

Assigned: Tuesday 1 December

Due: Friday 11 December, 5:00 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.

`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.

`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 on page 4.

The Markov Decision Process Structure

The Markov decision process (MDP) struct 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; /* 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 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 $P(s' = 0 | s = 8, a)$ 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)`, 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 compiling against `utilities.o`
- 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 following command should build both:

```
$ make mdp value
```

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 $P(s' | s, a)$,
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
       $U'[s] \leftarrow R(s) + \gamma \sum_{s'} P(s' | s, \pi[s]) U[s']$ 
    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 compiling against `utilities.o`
- `gdb` (1) still wants to be your friend

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 17.4 (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 following command should build both:

```
$ make policy
```


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

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

where $P(s' | s, a)$ 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.

What to turn in

Your submission should include the following

- Your completed `value_iteration.c`, `policy_evaluation.c`, and `policy_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-session transcript of **your** `value_iteration` program’s (and `utilities.c` if submitting extra credit) compilation and output for both grid worlds
 - A single-session transcript of **your** `policy_evaluation.o` object and `policy_iteration` programs’ compilations and output for both grid worlds
 - Answers to *Predict*, *Experiment*, and *Reflect* of Problems 5 and 6.

A missing PDF or files in any other format will receive a zero.

C files lacking transcripts will receive a zero.

