Assigned: Tuesday 29 November 2011

Due: Monday 5 December 2011, 11:59 pm

Objectives:

- Practice details of calculating optimal policies for Markov decision processes
- Learn how to build a world model through passive reinforcement learning
- Reinforce C programming skills

Collaboration: This homework assignment will be completed in pairs assigned by the instructor.

Introduction

We broaden our study of learning for Markov decision processes by cutting directly to the chase and determining optimal policies. We'll also let our agent operate on a fixed policy, bumbling around to build a model of the stochastic environment.

Background

Code

You will need `mdp.c` and `utilities.h` from the prior lab.

You may obtain the starter code for this assignment on the MathLAN from the directory

```
~weinman/courses/CSC261/code/adp
```

Here is an overview of the files it contains.

`utilities.o` An object file for calcultating expected and maximum expected utilities; implementation of the prior lab's assignment.

`policy_evaluation.h` A file containing declaration and documentation of Policy-Evaluation for estimating state utilities under a fixed policy.

`policy_iteration.c` A file containing a skeleton for an implementation of Policy-Iteration and a program for running the function on the command line with a particular environment (MDP), γ, and ϵ values.

`environment.h` A file declaring methods for simulating the results of taking actions in an MDP environment, as dictated by an implementation of `rl_agent_action` in a linked file.

`4x3.policy` A file containing the optimal policy of Figure 21.1(a).

Environment Simulation

When the agent has no model of the environment, it must learn one from experience. The functions in `environment.h` allow this. While you need not understand the function implementations, we will need to know how to use them. First, we set up an environment (which itself has/knows the world transition model)

```
environment_setup( mdpfile );
```

This call initializes a persistent variable containing the MDP (an `mdp` struct). To implement an agent, you'll need to provide the function

```
unsigned int rl_agent_action(unsigned int state, double reward);
```

which takes in a percept consisting of the current state and the reward signal (i.e., for having taken the previous action and arriving in the given state). The function returns an integer representing the action it desires to take. Once this function is implemented, the environment calls it, simulates the action requested by the agent, and reports the resultant state and reward reward in subsequent calls. We ask the environment to do this on our behalf with the call

```
environment_run( num_trials );
```

This repeatedly asks the agent for an action until a terminal state is reached (one trial). The agent is then returned to the initial state where the process repeats for the specified number of trials.

Assignment

Problem 1: Policy Evaluation

The text describes a modified policy iteration algorithm (p. 657) using a simplified Bellman update that relies on expected utility using the fixed action specified by a policy, rather than calculating the action having maximum expected utility. Such a policy evaluation algorithm, which is closely related to value iteration, is given below.

Algorithm 1 A policy evaluation algorithm using the simplified Bellman update that iterates value estimation until the utility change is sufficiently small.

function POLICY-EVALUATION(π, U, mdp, ϵ) **returns** a utility function
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
    rewards $R(s)$, discount γ
    π, a policy vector indexed by state
    U, a vector of utilities for states in S
    ϵ, change tolerance for utility updates
  local variables: δ, the maximum change in the utility of any state in an iteration
    U', a vector of utilities for states in S

  repeat
    $\delta \leftarrow 0$
    for each state s in S **do**
      if s.TERMINAL? **then**
        $U'[s] \leftarrow R(s)$
      else
        $U'[s] \leftarrow R(s) + \gamma \sum_{s'} P(s' \mid s, \pi[s]) U[s']$
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
    $U \leftarrow U'$
  until $\delta \leq \epsilon$
  return U

Implement the `policy_evaluation` function of `policy_evaluation.c`. using POLICY-EVALUATION as a guide. Some important things to note:

- You will need to declare, allocate, and ultimately free space for U', the updated array of utilities
- Make use of the `calc_eu` function
- Remember you can use `memcpy(3)` to copy larger chunks of memory (i.e., arrays)
- You can use `fabs(3)` to calculate absolute values

Problem 2: Policy Iteration

Implement the `policy_iteration` procedure in `policy_iteration.c`, as described in the POLICY-ITERATION pseudo-code of AIMA Figure 17.4 (p. 653). Some important things to note:

- You will need to declare, allocate, and ultimately free space for U, the array of utilities
- Initialize the utility of each state to its reward
- Do not attempt to calculate the policy for terminal states
- Make use of your `policy_evaluation`, `calc_meu`, and `calc_eu` functions
- `gdb(1)` is your friend

The `Makefile` you copied includes a command for building both the `policy_evaluation.o` object file as well as the `policy_iteration` program. The command

```
$ make policy
```

should build both.

Problem 3: Passive Reinforcement Learning

The file `adp.c` contains a set up that will run an agent without a world model in a sequential environment for a specified number of iterations. Implement the function `rl_agent_action` according to PASSIVE-ADP-AGENT of AIMA Figure 21.2 (p. 834).

At the top of the file you will notice several persistent variables that are initialized for you. To assist you, a summary of the conversions from the pseudo code to C statements are given below.

Pseudo-Code	C
current state s' and reward signal r'	parameters `state` and `reward`
s, a the previous state and action	`prevState`, `prevAction`
$s, a \leftarrow$ null	`prevValid = 0`
s' is new	`0 == visited[state]`
$U[s']$	`utilities[state]`
$R[s']$	`p_mdp->rewards[state]`
$N_{sa}[s, a]$	`state_action_freq[state][action]`
$N_{s'\|sa}[s', s, a]$	`outcome_freq[state][prevState][prevAction]`
$P(t \mid s, , a)$	`p_mdp->transitionProb[t][prevState][prevAction]`
$\pi[s']$	`policy[state]`
POLICY-EVALUATION(π,U,mdp)	`policy_evaluation(policy, p_mdp, epsilon, gamma, utilities)`

Even though an agent with no world model would not even know what or how many states the world has, we will overlook this issue because it makes dealing with array sizes much simpler. All the environment properties remain available to your agent (via `p_mdp`) except for the rewards and the transition proability, which are allocated but have no meaningful data. Therefore, your loop "**for each** t" can be over all states (but you must still check whether the outcome frequency for the state is nonzero).

Problem 4: Application

Part A: Policy Iteration

Test Test your policy iteration using the `4x3.mdp` grid world. With sufficiently small tolerance (ϵ), and sufficiently high discount factor, your answers should be identical to those reported in AIMA Figure 21.1.

Predict Before running your policy iteration on the larger 16×4 grid world, you should record *qualitatively* what your expectations are for the policy, given your experience with the 4×3 grid world and the utilities for this environment from the prior lab. Record your answers before proceeding.

Experiment Run your `policy_iteration` on the 16×4 grid world with the same discount factor and tolerance you used for 4×3 grid world. As in the prior lab, use whatever format and/or means you prefer to add the policy of each state to the graphic.

Reflect Reflect on the values you see. Where and how were your predictions confirmed? Where and how were they contradicted? What surprised you?

Part B: Adaptive Dynamic Programming

Test Test your ADP implementation using the (presumably optimal) policy learned by policy iteration. For example,

```
$ ./policy_iteration gamma epsilon 4x3.mdp > 4x3.policy
$ ./adp gamma epsilon 4x3.mdp trials < 4x3.policy
```

How many trials does it take before your utilities match those of Figure 21.1(b) to at least two decimals?

Predict Before running your ADP implementation on the larger 16×4 world, record how many trials do you expect your agent to require to learn a transition model that is accurate enough to estimate state utilities accurately? Why?

Experiment

Run your `adp` on the 16×4 grid world with the same discount factor and tolerance you used for the 4×3 grid world. How many trials does it take for the learned utilities to match those from your prior lab as closely as possible?

Reflect

Compare your predictions to your experimental outcome. Speculate about the cause of any discrepancies.

What to turn in

Your submission should include the following

- Your completed `policy_evaluation.c`, `policy_iteration.c and adp.c`
- A single PDF containing (merged)
  - Your C files (Note that your `enscript` command should use the flag `-Ec` rather than `-Escheme`)
  - A transcript of your `policy_iteration` and `adp` programs' compilations and output for both grid worlds
  - An essay tying together the tests, predictions, experiments, and reflections of Problem 4.