Assigned: Tuesday 26 November 2013

Due: Monday 2 December 2013, 11:59 pm

Objectives:

- **Reflect** on Markov decision process behavior
- Attend to **details** of the value iteration algorithm
- **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 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.

`utilities.o` A pre-compiled implementation of two procedures you may optionally implement for extra credit.

`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.

`value_iteration` A pre-compiled implementation including Value-Iteration you may optionally use for analysis.

`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 Markov Decision Process Structure

We first need an overview of what the Markov decision process (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 action = 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][action];
```

Assignment

In addition to those listed below, 12 points will be allotted for general good programming practice.

Problem 1: Value Iteration [20 points]

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 need to declare, **`malloc(3)`**, and ultimately **`free(3)`** space for U', the updated array of utilities (consult `main` for an example)

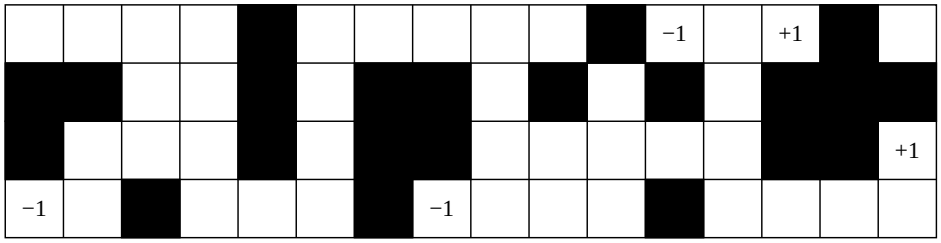

Figure 1: A 16×4 grid world, adapted from AIMA Figure 4.18, with the value of terminal states shown. The start state is in the upper-left corner. The reward for other states is -0.04 and the probability of moving forward is 0.8 or to either side 0.1 (when possible), just as in the original 4×3 environment.

- 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 the **`calc_meu`** function declared in `utilities.h` by compiling against `utilities.o`
- You can use **`fabs(3)`** to calculate absolute values
- **`gdb(1)`** is your friend

Note: The book's 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` object file as well as the `value_iteration` program with `utilities.o`. The command

```
$ make value
```

should build both.

Problem 2: Application [20 points]

A complete, compiled version of `value_iteration` is included with the original files copied from the MathLAN; you may use it or your own version for experimentation.

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.

Part B

Consider the grid world of Figure 1. Before running value iteration on this grid world, record *qualitatively* what your expectations are for the utilities of the states, given your experience with the 4×3 grid world. 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 predictions before running the algorithm.

Part C

Run `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?

Problem 3: Calculating Utilities [Extra Credit: 16 points]

Write the "utility" functions for calculating the expected utility of a state.

Part A [4 points]

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 [12 points]

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 (see Lab Exercise B: "Understanding actions").

The `Makefile` includes commands for building the `utilities.o` object file. The command

```
$ make value
```

should build everything.

What to turn in

Your submission should include the following

- Your completed `value_iteration.c`
- Your completed `utilities.c` (if submitting extra credit)
- A single PDF containing (merged)
  - Your C files (Note that your `enscript` command should use the flag `-Ec` rather than `-Escheme`)
  - A single transcript of **your** `value_iteration` program's (and `utilities.c` if submitting extra credit) compilation and output for both grid worlds
  - Answers to parts B,C, and D of Problem 2.

Submissions missing the PDF or files in any other formats will not be graded. C files lacking transcripts will not be graded.