CSC 261 **Lab 11: Markov Decision Processes** **Spring 2020**

Assigned: Tuesday 21 April

Due: Monday 27 April, 10:30 pm

Objectives:

- Attend to **details** of the value iteration and policy iteration algorithms
- **Reflect** on Markov decision process behavior
- **Reinforce** C programming skills

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

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 and policy of states in a small domain with the value iteration and policy iteration algorithms.

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.

`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.

`policy_iteration` A pre-compiled implementation including Policy-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 function `mdp_read(...)`.

`16x4.mdp` A file containing a text representation of the grid world described in Figure 1 on page 4.

The Markov Decision Process Structure

The Markov decision process (MDP) `struct` below is integral to all our learning agent's activities. All accesses to the `mdp` type will be via pointers, so variables like `p_mdp` represent pointers to the contents of the structure:

```
typedef struct {
  unsigned int numStates;  /* Total number of possible states */
  unsigned int numActions; /* Total number of possible actions */
  unsigned int start;      /* Starting state for this MDP */
  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, where each
                                        entry indicates 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 */
  bool *terminal;          /* A numStates length array, each entry indicating
                              whether a given state is terminal */
} mdp;
```

Many struct elements are pointers 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, 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) and the transition probability $P(s' = 0 \mid s = 8, a)$ we would use

```
unsigned int action = p_mdp->actions[8][0];
      double prob   = p_mdp->transitionProb[0][8][action];
```

Assignment

Implementations [40 points]

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:

- Declare, **`malloc`**`(3)` or **`calloc`**`(3)`, and ultimately **`free`**`(3)` space for U', the updated array of utilities (consult `main` for an example)
- Initialize all utilities to zero
- Use **`memcpy`**`(3)` to copy larger chunks of memory (i.e., arrays)
- Use **`calc_meu`** declared in `utilities.h` by linking with `utilities.o`
- Use **`fabs`**`(3)` to calculate absolute values
- **`gdb`**`(1)` and AddressSanitizer[1] are your friends

[1] Compile and link in `clang` with the `-g` and `-fsanitize=address` flag. See https://clang.llvm.org/docs/AddressSanitizer.html.

Important 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 shell command **`make value`** should build both.

Problem 2: Policy Evaluation [8 points]

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, closely related to value iteration, is 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:

- Declare, allocate, and ultimately free space for U'
- Use the **`calc_eu`** function declared in `utilities.h` by linking with `utilities.o`
- **`gdb`**`(1)` and AddressSanitizer still want to be your friends

Problem 3: Policy Iteration [12 points]

Implement the `policy_iteration` procedure in `policy_iteration.c`, as described in the POLICY-ITERATION pseudo-code of AIMA Figure 17.7 (p. 657). Some important things to note:

- Declare, allocate, and ultimately free space for U
- 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`** and the **`calc_meu`** and **`calc_eu`** functions
- **`gdb`**`(1)` is desperate and lonely; please call it

The `Makefile` you copied includes a command for building both the `policy_evaluation.o` object file as well as the `policy_iteration` program. The shell command **`make policy`** should build both.

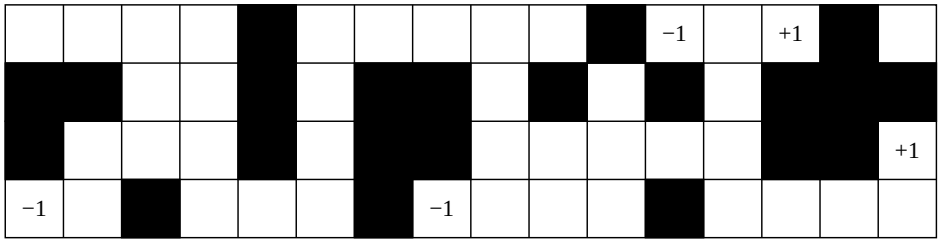

Figure 1: A 16×4 grid world, adapted from AIMA Figure 4.18, with the value of terminal states shown. The start state is in the upper-left corner. The reward for other states is -0.04 and the probability of moving forward is 0.8 or to either side 0.1 (when possible), just as in the original 4×3 environment.

Analysis [32 points]

Complete, compiled versions of `value_iteration` and `policy_iteration` are included with the original files copied from the MathLAN; you may use them (with acknowledgment) or your own versions for experimentation. In any case, your report should cite which program provides your data.

Problem 5: Value Iteration [16 points]

Test 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.

Predict 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.

Experiment 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 for visualization.

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

Problem 6: Policy Iteration [16 points]

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

Predict Before running policy iteration on the larger 16×4 grid world, 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 value iteration. 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. For visualization, add the policy of each state to the grid world graphic. (*Note:* The ODS file converts action numbers to arrows for you.)

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

Extra Credit [16 points]

Problem 7: Calculating Utilities

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 utilities value
```

should build everything. Note that doing so will overwrite the `utilities.o` object provided for you, so if your implementation is wrong and you recompile and link `value_iteration` or `policy_iteration`, they too will be incorrect.

What to turn in

Your submission should include the following

- Your completed `value_iteration.c`, `policy_evaluation.c`, and `policy_iteration.c`
- [Optional] Your completed `utilities.c` (if submitting extra credit)
- A single-session transcript `value.txt` of **your** `value_iteration` program's (and `utilities.c` if submitting extra credit) compilation and output for *both* grid worlds
- A single-session transcript `policy.txt` of **your** `policy_evaluation.o` object and `policy_iteration` programs' compilations and output for *both* grid worlds
- A single PDF file `analysis.pdf` containing answers to *Predict*, *Experiment* (including value and policy figures), and *Reflect* of Problems 5 and 6.

  Implementations lacking transcripts will receive a zero.
  Analyses in any other format other than PDF will receive a zero.

Copyright ©2013, 2015, 2018 Jerod Weinman. This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 4.0 International License.