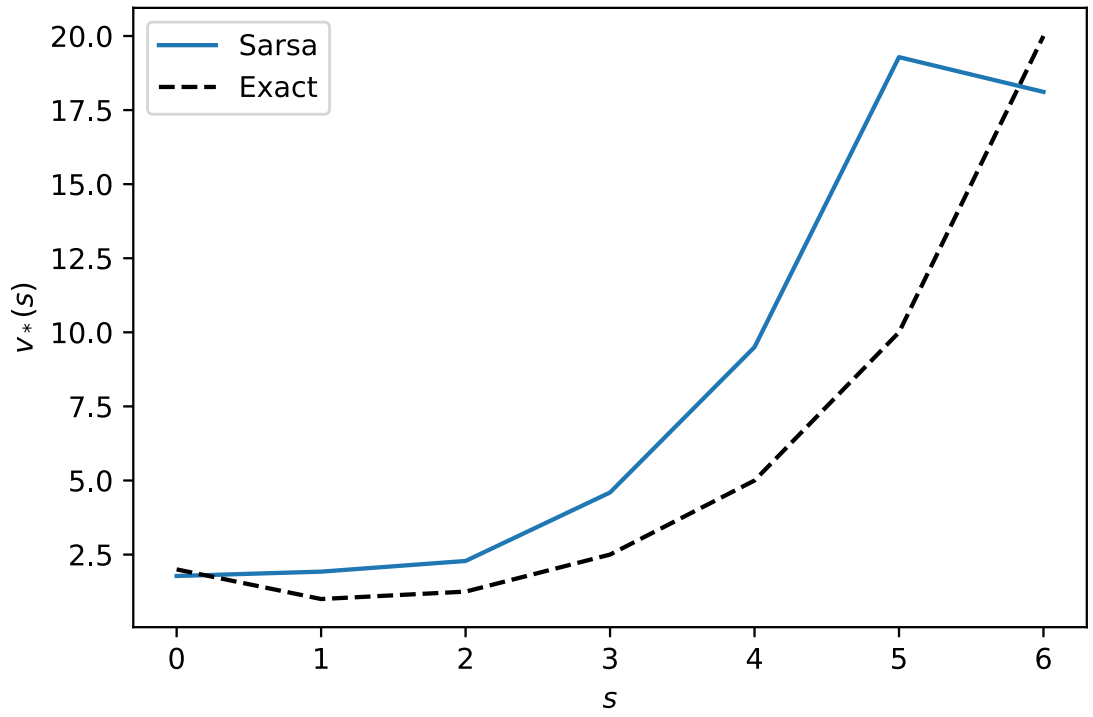
To obtain better convergence, we should decrease the annealing parameter in time.

(b)

In [27]:

```
def eps_greedy_action(qval, eps):
    if np.random.random()<eps:
        a = np.random.choice([0,1])
    else:
        a = np.argmax(qval)

    return a
```

In [28]:
```
episodes = 1000
finaltime = 1000
gamma = 0.5
eps = 0.1
alpha = 0.2
q = np.zeros((7,2))

for j in range(episodes):
    s = np.random.randint(0,7)
    a = eps_greedy_action(q[s,:], eps)

    for k in range(finaltime):
        sp = update_state(s, a)
        reward = rho[sp]
        ap = eps_greedy_action(q[sp,:], eps)
        q[s, a] = (1-alpha)*q[s, a] + alpha*(reward + gamma*q[sp, ap])
        s = sp
        a = ap

v = np.max(q, axis=1)
policy = np.argmax(q, axis=1)

plt.plot(sval, v, label='Sarsa')
plt.plot(sval, vopt, 'k--', label='Exact')
plt.legend(loc='upper left')
plt.xlabel(r'$s$')
plt.ylabel(r'$v_*(s)$')
plt.show()
print('Optimal policy:', policy)
```

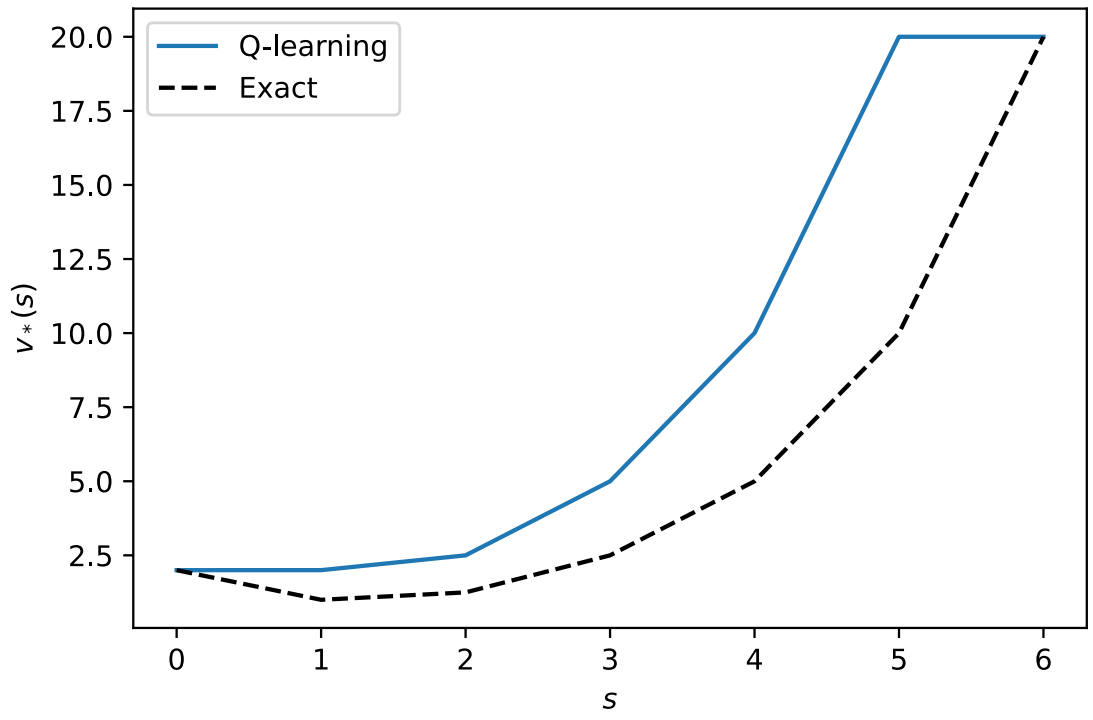



```
Optimal policy: [0 0 1 1 1 1 1]
```

The results overestimate the true value function (positive bias) but the optimal policy obtained is correct. There's also a bias because we never decrease ϵ to 0, so we're never fully optimal, and don't decrease the annealing parameter α. Try decreasing those in time to see what you get.

(c)

In [29]:
```
episodes = 1000
finaltime = 1000
gamma = 0.5
eps = 0.1
alpha = 0.2
q = np.zeros((7,2))

for j in range(episodes):
    s = np.random.randint(0,7)

    for k in range(finaltime):
        a = eps_greedy_action(q[s,:], eps)
        sp = update_state(s, a)
        reward = rho[sp]
        maxq = np.max(q[sp,:])
        q[s, a] = (1-alpha)*q[s, a] + alpha*(reward + gamma*maxq)
        s = sp

v = np.max(q, axis=1)
policy = np.argmax(q, axis=1)

plt.plot(sval, v, label='Q-learning')
plt.plot(sval, vopt, 'k--', label='Exact')
plt.legend(loc='upper left')
plt.xlabel(r'$s$')
plt.ylabel(r'$v_*(s)$')
plt.show()
print('Optimal policy:', policy)
```




```
Optimal policy: [0 0 1 1 1 1 1]
```

Q5

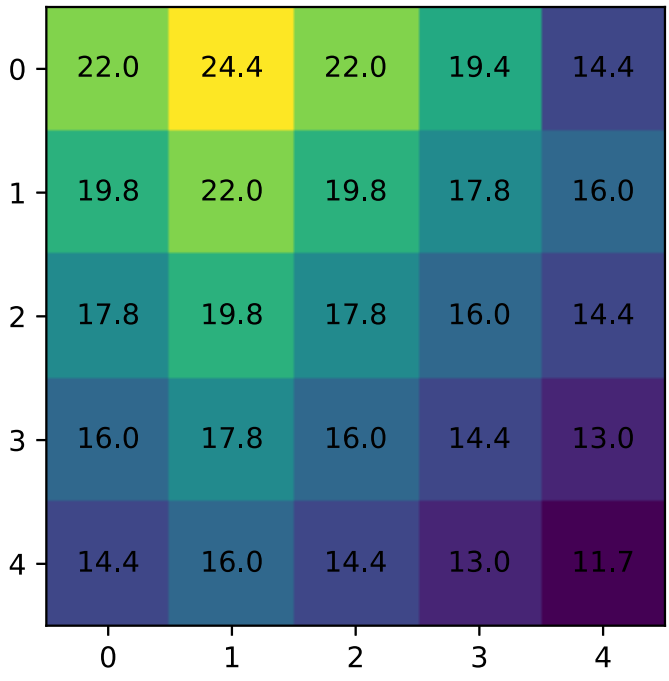
The actions (N, S, E, W) are coded as (0, 1, 2, 3) respectively for the purpose of indexing the actions. The cells in the grid, corresponding to the states, are coded in the normal matrix way.

In [30]:

```
def gridmove(sx, sy, a):
    # Special states
    if sx==0 and sy==1:
        return 4, 1, 10
    elif sx==0 and sy==3:
        return 2, 3, 5
    # North
    if a==0:
        if sx==0:
            return sx, sy, -1
        else:
            return sx-1, sy, 0
    # South
    elif a==1:
        if sx==4:
            return sx, sy, -1
        else:
            return sx+1, sy, 0
    # East
    elif a==2:
        if sy==4:
            return sx, sy, -1
        else:
            return sx, sy+1, 0
    # West
    elif a==3:
        if sy==0:
            return sx, sy, -1
        else:
            return sx, sy-1, 0
```

In [31]:

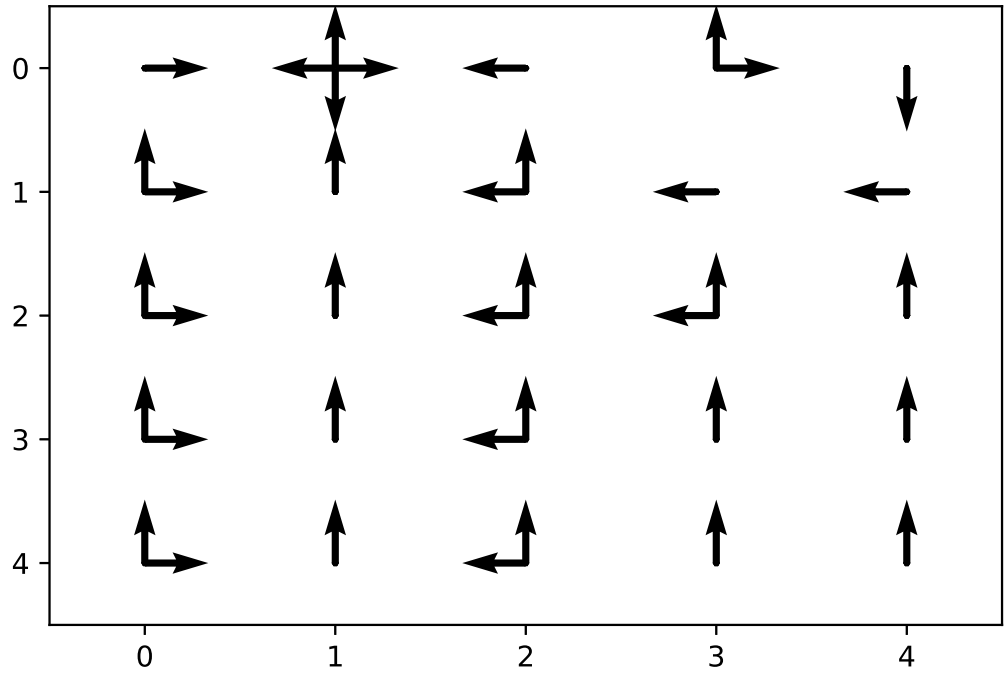
```
episodes = 1000
finaltime = 1000
eps = 0.1
alpha = 0.2
gamma = 0.9
q = np.zeros((5,5,4))

for j in range(episodes):
    sx = np.random.randint(0,5)
    sy = np.random.randint(0,5)

    for k in range(finaltime):
        # epsilon-greedy action
        if np.random.random()<eps:
            a = np.random.randint(0,4)
        else:
            a = np.argmax(q[sx,sy,:])

        # Next state and reward from action
        sxp, syp, reward = gridmove(sx, sy, a)

        # Q-learning update
        maxq = np.max(q[sxp, syp, :])
        q[sx, sy, a] = (1-alpha)*q[sx, sy, a] + alpha*(reward + gamma*max

        # Next step override
        sx = sxp
        sy = syp

# Value function from q
q = np.round(q, 1)
v = np.max(q, axis=2)

plt.imshow(v)
for (i, j), z in np.ndenumerate(v):
    plt.text(j, i, z, ha='center', va='center')

plt.show()
```



The values are sensibly the same as those found in Sutton and Barto, p. 53.

To get the optimal policy, we have to look at all the actions that achieve the same max in the action value function.

In [32]:
```
# Mapping of actions to vectors
dirx = [0,0,1,-1]
diry = [1,-1,0,0]

# Meshgrid and containers for the vector plot
x, y = np.meshgrid(np.linspace(0,4,5),np.linspace(0,4,5))
dirux = np.zeros((5,5,4))
dirvy = np.zeros((5,5,4))

for i in range(5):
    for j in range (5):

        # Max q = value function defining optimal actions
        m = np.max(q[i,j,:])

        # Check all entries to see if the same as max
        for a, b in enumerate(q[i,j,:]):
            if b==m:
                dirux[i,j,a]=dirx[a]
                dirvy[i,j,a]=diry[a]

for a in range(4):
    plt.quiver(x, y, dirux[:,:,a], dirvy[:,:,a], scale=15)

plt.xlim(-0.5,4.5)
plt.ylim(4.5,-0.5)
plt.show()
```



The optimal policy is also nearly the same: some actions are missing, possibly due to the simulation being limited in the number of time steps and episodes.