

# Markov Decision Processes and the UCT algorithm

Carlo Hämäläinen  
carlo.hamalainen@gmail.com

October 28, 2009

## Abstract

In this note we introduce the concept of a Markov Decision Process. We show how to solve for the optimal policy and value vectors for two examples (iPod shuffle and sailing shortest path) using value iteration. We then show how to use the UCT algorithm on the same two problems. Complete source code (in Python) is available.

## 1 Introduction

This file and related source code is available at  
<http://carlo-hamalainen.net/stuff/mdpnotes>

An MDP (Markov Decision Process) is fully described by the following items:

- A set of states  $S$ . Here we will only consider the case that  $S$  is finite, but it may in general be infinite.
- A set of actions  $A$ . An action is a function that takes us from some state to a new state.
- For each action  $a \in A$  and state  $s \in S$ , the transition probability  $P_a(s, s')$  is the probability of moving from state  $s$  to  $s'$  under action  $a$ .
- For each state  $s$ , the (local) cost of taking action  $a \in A$  is denoted  $\text{cost}(s, a)$ .

Here we will assume that the cost is fixed for each state-action pair  $(s, a)$ . (It is possible to extend the definition to stochastic costs.)

If we reach a *target state*  $t \in S$  then we stop. The goal is to find a *policy*  $\pi$  that gives the optimal action  $\pi(s) \in A$  for each state. The Markov property means that the transition probabilities  $P_a(s, s')$  only depend on the current state  $s$  and the action  $a$  (so no information about past states is used). Once we have an optimal policy  $\pi$  we can use it in a simulation as shown in the pseudo-Python of Figure 1. The `next_state` function assumes that we have available a *generative model* of the

---

```
s = initial_state
this_cost = 0
while s != target_state:
    best_action = pi[s]
    this_cost += cost(s, best_action)
    s = next_state(s, best_action)

print this_cost
```

---

Figure 1: Using an optimal policy  $\pi$  to run a simulation of an MDP.

MDP that allows us to sample from the possible next states given a state and action. In particular we do not need to know the explicit transition probabilities  $P_a(s, s')$ .

## 2 iPod example

A “real world” example [5] comes from the iPod shuffle, an MP3 player with two modes: sequential (in order) play and shuffle. There is no screen so finding a particular song can be difficult – you can either jump around randomly or try to move forwards (or backwards) in a linear manner.

We can assume that the  $N$  songs on our iPod have been sorted in some sensible way, and we’ll label the songs  $0, 1, \dots, N - 1$ . There are two actions that we can take at any given song (state)

- Sequential: set the iPod to non-random playback and use the forward or backward track buttons to get to the target song. If we are at song  $s$  and the

target song is  $t$ , then this strategy will take  $|s - t|$  button presses to find the target.

- Shuffle: set the iPod to random playback and move to the next track if we are not at the target song. This incurs a cost  $T$  because it will take us some time to recognise the new song.

In a typical execution of this algorithm we will start at some initial song, randomly skip around a bit, and then use the sequential action to get to the final song. Or, if the target song is very close, we would only use the sequential strategy.

Suppose that the iPod has  $N$  songs. The set of states for this MDP is

$$S = \{0, 1, \dots, N - 1\}.$$

There are two actions, so

$$A = \{\text{sequential}, \text{shuffle}\}.$$

For the costs, the sequential strategy takes  $|s - t|$  button presses from state  $s$ , so

$$\text{cost}(s, \text{sequential}) = |s - t|.$$

The shuffle action only takes us to a different track but we have to recognise if this is the target song, so

$$\text{cost}(s, \text{shuffle}) = T$$

for some constant  $T$ . Finally we have the transition probabilities. The sequential action is guaranteed to take us to the target song, so  $P_{\text{sequential}}(s, s') = 1$  if  $s' = t$  and 0 otherwise. The shuffle action merely takes us to another track, with equal probability, so  $P_{\text{shuffle}}(s, s') = 1/N$  for any state  $s'$ .

### 3 Value iteration

To find the optimal policy  $\pi$  we will compute the value vector  $V$ . For each state  $s \in S$ ,  $V(s)$  is the expected cost of reaching the target state, using the best possible sequence of actions starting at state  $s$ .

We can start from an initial value vector  $V_0(s) = 0$  for all  $s \in S$ . Then the update step is

$$V_{t+1}(s) = \min_{a \in A} \left\{ \text{cost}(s, a) + \sum_{s' \in S} P_a(s, s') V_t(s') \right\}.$$

In other words, we minimise the sum of the local cost of taking some action, along with the expected cost from the possible new states. The policy is updated at each step by

$$\pi_{t+1}(s) = \operatorname{argmin}_{a \in A} \left\{ \operatorname{cost}(s, a) + \sum_{s' \in S} P_a(s, s') V_t(s') \right\}.$$

The value iteration algorithm is guaranteed to converge [1].

We can also apply a discounting factor  $\gamma$  to reflect the fact that costs (or rewards) incurred in the future are worth less than ones incurred in the present. If  $0 \leq \gamma \leq 1$  then the value iteration update is simply

$$V_{t+1}(s) = \min_{a \in A} \left\{ \operatorname{cost}(s, a) + \gamma \sum_{s' \in S} P_a(s, s') V_t(s') \right\}.$$

### 3.1 Value iteration for the iPod example

See Section 7 for the Python source code for value iteration on the iPod example. In this example we have  $\gamma = 1$ , so there is no discounting.

With  $N = 10$  songs, recognition cost of  $T = 0.5$ , and target song  $N/2$ , we get the following value and policy vectors:

```
>>> from ipod_mdp import *
>>> N = 10; value_iteration(N, 0.5, N/2)
([2.1984130546875003, 2.1984130546875003, 2.1984130546875003, 2.0,
1.0, 0.0, 1.0, 2.0, 2.1984130546875003, 2.1984130546875003],
['shuffle', 'shuffle', 'shuffle', 'sequential', 'sequential',
'sequential', 'sequential', 'sequential', 'shuffle', 'shuffle'], 2)
```

For example, if the initial state is 1, an execution may proceed as follows:

- $s = 1$ ,  $\pi(1) = \text{shuffle}$ . Jump to random song  $s = 8$ .
- $s = 8$ ,  $\pi(8) = \text{shuffle}$ . Jump to random song  $s = 3$ .
- $s = 3$ ,  $\pi(3) = \text{sequential}$ . Click  $5 - 3 = 2$  times to get to the target state 5.

The cost of this run is  $T + T + 2 = 0.5 + 0.5 + 2 = 3$ , while the value vector has  $V(1) = 2.1992065273437502$ , which is reasonably close.

A general execution with  $N = 250$  and  $T = 0.5$ :

```

$ python ipod_mdp.py 250 0.5
mean(V) = 10.6647967086
mean (simulation): 10.679298
difference: -0.0145012913545
shuffle when: 12 or more away

```

## 4 Sailing

Original reference: [6]. Imagine a sailing boat on a rectangular lake. Our goal is to get to the north-east corner of the lake from some starting position. To simplify things we assume that the lake is divided into a grid of *waypoints*, and that we sail from one waypoint to another (so the boat can only travel in one of eight directions north, north-east, east, etc). To further simplify the situation, we assume that the wind blows in only one direction for the duration of a journey from one waypoint to another waypoint. The wind changes just as we set off towards a new waypoint.

Directions will be the integers  $D = \{0, 1, \dots, 7\}$  where 0 is north, 1 is north-east, 2 is east, etc. A state  $s = (x, y, d, w_1, w_2)$  consists of a location  $(x, y) \in \mathbb{N} \times \mathbb{N}$ , the previous boat direction  $d \in D$ , previous wind direction  $w_1 \in D$ , next wind direction  $w_2 \in D$ . An action is the direction  $d \in D$  to sail the boat on the next leg. For each action  $a \in A$  and state  $s \in S$ , the transition probability

$$P_{a'}((x, y, d, w_1, w_2), (x', y', d', w'_1, w'_2)) = 0$$

unless  $w_2 = w'_1$ ,  $(x', y') = (x, y) + \vec{d}'$ , in which case the probability is  $P(w'_1, w'_2)$ , the probability that the wind changes from direction  $w'_1$  to  $w'_2$ .

If the boat is in direction  $d$  and the wind is in direction  $w$  then we have the following costs:

$$\text{cost}(d, w) = \begin{cases} 1 \text{ minute,} & \text{if } \alpha = 0 \\ 2 \text{ minutes,} & \text{if } \alpha = 1 \\ 3 \text{ minutes,} & \text{if } \alpha = 2 \\ 4 \text{ minutes,} & \text{if } \alpha = 3 \end{cases}$$

where  $\alpha = \min(d - w, 8 - (d - w))$ . We do not allow the case  $\alpha = 4$  which corresponds to sailing directly into the wind. The original description of the sailing problem also includes a cost for changing tack of the boat (i.e. when the wind goes from being on the left side of the sails to the right side or vice versa) but we ignore that cost to simplify the model.

The wind transition probabilities are given in an  $8 \times 8$  matrix  $W$ . The probability that the wind will be in direction  $w_2$  given that it is currently in direction  $w_1$  is  $W_{w_1, w_2}$ . In our simulations we fix  $W$  to the following matrix:

$$W = \begin{bmatrix} 0.4 & 0.3 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.3 \\ 0.4 & 0.3 & 0.3 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.4 & 0.3 & 0.3 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.4 & 0.3 & 0.3 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.4 & 0.2 & 0.4 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.3 & 0.3 & 0.4 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.3 & 0.3 & 0.4 \\ 0.4 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.3 & 0.3 \end{bmatrix}$$

We can now use value iteration to solve for the value and policy vectors.

## 4.1 Example of sailing

See Section 8 for Python code for value iteration on the sailing example. We run value iteration and 10,000 simulations with a lake of size  $5 \times 5$ , target location  $(4, 4)$ , and all other parameters as described above.

```
$ python sailing.py "value_iteration_example()"
Top of value_iteration(), max difference: 1.0
Top of value_iteration(), max difference: 4.0
Top of value_iteration(), max difference: 2.89
Top of value_iteration(), max difference: 2.2653
Top of value_iteration(), max difference: 1.935648
Top of value_iteration(), max difference: 1.66572639
Top of value_iteration(), max difference: 1.4497415721
Top of value_iteration(), max difference: 1.25418853297
Top of value_iteration(), max difference: 0.998954591375
Top of value_iteration(), max difference: 0.769139503728
Top of value_iteration(), max difference: 0.510518276155
Top of value_iteration(), max difference: 0.337118955482
Top of value_iteration(), max difference: 0.193325220838
Top of value_iteration(), max difference: 0.0315788407471
Done with value iteration
Running simulations...
```

Lake size: 5 x 5

Value iteration:

Mean cost: 7.3

Median cost: 7.1

Standard dev: 3.7

Simulations (run 1000 times):

Mean cost: 7.3

Median cost: 6.8

Standard dev: 4.1

Mean cost to sail across lake from (1, 1) to (4, 4): 8.5

## 5 UCT: Upper Confidence bounds on Trees

Solving an MDP system using value iteration can become computationally intensive on large examples (e.g. sailing on a lake of size  $40 \times 40$ ) because each update step necessarily reads and changes every element of the value and policy vectors.

An alternative approach is to use Monte Carlo planning. See Figure 2 for pseudo-Python. The algorithm alternates randomly between trying new actions at each state (in order to search for better policies) or using the current best policy to improve the estimate of the policy. The policy vector  $\pi$  is not explicitly computed. Instead we have the state-action vector  $Q(s, a)$ , the average seen cost (reward) of taking action  $a$  when in state  $s$ . The update line

```
new_average = old_average + (0.5)*(q - old_average)
```

has a bias of 0.5 instead of  $1/n$  and this may need to be tuned for each application of Monte Carlo planning. There are some subtleties to the implementation. See Section 10 for source code.

The UCT algorithm [4] just changes the way that actions are selected during a rollout. The algorithm used is called UCB, and described in the next section.

### 5.1 UCB: Upper Confidence Bounds

Imagine that we have  $K$  gambling machines with arbitrary reward distributions  $P_1, \dots, P_K$ . At each time step we can play any machine  $j$  that we choose, and receive a

reward according to the distribution  $P_j$ . We would like a strategy that maximises our total reward over  $n$  plays. Since the distributions  $P_j$  are fixed but unknown, we want to avoid sampling too many times from machines with low reward. In other words, we have to balance exploration versus exploitation.

The UCB1 strategy [2] is:

1. Play each machine once.
2. Play machine  $j$  that maximises  $\bar{x}_j + \sqrt{\frac{2 \ln n}{n_j}}$  where  $\bar{x}_j$  is the average reward from machine  $j$ , machine  $j$  has been played  $n_j$  times so far, and  $n$  is the total number of plays so far.

The  $\bar{x}_j$  term gives preference to machines that have performed well in past plays, while the  $\sqrt{2 \ln n / n_j}$  term gives preference to machines that have not been played many times so far, relative to  $\ln n$ .

Write  $\mu_j$  for the mean of distribution  $P_j$ . Obviously the best possible strategy is to only play the machine with the maximum mean  $\mu^* = \max_j \mu_j$ . Figure 3 shows best and actual reward for some simulations using the UCB1 strategy. See Section 9 for Python source code. The *regret* is the lost reward due to not having played the optimal machine at each step:

$$n\mu^* - \mu_j \sum_{j=1}^K \mathbb{E}(T_j(n))$$

where  $T_j(n)$  is the number of times that machine  $j$  has been played after  $n$  plays. Importantly, the regret can be bounded to be logarithmic in the number of plays so far:

**Theorem 5.1** ([2]). *For all  $K > 1$ , if policy UCB1 is run on  $K$  machines having arbitrary reward distributions  $P_1, \dots, P_K$  with support in  $[0, 1]$  then its expected regret after  $n$  plays is at most*

$$\left[ 8 \sum_{i: \mu_i < \mu^*} \left( \frac{\ln n}{\Delta_i} \right) \right] + \left( 1 + \frac{\pi^2}{3} \right) \left( \sum_{j=1}^K \Delta_j \right)$$

where  $\Delta_i = \mu^* - \mu_i$ .

Figure 4 shows the result of some simulations with  $K = 10$  machines and fixed reward distributions.

## 5.2 UCT

The UCT algorithm is the same as Monte Carlo planning except that UCB is used to select the action at each tree node. To do this we have to keep track of the average cost so far (i.e. up to time  $t$ ) of taking action  $a$  from state  $s$ , denoted  $Q_t(s, a)$ . We also track  $N_s(t)$ , the number of times that state  $s$  has been visited up to time  $t$ . If we are at a tree node  $s$  then the action is chosen using the UCB rule

$$a^* = \operatorname{argmax}_{a \in A} \left\{ Q_t(s, a) + \alpha \sqrt{\frac{\ln N_s(t)}{N_{s,a}(t)}} \right\}. \quad (1)$$

where  $\alpha$  is a constant that has to be chosen empirically. A badly chosen  $\alpha$  can have a huge effect on the convergence of the algorithm.

See Section 11 for Python source code that uses UCT to solve the sailing problem.<sup>1</sup>

## 5.3 UCT on the sailing problem

First we used value iteration to solve the sailing problem on lakes of size  $5 \times 5$ ,  $10 \times 10$ ,  $20 \times 20$ , and  $30 \times 30$ , with  $\epsilon = 0.01$  (solving just the  $20 \times 20$  instance took about 5 hours on a 2.5Ghz Intel Xeon server). The optimal value and policy vectors  $V^*$  and  $\pi^*$  are available in the pickled Python format as `lake_5.pkl`, `lake_10.pkl`, `lake_20.pkl`, and `lake_30.pkl`.

Next, we chose 20 random points (excluding the target point). For each of these states  $s$ , we ran the Monte Carlo and UCT algorithm until the estimated cost for sailing from  $s$  to the target state was within 0.1 of the optimal value  $V^*(s)$ . A good measure of the efficiency of a Monte Carlo type of planning algorithm is the total number of samples taken from the underlying MDP's generative model. Figure 5 shows the number of samples required to get the estimated error to within the bounds specified.

## 6 What else?

There are a few factors that have to be chosen by hand in `sailing_uct.py`. If we are working on a huge problem, where value iteration isn't feasible, how do we know

---

<sup>1</sup>I asked the authors of [4] for their full list of parameter values and source code but got no reply. If anyone improves my code I will be happy to update this document and give a reference or URL.

if we are converging to the optimal solution? The 0.1 factor for the  $Q$  value update step in Section 5 is meant to decay to zero if we want convergence. But at what rate? Using something like  $1/n$  works badly on the sailing domain with lake sizes up to  $20 \times 20$ .

Also, there are many ways to improve the UCT algorithm, for example combining online and offline data to speed up convergence of the  $Q$  values: [3].

```

def select_action(state):
    if random() < 0.01:
        return random_action(state)
    else:
        return best_action(state)

def search(state):
    if state == terminal: return 0

    action = select_action(state)
    new_state, cost = simulate_action(state, action)

    state_visit_counts[(state)] += 1
    state_action_counts[(state, action)] += 1

    try:
        old_average = Q[(state, action)]
        n = state_action_counts[(state, action)]
        new_average = old_average + (0.5)*(q - old_average)
    except KeyError:
        new_average = q

    self.Q[(state, action)] = new_average

    return q

def monte_carlo_planning(initial_state):
    while True:
        search(initial_state)

```

Figure 2: Pseudo-Python for Monte Carlo planning.

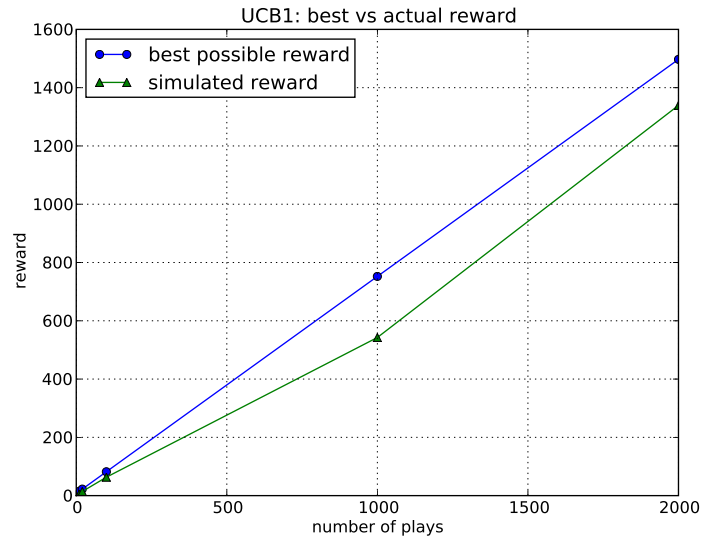


Figure 3: Comparing best and actual reward using the UCB1 strategy with 10 machines. See Section 9 for details on the machines' reward distributions.

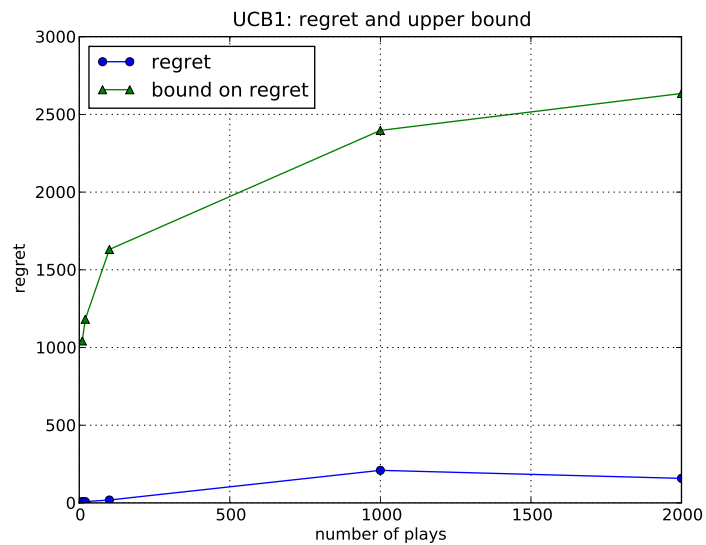


Figure 4: Regret and upper bound on regret given by Theorem 5.1

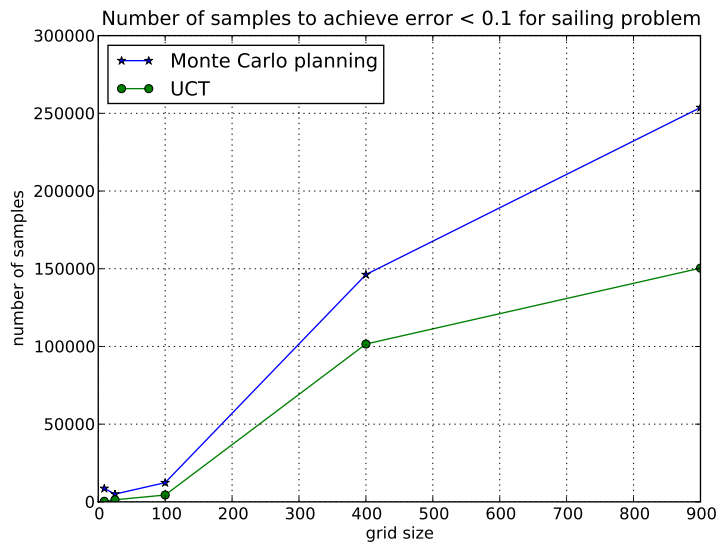


Figure 5: Performance of plain Monte Carlo planning and UCT on the sailing problem.

## 7 iPod value iteration

Here is `ipod_mdp.py`, also available at

<http://carlo-hamalainen.net/stuff/mdpnotes>

```
# -*- coding: utf-8 -*-

#####
#       Copyright (C) 2009 Carlo Hamalainen <carlo.hamalainen@gmail.com>,
#
#   Distributed under the terms of the GNU General Public License (GPL)
#
#   This code is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
#   General Public License for more details.
#
#   The full text of the GPL is available at:
#
#       http://www.gnu.org/licenses/
#####

"""
A Markov Decision Process (MDP) for the iPod problem
described at http://norvig.com/ipod.html
"""

import random
import sys

def value_iteration(N, T, target, epsilon = 0.001):
    # The possible actions from any state:
    actions = ['sequential', 'shuffle']

    # Transition probabilities:
    # transitions[s, a, w] = probability of moving from s to w by action a.

    transitions = {}
```

```

# Sequential strategy gets us to the target directly.
for s in range(N):
    for w in range(N):
        if w == target:
            transitions[s, 'sequential', w] = 1.0
        else:
            transitions[s, 'sequential', w] = 0.0

# The Shuffle strategy takes us to any state with equal
# probability.
for s in range(N):
    for w in range(N):
        transitions[s, 'shuffle', w] = 1.0/N

# Cost of each action from any state s.
cost = {}
for s in range(N): cost[s, 'sequential'] = abs(target - s)
for s in range(N): cost[s, 'shuffle'] = T

V1 = [0] * N
V2 = [1] * N
policy = ['shuffle'] * N

while max([abs(V1[i] - V2[i]) for i in range(N)]) > epsilon:
    for s in range(N):
        min_action = actions[0]
        min_action_cost = cost[s, actions[0]] \
            + sum([transitions[s, actions[0], w]*V1[w] for w in range(N)])

        for a in actions:
            this_cost = cost[s, a] + sum([transitions[s, a, w]*V1[w] \
                for w in range(N)])

            if this_cost < min_action_cost:
                min_action = a
                min_action_cost = this_cost

```

```

        V2[s] = min_action_cost
        policy[s] = min_action

    V1, V2 = V2, V1    # swapsies

try:
    p = min([s for s in range(N) if policy[s] == 'sequential']) - 1
    q = min([s for s in range(N) if V2[s] == target - s]) - 1

    # fixme: fails if epsilon is too large, ie. our policy vector isn't
    # optimal.
    # assert p == q

    p = min([s for s in range(N) if V2[s] == target - s]) - 1

    assert policy[p] == 'shuffle'
    assert V2[p] != target - p

    assert policy[p + 1] == 'sequential'
    assert V2[p + 1] == target - (p + 1)

    assert policy[p + 2] == 'sequential'
    assert V2[p + 2] == target - (p + 2)
except ValueError:
    # we must have come in with a high epsilon value and didn't get
    # a 'correct' value/policy vector.
    p = None

return V2, policy, p

def simulate(N, T, initial_state, target, policy):
    s = initial_state
    total_cost = 0

    while s != target:
        if policy[s] == 'sequential':
            total_cost += abs(target - s)
            s = target

```

```

        else: # policy[s] == 'shuffle'
            total_cost += T
            s = random.randrange(N)

    return total_cost

def average_simulation(N, T, policy):
    # The target state:
    target = N/2

    nr_iters = 1000*N

    return sum([simulate(N, T, random.randrange(N), target, policy) \
                for _ in range(nr_iters)]) / float(nr_iters)

def usage():
    print
    print "Usage:"
    print "$ python ipod_mdp.py <N> <T>"
    print
    sys.exit(0)

if __name__ == "__main__":
    if len(sys.argv) == 1: usage()

    if len(sys.argv) == 3:
        N = int(sys.argv[1])
        T = float(sys.argv[2])

        V, policy, p = value_iteration(N, T, N/2)

        away = N/2 - p

        assert policy[N/2 - (away - 1)] == 'sequential'
        assert policy[N/2 - (away)] == 'shuffle'
        assert policy[N/2 - (away + 1)] == 'shuffle'

```

```
mean_V = float(sum(V))/float(len(V))
mean_sim = average_simulation(N, T, policy)

print "mean(V) =", mean_V
print "mean (simulation):", mean_sim
print "difference:", mean_V - mean_sim

print "shuffle when:", away, "or more away"

sys.exit(0)

usage()
```

## 8 Value iteration for sailing

Here is `sailing.py`, also available at

<http://carlo-hamalainen.net/stuff/mdpnotes>

```
# -*- coding: utf-8 -*-

#*****
#       Copyright (C) 2009 Carlo Hamalainen <carlo.hamalainen@gmail.com>,
#
#   Distributed under the terms of the GNU General Public License (GPL)
#
#   This code is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
#   General Public License for more details.
#
#   The full text of the GPL is available at:
#
#       http://www.gnu.org/licenses/
#*****

# For debugging, put these lines somewhere to drop into an ipython shell:
#import IPython
#IPython.Shell.IPShell(user_ns=dict(globals(), **locals())).mainloop()

# To profile, put these in __main__():
#import cProfile
#cProfile.run('thing_to_run()')

import math
import pickle
import random
import sys

from scipy import arange, log, logn
from scipy.stats import rv_discrete
```

```

def mean(L): return sum(L)/(1.0*len(L))

def stddev(values, meanval=None):
    # copied from http://aima.cs.berkeley.edu/python/utils.html
    # and fixed the denominator.
    """The standard deviation of a set of values.
    Pass in the mean if you already know it."""
    if meanval == None: meanval = mean(values)
    return math.sqrt(sum([(x - meanval)**2 for x in values]) / (len(values)))

def median(values):
    # copied from http://aima.cs.berkeley.edu/python/utils.html
    """Return the middle value, when the values are sorted.
    If there are an odd number of elements, try to average the middle two.
    If they can't be averaged (e.g. they are strings), choose one at random.
    >>> median([10, 100, 11])
    11
    >>> median([1, 2, 3, 4])
    2.5
    """
    n = len(values)
    values = sorted(values)
    if n % 2 == 1:
        return values[n/2]
    else:
        middle2 = values[(n/2)-1:(n/2)+1]
        try:
            return mean(middle2)
        except TypeError:
            return random.choice(middle2)

def my_randint(n):
    """
    Return a random integer from [0, n).
    """

    return random.randint(0, n - 1)

```

```

def add_vector(x, y, v):
    """
    Returns (x + v[0], y + v[1]).

    EXAMPLES::

        >>> add_vector(0, 0, (0, 1))
        (0, 1)
        >>> add_vector(0, 0, (-2, 1))
        (-2, 1)
    """
    return (x + v[0], y + v[1])

def check_probability_matrix(P):
    """
    Rows must sum to 1 and no entry can be negative.

    EXAMPLE::

        >>> wind_array = [ [0.4, 0.3, 0.0, 0.0, 0.0, 0.0, 0.0, 0.3], \
                          [0.4, 0.3, 0.3, 0.0, 0.0, 0.0, 0.0, 0.0], \
                          [0.0, 0.4, 0.3, 0.3, 0.0, 0.0, 0.0, 0.0], \
                          [0.0, 0.0, 0.4, 0.3, 0.3, 0.0, 0.0, 0.0], \
                          [0.0, 0.0, 0.0, 0.4, 0.2, 0.4, 0.0, 0.0], \
                          [0.0, 0.0, 0.0, 0.0, 0.3, 0.3, 0.4, 0.0], \
                          [0.0, 0.0, 0.0, 0.0, 0.0, 0.3, 0.3, 0.4], \
                          [0.4, 0.0, 0.0, 0.0, 0.0, 0.0, 0.3, 0.3] ]
        >>> check_probability_matrix(wind_array)
        True
    """

    for x in P:
        if sum(x) != 1: return False

        for y in x:
            if y < 0: return False

```

```

    return True

def abs_direction_difference(d1, d2):
    """
    Absolute difference in directions.

    EXAMPLES::

        >>> abs_direction_difference(0, 1)
        1
        >>> abs_direction_difference(3, 2)
        1
        >>> abs_direction_difference(3, 5)
        2
        >>> abs_direction_difference(3, 7)
        4
    """

    assert d1 >= 0
    assert d1 < 8

    assert d2 >= 0
    assert d2 < 8

    x = abs(d1 - d2)

    if x < 8 - x:    return x
    else:            return 8 - x

def tack(boat_direction, wind_direction):
    """
    The tack of the boat depends on the relative difference of
    the boat's direction and the wind.

    EXAMPLES::

        >>> tack(0, 0)
        'away'

```

```

    >>> tack(0, 7)
    'down'
    >>> tack(0, 2)
    'cross'
    >>> tack(0, 3)
    'up'
    >>> tack(0, 4)
    'into'
    """

assert boat_direction >= 0
assert boat_direction < 8

assert wind_direction >= 0
assert wind_direction < 8

d = abs_direction_difference(boat_direction, wind_direction)

if d == 0: return 'away'
if d == 1: return 'down'
if d == 2: return 'cross'
if d == 3: return 'up'
if d == 4: return 'into'

raise ValueError

def wind_on_left(boat_dirn, wind_dirn):
    """
    Relative to the boat, is the wind blowing to the left of the boat?
    """

    # If the boat had been going north then we just need to check
    # if the wind direction is in [5, 6, 7]

    w = wind_dirn - boat_dirn

    while w < 0: w += 8

```

```

    return w in [5, 6, 7]

def direction_vector(d):
    """
    The direction 0 is north so we move by (0, 1) in cartesian
    coordinates.

    EXAMPLES::

        >>> direction_vector(0) # north
        (0, 1)
        >>> direction_vector(6) # west
        (-1, 0)
    """

    assert d >= 0
    assert d < 8

    if d == 0: return (0, 1)
    if d == 1: return (1, 1)
    if d == 2: return (1, 0)
    if d == 3: return (1, -1)
    if d == 4: return (0, -1)
    if d == 5: return (-1, -1)
    if d == 6: return (-1, 0)
    if d == 7: return (-1, 1)

class Sailing:
    def __init__(self, lake_size):
        self.gamma = 0.9 # discounting factor

        self.lake_size = lake_size

        # Wind transition probabilities.
        self.wind_array = [ \
            [0.4, 0.3, 0.0, 0.0, 0.0, 0.0, 0.0, 0.3], \
            [0.4, 0.3, 0.3, 0.0, 0.0, 0.0, 0.0, 0.0], \
            [0.0, 0.4, 0.3, 0.3, 0.0, 0.0, 0.0, 0.0], \

```

```

        [0.0, 0.0, 0.4, 0.3, 0.3, 0.0, 0.0, 0.0], \
        [0.0, 0.0, 0.0, 0.4, 0.2, 0.4, 0.0, 0.0], \
        [0.0, 0.0, 0.0, 0.0, 0.3, 0.3, 0.4, 0.0], \
        [0.0, 0.0, 0.0, 0.0, 0.0, 0.3, 0.3, 0.4], \
        [0.4, 0.0, 0.0, 0.0, 0.0, 0.0, 0.3, 0.3] \
    ]

    self.end_x = lake_size - 1
    self.end_y = lake_size - 1

    self.costs = { 'up':4, 'cross':3, 'down':2, 'away':1 }

    self.wind_distribution = []
    for i in range(len(self.wind_array)):
        vals = [arange(len(self.wind_array[i])), self.wind_array[i]]
        self.wind_distribution.append(rv_discrete(name='custm', \
            values=vals))

def states(self):
    """
    Instead of storing the states in a large list/dictionary, we
    provide an iterator.
    """

    for x in range(self.lake_size):
        for y in range(self.lake_size):
            for d in range(8):
                for w1 in range(8):
                    for w2 in range(8):
                        yield (x, y, d, w1, w2)

def is_terminal(self, state):
    return (state[0], state[1]) == (self.end_x, self.end_y)

def is_into(self, state, action):
    try:
        self.cost(state, action)
    except KeyError:

```

```

        return True

    return False

def random_state(self):
    w1 = my_randint(8)
    while True: # we weren't sailing into the wind...
        d = my_randint(8)
        if tack(d, w1) != 'into': break
    w2 = self.new_wind(w1)

    while True:
        state = (my_randint(self.lake_size), my_randint(self.lake_size), d, w1)
        if not self.is_terminal(state): break

    return state

def stays_in_lake(self, state, action):
    x, y, _, _, _ = state
    x2, y2 = add_vector(x, y, direction_vector(action))

    if x2 in range(self.lake_size) and y2 in range(self.lake_size):
        return True

    return False

def average_cost_of_transition(self, V, s, new_d):
    x, y, _, _, w2 = s

    x2, y2 = add_vector(x, y, direction_vector(new_d))

    if not self.stays_in_lake(s, new_d): return None

    this_cost = 0

    # We perform the local action
    try:

```

```

        this_cost += self.cost(s, new_d)
except KeyError:
    # don't sail into the wind...
    return None

for w3 in range(8):
    s_new = (x2, y2, new_d, w2, w3)

    this_cost += self.gamma*self.transition_probability(s, s_new)*V[s_new]

return this_cost

def transition_probability(self, s1, s2):
    """
    What is the probability of moving from state s1 to s2?

    s1 = (x1, y1, _, w1, w2)
    s2 = (x2, y2, d2, w2_2, w3)

    The boat was at (x1, y1) and travelled to (x2, y2). Then it must
    be the case that (x2, y2) = (x1, y1) + direction_vector(d2). If this
    does not hold then the probability is 0.

    If the previous wind direction of s2 does not match the new
    wind direction of s1 then the probability is 0 (so we need
    w2_2 == w2).

    Finally, the probability of moving from s1 to s2 is just the
    probability of the wind changing from w2=w2_2 to w3, which is
    stored in the global variable wind_array.

    >>> x1, y1 = 1, 1
    >>> x2, y2 = 2, 1
    >>> d1 = 0
    >>> d2 = 2 # the direction that we just travelled
    >>> w1 = 3
    >>> w2 = 2
    >>> w2_2 = w2

```

```

>>> lake_size = 5

>>> S = Sailing(lake_size)

>>> S.transition_probability((x1, y1, d1, w1, w2), \
                             (x2, y2, d2, w2_2, 1))
0.40000000000000002
>>> S.transition_probability((x1, y1, d1, w1, w2), \
                             (x2, y2, d2, w2_2, 0))
0.0
"""

x1, y1, _, w1, w2 = s1
x2, y2, d2, w2_2, w3 = s2

d_vec1, d_vec2 = direction_vector(d2)

# The boat was at (x1,y1) and travelled in
# direction d2 to arrive at (x2, y2).
if x2 != x1 + d_vec1: return 0
if y2 != y1 + d_vec2: return 0

# The new wind direction for s1 must be the
# previous wind direction of s2.
if w2 != w2_2: return 0

# Now we just have the probability of going from
# wind direction w2 to wind direction w3.

return self.wind_array[w2][w3]

def new_wind(self, w):
    """
    The wind is currently blowing in direction w and it changes to a
    new direction according to the matrix wind_array, which is encoded
    by general probability distribution in wind_probability_space.

```

*EXAMPLES::*

```
>>> lake_size = 5
>>> S = Sailing(lake_size)
>>> S.new_wind(0) in range(8)
True
>>> S.new_wind(4) in range(8)
True
"""

#assert w in range(8)

#return self.wind_distribution[w].get_random_element()
return self.wind_distribution[w].rvs()
```

```
def cost(self, s, d):
```

```
    """
```

*If we are in state s and we decide to travel in direction d, how much will this cost? Note that the wind for this new leg is in the last element of s.*

*EXAMPLES::*

```
>>> x1, y1 = 1, 1
>>> x2, y2 = 2, 1
>>> d1 = 0
>>> d2 = 2
>>> w1 = 3
>>> w2 = 2
>>> s = (x1, y1, d1, w1, w2)

>>> lake_size = 5
>>> S = Sailing(lake_size)
>>> S.cost(s, 0)
3
>>> S.cost(s, 1)
2
"""
```

```

new_wind = s[-1]

return self.costs[tack(d, new_wind)]

def best_action(self, s, V):
    """
    If we are in state s, use the value vector V to work out the
    best direction to travel in and its estimated cost.
    """

    x, y, d, w1, w2 = s

    # this is the end state
    if self.is_terminal(s): return (-1, 0) # (no action, terminal cost)

    # Otherwise we have to loop through all possible actions
    # and find the one with the minimum cost.

    min_d = None
    min_d_cost = None

    for new_d in range(8):
        new_d_cost = self.average_cost_of_transition(V, s, new_d)
        if new_d_cost == None: continue

        if min_d is None:
            min_d = new_d
            min_d_cost = new_d_cost
        elif new_d_cost < min_d_cost:
            min_d = new_d
            min_d_cost = new_d_cost

    return (min_d, min_d_cost)

def value_iteration(self, epsilon):
    V1 = {}
    V2 = {}
    policy = {}

```

```

for s in self.states():
    V1[s] = 0
    V2[s] = 10*epsilon
    policy[s] = -1

    if self.is_terminal(s):
        V1[s] = 0
        V2[s] = 0

while True:
    max_diff = max([abs(V1[i] - V2[i]) for i in self.states()])
    print "Top of value_iteration(), max difference:", max_diff
    sys.stdout.flush()

    if max_diff < epsilon: break

    for s in self.states():
        if self.is_terminal(s): continue

        policy[s], V2[s] = self.best_action(s, V1)

    V1, V2 = V2, V1

V = V2

V_avg = sum(V.values())/len(V)
V_stddev = stddev(V.values(), meanval = V_avg)

return V, policy, V_avg, V_stddev

def simulate(self, policy):
    w1 = my_randint(8)

    # boat not facing into the wind
    while True:
        d = my_randint(8)
        if tack(d, w1) != 'into': break

```

```

w2 = self.new_wind(w1)

current_state = (my_randint(self.lake_size), my_randint(self.lake_size), 0)

this_cost = 0
factor = 1.0

while not self.is_terminal(current_state):
    new_d = policy[current_state]

    this_cost += factor*self.cost(current_state, new_d)

    x2, y2 = add_vector(current_state[0], current_state[1], \
        direction_vector(new_d))
    w3 = self.new_wind(current_state[-1])
    current_state = x2, y2, new_d, current_state[-1], w3

    factor *= self.gamma

return this_cost

def run_simulations(self, policy, nr_sims):
    sims = []
    for i in range(1, nr_sims + 1):
        sims.append(self.simulate(policy))

    sims_avg = sum(sims)/len(sims)
    sims_stddev = stddev(sims, meanval = sims_avg)

    return sims, sims_avg, sims_stddev

def sample_next_state(self, state, action):
    """
    Use the generative model of S to find the next state given that
    we are in state and take action action.
    """

    if self.is_terminal(state):

```

```

        return None

    cost = self.cost(state, action)
    x2, y2 = add_vector(state[0], state[1], direction_vector(action))
    w3 = self.new_wind(state[-1])
    new_state = x2, y2, action, state[-1], w3

    return new_state, cost

def value_iteration_example():
    lake_size = 5

    S = Sailing(lake_size = lake_size)
    V, policy, V_avg, V_stddev = S.value_iteration(epsilon = 0.1)
    print "Done with value iteration"

    # Run a few thousand simulations:
    nr_sims = 1000
    print "Running simulations..."
    sims, sims_avg, sims_stddev = S.run_simulations(policy, nr_sims)

    print
    print "Lake size: %d x %d" % (lake_size, lake_size)
    print
    print "Value iteration:"
    print "    Mean cost: %.1f" % V_avg
    print "    Median cost: %.1f" % median(V.values())
    print "    Standard dev: %.1f" % V_stddev
    print
    print "Simulations (run %d times):" % nr_sims
    print "    Mean cost: %.1f" % sims_avg
    print "    Median cost: %.1f" % median(sims)
    print "    Standard dev: %.1f" % sims_stddev

    print
    v_11 = 0.0
    v_11_count = 0

```

```

for s in V.keys():
    if (s[0], s[1]) == (1, 1):
        v_11_count += 1
        v_11 += V[s]

print "Mean cost to sail across lake from (1, 1) to (%d, %d): %.1f" \
      % (S.end_x, S.end_y, (v_11/v_11_count))

def save_optimal_solution(lake_size):
    S = Sailing(lake_size)
    V, policy, V_avg, V_stddev = S.value_iteration(0.01)

    lake_filename = "lake_" + str(lake_size) + ".pkl"

    output = open(lake_filename, 'wb')
    pickle.dump(V, output)
    pickle.dump(policy, output)
    pickle.dump(V_avg, output)
    pickle.dump(V_stddev, output)
    output.close()

if __name__ == "__main__":
    if len(sys.argv) == 2:
        eval(sys.argv[1])
    else:
        value_iteration_example()

```

## 9 UCB

Here is `ucb.py`, also available at

<http://carlo-hamalainen.net/stuff/mdpnotes>

```
# -*- coding: utf-8 -*-

#####
#       Copyright (C) 2009 Carlo Hamalainen <carlo.hamalainen@gmail.com>,
#
#   Distributed under the terms of the GNU General Public License (GPL)
#
#   This code is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
#   General Public License for more details.
#
#   The full text of the GPL is available at:
#
#           http://www.gnu.org/licenses/
#####

import pylab
from scipy import arange, log, pi, sqrt
from scipy.stats import rv_discrete

def avg(L): return sum(L)/(1.0*len(L))

def make_scipy_rv(L):
    vals = [arange(len(L)), L]
    return rv_discrete(name='custm', values=vals)

"""
I want to have some number of machines, and the j-th machine has a spike
in the distribution at the j-th point.
"""
```

```

nr_machines = 10

machine_distributions = []
means = []
max_mean = None

for j in range(nr_machines):
    d = [1] * nr_machines
    d[j] = 20

    # normalise d
    d_sum = float(sum(d))
    d = [float(x/d_sum) for x in d]

    machine_distributions.append(make_scipy_rv(d))
    means.append(sum([x*d[x]/nr_machines for x in range(len(d))]))

max_mean = max(means)

def play_machine(k):
    """
    Play the k-th machine.
    """

    return machine_distributions[k].rvs()/(1.0*nr_machines)

def best_mu(): return max_mean

def mu(k):
    """
    Expected reward of machine k.
    """

    return means[k]

def run_ucb1(n):
    """

```

*Perform n plays using the UCB1 strategy.*  
"""

```
total_nr_plays = 0
nr_plays = [0] * nr_machines
average_reward = [0] * nr_machines

for j in range(nr_machines):
    average_reward[j] = play_machine(j)
    nr_plays[j] += 1

    total_nr_plays += 1

total_reward = 0

for _ in range(n):
    max_j = None
    max_xj = None

    for j in range(nr_machines):
        xj = float(average_reward[j] + \
                    sqrt(2.0*log(total_nr_plays)/nr_plays[j]))

        if max_j is None:
            max_j = j
            max_xj = xj
        elif xj > max_xj:
            max_j = j
            max_xj = xj

    reward = play_machine(max_j)
    total_reward += reward

    average_reward[max_j] = (nr_plays[j]*average_reward[max_j] + reward) \
                            /(nr_plays[j] + 1)

    nr_plays[j] += 1
    total_nr_plays += 1
```

```

# best possible reward, our reward, regret, upper bound on expected regret.
return (total_nr_plays*best_mu(),
        total_reward,
        total_nr_plays*best_mu() - total_reward,
        8*sum([float(log(total_nr_plays)/(best_mu() - mu(j))) \
                for j in range(nr_machines) if mu(j) < best_mu()]) \
            + float(1 + pi**2/3) + sum([best_mu() - mu(j) \
                                       for j in range(nr_machines)]))

runs = [(n, run_ucb1(n)) for n in [10, 20, 100, 1000, 2000]]

xrange = [x[0] for x in runs]

best_possible_data = [x[1][0] for x in runs]
total_reward_data = [x[1][1] for x in runs]
regret_data = [x[1][2] for x in runs]
regret_bound_data = [x[1][3] for x in runs]

pylab.plot(xrange, best_possible_data, '-o', \
            xrange, total_reward_data, '-^')
pylab.xlabel('number of plays')
pylab.ylabel('reward')
pylab.title('UCB1: best vs actual reward')
pylab.legend( ("best possible reward", "simulated reward"), loc='upper left')

pylab.grid(True)
pylab.savefig("best_and_total_reward.pdf")
pylab.close()

pylab.plot(xrange, regret_data, '-o', \
            xrange, regret_bound_data, '-^')
pylab.xlabel('number of plays')
pylab.ylabel('regret')
pylab.title('UCB1: regret and upper bound')
pylab.legend( ("regret", "bound on regret"), loc='upper left')

pylab.grid(True)
pylab.savefig("regret_and_bound.pdf")

```

## 10 Sailing MCT

Here is `sailing_mc.py`, also available at  
<http://carlo-hamalainen.net/stuff/mdpnotes>

```
# -*- coding: utf-8 -*-

#####
#       Copyright (C) 2009 Carlo Hamalainen <carlo.hamalainen@gmail.com>,
#
#   Distributed under the terms of the GNU General Public License (GPL)
#
#   This code is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
#   General Public License for more details.
#
#   The full text of the GPL is available at:
#
#       http://www.gnu.org/licenses/
#####

# For debugging, put these lines somewhere to drop into an ipython shell:
#import IPython
#IPython.Shell.IPShell(user_ns=dict(globals(), **locals())).mainloop()

# To profile, put these in __main__():
#import cProfile
#cProfile.run('thing_to_run()')

from sailing import *

from random import random
from scipy import log, sqrt
from incdict import IncDict

class SailingPlanner(Sailing):
```

```

def __init__(self, lake_size):
    Sailing.__init__(self, lake_size)

def random_action(self, state):
    possible_actions = []

    for action in range(8):
        if self.is_into(state, action): continue
        if not self.stays_in_lake(state, action): continue

        possible_actions.append(action)

    assert len(possible_actions) > 0

    return possible_actions[my_randint(len(possible_actions))]

def tree_policy(self, state):
    return self.best_Q_value(state)[0]

def select_action(self, state):
    if random() < 0.01:
        return self.random_action(state)

    action = self.tree_policy(state)
    if action is None: action = self.random_action(state)

    return action

def search_init(self, initial_state):
    self.nr_samples = 0

    self.initial_state = initial_state

    # keys: state
    # values: how many times we have visited this state during the
    # searches.
    self.state_visit_counts = IncDict()

```

```

# keys: (state, action) tuples
# values: how many times we have taken 'action' from 'state'
self.state_action_counts = IncDict()

# keys: (state, action) tuples
# values: average cost of taking 'action' from 'state'.
self.Q = {}

def search(self, state, depth = 0):
    if self.is_terminal(state): return 0

    action = self.select_action(state)
    new_state, cost = self.sample_next_state(state, action)

    if random() < 1.0/(self.state_visit_counts[(state)] + 1):
        try: q = cost + self.gamma*self.Q[(new_state, action)]
            except KeyError: q = cost + self.gamma*self.V_approx[new_state]
    else:
        q = cost + self.gamma*self.search(new_state, depth + 1)

    assert q != 0

    self.state_visit_counts[(state)] += 1
    self.nr_samples += 1
    self.state_action_counts[(state, action)] += 1

    try:
        old_average = self.Q[(state, action)]
        n = self.state_action_counts[(state, action)]
        #new_average = old_average + (1.0/n)*(q - old_average)
        new_average = old_average + (0.5)*(q - old_average)
    except KeyError:
        new_average = q

    self.Q[(state, action)] = new_average
    return q

def best_Q_value(self, state):

```

```

best_avg = None
best_action = None

for action in range(8):
    try:
        average_cost = self.Q[(state, action)]
    except KeyError:
        continue

    assert average_cost != 0

    if best_avg is None or average_cost < best_avg:
        best_avg = average_cost
        best_action = action

return best_action, best_avg

#####

def sailing_mc_planner(lake_size, V_optimal, V_approx, initial_state, max_nr_sampl
    S = SailingPlanner(lake_size)
    S.V_approx = V_approx

    S.search_init(initial_state)

    S.search(initial_state)

    optimal_cost = V_optimal[initial_state]

    while True:
        _, min_cost = S.best_Q_value(initial_state)

        error = abs(min_cost - optimal_cost)

        if error < 0.1: break

    q = S.search(initial_state)

```

```
    if S.nr_samples > max_nr_samples:  
        return None  
  
return S.nr_samples
```

## 11 UCT

Here is `sailing_uct.py`, also available at  
<http://carlo-hamalainen.net/stuff/mdpnotes>

```
# -*- coding: utf-8 -*-

#####
#       Copyright (C) 2009 Carlo Hamalainen <carlo.hamalainen@gmail.com>,
#
#   Distributed under the terms of the GNU General Public License (GPL)
#
#   This code is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
#   General Public License for more details.
#
#   The full text of the GPL is available at:
#
#       http://www.gnu.org/licenses/
#####

# For debugging, put these lines somewhere to drop into an ipython shell:
#import IPython
#IPython.Shell.IPShell(user_ns=dict(globals(), **locals())).mainloop()

# To profile, put these in __main__():
#import cProfile
#cProfile.run('thing_to_run()')

from sailing import *

from random import random
from scipy import log, sqrt
from incdict import IncDict
from sailing_mc import SailingPlanner
```

```

class SailingUCT(SailingPlanner):
    def random_action_of_untried(self, state):
        possible_actions = []

        for action in range(8):
            if self.is_into(state, action): continue
            if not self.stays_in_lake(state, action): continue
            if self.Q.has_key((state, action)): continue

            possible_actions.append(action)

        if len(possible_actions) == 0: return None

        return possible_actions[my_randint(len(possible_actions))]

    def select_action_uct(self, state):
        # If there is an untried action, give that a go.
        action = self.random_action_of_untried(state)
        if action is not None: return action

        uct_best = None
        uct_best_action = None

        for action in range(8):
            if not self.Q.has_key((state, action)): continue

            average_reward = -1.0*self.Q[(state, action)]

            assert average_reward != 0

            n_s_a = self.state_action_counts[(state, action)]
            n_s = self.state_visit_counts[(state)]

            uct_factor = 15.0

            this_val = average_reward + uct_factor*sqrt(log(n_s)/n_s_a)

            if this_val > uct_best:

```

```

        uct_best = this_val
        uct_best_action = action

    return uct_best_action

def tree_policy(self, state):
    return self.select_action_uct(state)

#####

def random_state(S):
    w1 = my_randint(8)
    while True: # we weren't sailing into the wind...
        d = my_randint(8)
        if tack(d, w1) != 'into': break
    w2 = S.new_wind(w1)

    while True:
        state = (my_randint(S.lake_size), my_randint(S.lake_size), d, w1, w2)
        if not S.is_terminal(state): break

    return state

def sailing_uct_planner(lake_size, V_optimal, V_approx, initial_state, max_nr_samp
S = SailingUCT(lake_size)
S.V_approx = V_approx

S.search_init(initial_state)
S.search(initial_state)

optimal_cost = V_optimal[initial_state]

while True:
    _, min_cost = S.best_Q_value(initial_state)

    error = abs(min_cost - optimal_cost)

    if error < 0.1: break

```

```
q = S.search(initial_state)

if S.nr_samples > max_nr_samples:
    return None

return S.nr_samples
```

## References

- [1] The convergence of a general value iteration process. <http://jmlr.csail.mit.edu/papers/volume3/szita02a/html/node21.html>. 3
- [2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256, May 2002. 5.1, 5.1
- [3] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 273–280, New York, NY, USA, 2007. ACM. 6
- [4] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006. 5, 1
- [5] Peter Norvig. Doing the martin shuffle (with your ipod). <http://norvig.com/ipod.html>. 2
- [6] Robert J. Vanderbei. Sailing strategies: An application involving stochastics, optimization, and statistics (sos). <http://www.orfe.princeton.edu/~rvdb/sail/sail.html>. 4