Assigned: Tuesday 22 November 2011

Due: Monday 28 November 2011, 11:59 pm

Objectives:

- Practice details of calculating state utilities for Markov decision processes
- Reinforce C programming skills

Collaboration: This homework assignment will be completed in pairs of your choice.

Introduction

How should an agent act in a fully observable, stochastic, sequential environment? In this assignment we will implement the answer to that question by determining the utility of states in a small domain. This work will ground future approaches where we take the shortcut of identifying the best action directly, as well as learning state utilities, and policies, and even transition models.

Background

Code

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

```
~weinman/courses/CSC261/code/mdp
```

Here is an overview of the files it contains.

`mdp.c` A file (with header) containing a Markov decision process (MDP) struct and some related functions.

`utilities.h` A file containing declarations for two procedures, one for calculating the expected utility taking an action in a state of an MDP, and another that returns the action giving maximal expected utility in a state of an MDP. You will implement these two procedures.

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

`4x3.mdp` A file containing a text representation of the grid world described in AIMA Figure 17.1 that can be read with `mdp_read`.

`16x4.mdp` A file containing a text representation of the grid world described in Figure 1 below.

The MDP Struct

We first need an overview of what the MDP struct contains because it is integral to all our learning agent's activities. All your accesses to the `mdp` type will be via pointers. Thus variables like `p_mdp` represent pointers to the contents of struct, which is reproduced here:

```
typedef struct {
  unsigned int numStates;  /* Discrete total number of possible states */
  unsigned int numActions; /* Discrete total number of possible actions */
  unsigned int start;      /* Value of starting state */
  double ***transitionProb;/* A numStates x numStates x numActions array of
                              transition probabilities for the model world:
                              transitionProb[t][s][a] := P(t|s,a) */
  unsigned int *numAvailableActions; /* A numStates length array, each
                                        entry indicating the number of
                                        actions available in the given state */
  unsigned int **actions;  /* A numStates x numAvailableActions[state] length
                              array of the actions available in a given state */
  double *rewards;         /* A numStates length array of the reward
                              for a given state */
  unsigned int *terminal;  /* A numStates length array, each entry indicating
                              whether a given state is terminal */
} mdp;
```

Many of the struct elements are pointers themselves because they refer to arrays. However, because `p_mdp->numStates` tells us how many entries are in `p_mdp->rewards`, `p_mdp->terminal`, and even the first two indices of `p_mdp->transitionProb`, we can access these via array indexing. For example, to get the reward of state 8, we might write

```
p_mdp->rewards[8]
```

and to determine whether state 0 is terminal we might write

```
if ( p_mdp->terminal[0] ) { printf("No escape! It's terminal.\n"}
```

and so on. Because different states may have different numbers of actions available in them, we must represent the number of available actions, and the array of available actions separately. To get a, the first action available from state 8 (assuming there is one), we would use

```
unsigned int a = p_mdp->actions[8][0];
```

To get the transition probability $P(s' = 0 \mid s = 8, a)$ we could then write

```
double prob = p_mdp->transitionProb[0][8][a];
```

Assignment

Problem 1: Utilities

Because they crop up again and again, we will first write some "utility" functions for calculating the expected utility of a state.

Part A

Implement `calc_eu` in `utilities.c`, as described by the `utilities.h` header file. The returned value should represent

$$EU(a \mid s) = \sum_{s'} P(s' \mid s, a) U(s')$$

where $P(s' \mid s, a)$ is given by the `transitionProb` element of `p_mdp`, $U(s)$ is given by the `utilities` array, the current state s is given by the `state` parameter, and the action a is given by the `action` parameter. Note that the sum is over all states in the MDP (`numStates`).

Part B

Implement `calc_meu` in `utilities.c` as described by the `utilities.h` header file. The values `meu` and `action` are passed by reference, and these will end up containing the results of the action giving the highest expected utility (`action`) and that maximal expected utility itself (`meu`).

For the given `state` and `utilities`, you will need to loop over the available actions keeping track of which action gave the highest expected utility and what it was.

Caution: Remember that looping over the actual available actions themselves requires some care.

Problem 2: Value Iteration

Implement the `value_iteration` function in `value_iteration.c`, as described in the VALUE-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 updated array of utilities
- Be sure to initialize all utilities to zero
- Remember you can use `memcpy(3)` to copy larger chunks of memory (i.e., arrays)
- Make use of your `calc_meu` function
- You can use `fabs(3)` to calculate absolute values
- `gdb(1)` is your friend

One other important note is worth mentioning. The pseudo-code does not mention what should happen with terminal states. Rather than use the maximum expected utility with the discount factor, terminal states should simply assign their utility to be the state's reward.

The `Makefile` you copied includes commands for building both the `mdp.o` and `utilities.o` object files as well as the `value_iteration` program. The command

```
$ make value
```

should build all three.

Problem 3: Application

Part A

Test your value iteration using the `4x3.mdp` grid world. Depending on your allowable error (ϵ), and discount factor, your answers should be nearly identical to those reported in Figure 17.3. Keep in mind the convergence criterion for value iteration.

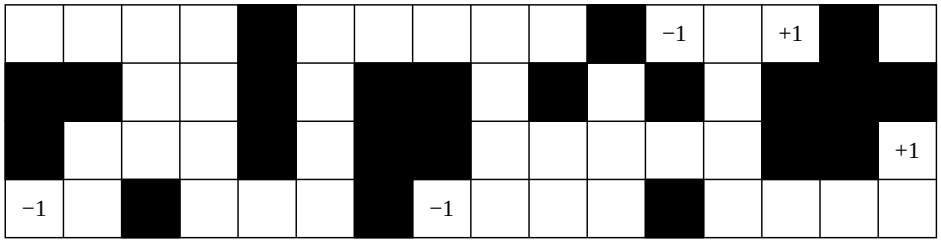

Figure 1: A 16×4 grid world, adapted from AIMA Figure 4.18, with the value of terminal states shown. The start state is in the upper-left corner. The reward for other states is -0.04 and the probability of moving forward is 0.8 or to either side 0.1 (when possible), just as in the original 4×3 environment.

Part B

Consider the grid world of Figure 1. Before running your value iteration on this grid world, you should record *qualitatively* what your expectations are for the utilities of the states, given your experience with the 4×3 grid world. That is, what will the relative values of the utilities be? Consider the path lengths to various terminal states. What do you predict the policy will be? Record your answers to these before proceeding.

Part C

Run your `value_iteration` with the same discount factor and error tolerance you used for 4×3 grid world. Various graphical formats of the grid world are in the starter directory from the introduction (`rl-maze.*`). Using whatever format and/or means you prefer, add the utility of each state (to three significant digits) to the graphic.

Part D

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

What to turn in

Your submission should include the following

- Your completed `value_iteration.c and utilities.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 `value_iteration` program's compilation and output for both grid worlds
  - A short essay tying together parts B,C, and D of Problem 3.