CSCI–4150 Introduction to Artificial Intelligence, Fall 2004
Assignment 7 addendum

A. Problems

Here are the problems for this assignment with the correction to the procedure names for problems 2 and 3, a bit more explanation for some problems, and details for problem 5.

1. (15 points) Write the procedures: `calc-initial-state` and `calc-new-state` that transform the game state into a reinforcement learning state; `terminal-state?` that indicates whether a reinforcement learning state is a terminal state; and `init-tables` that initializes the tables for transition probabilities, rewards, and utility values.

   The details for these procedures appear in the original Assignment 7 handout.

2. (15 points) Write the procedure `(basic-rl-strategy fs-num actions)` that takes a nonnegative integer `fs-num` and a list `actions` that are permissible. It should return the action that maximizes expected utility. Essentially, your procedure should implement the following equation:

   $$a = \text{argmax}_{a_i \in A} \sum_{s'} T(s, a_i, s')U(s')$$

   Note: your `basic-rl-strategy` procedure should be able to work with any valid solution for Problem 1.

3. (15 points) Write the procedure `(create-exploring-rl-strategy R+ Ne)`. It should return a strategy procedure that incorporates the simple exploration function described in our text (page 774) where `Ne` and `R+` are parameters in the exploration function. This procedure can do some initialization, for example to initialize nonterminal utilities to random values, to zero, or to `R+`.

   Note that Equation 21.5 in the text computes optimistic values in terms of optimistic utility values. However, this equation (as the text notes) is given as an example to use with value iteration, not temporal differencing. You can implement your optimistic utility calculation using the real utility values on the right hand side, even though this is not correct (as explained on page 774). Because of the relatively short action sequences for this problem domain, it should work OK for large enough values of `Ne`. You might also try initializing all nonterminal utility values to `R+` in addition to implementing this exploration function.

   Note: the returned strategy procedure should be able to work with any valid solution for Problem 1.

4. (15 points) Write the procedure `(create-td-learning alpha)` that returns a procedure of the form `(td-learning fs-num action ts-num)`. This `td-learning` procedure will be called for each state transition and should update the utilities according to the temporal differencing update equation:

   $$U(s) \leftarrow U(s) + \alpha(R(s) + U(s') - U(s))$$

   Note that this is equation 21.3 from our text, but I have omitted the discount factor γ and changed the notation from $U^{\pi}(s)$ to $U(s)$ since we do not have a fixed policy π here.

   Note: the returned learning procedure should be able to work with any valid solution for Problem 1.

5. (75 points) For this problem, you will use your code to learn a blackjack player and evaluate its performance. You will write up (and turn in) the results.

   (a) Describe how your `calc-X-state` procedures (this means the `calc-initial-state` and `calc-new-state` procedures) turn the game state into a reinforcement learning state. I am interested, not in the details of how you do the calculation, but in how the reinforcement learning states you use correspond to game states. Make sure you say how many states you used, and which are terminal states.

Give a brief explanation why you transformed the game state to a reinforcement learning state this way.

(b) Play blackjack using your implementation of the procedures from Problem 1 to "learn" transition probabilities and average rewards for your states. One way to do this is to run the random player on a large number of hands. (I would say at least 10,000 hands, but if you have many states, you will need to play more hands.) You should verify that your transition probabilities and average rewards have converged by checking that the values do not change (or do not change much) with additional hands.

Save the resulting tables and upload them to the web tester for this problem.

In your writeup, briefly describe how you learned these transition probabilities and average rewards (e.g., which player, how many hands played, and how you verified that your values had converged). Examine the state transition probabilities for a few states. Are they correct? Assume that you are equally likely to receive any card.

(c) Load your tables from the previous part and use a temporal differencing learning player (i.e., a strategy procedure created by your `create-exploring-rl-strategy` procedure, and a learning procedure created by your `create-td-learning` procedure) to learn utilities for non-terminal states as the agent explores the state space.

You will have to pick values for the parameters required by the `create-exploring-rl-strategy` and `create-td-learning` procedures. You should turn off table updates for this part. Also note that the initial utility values will be whatever your original call to `init-tables` set them to be, unless you do some initialization in your `create-exploring-rl-strategy` procedure.

You should repeatedly play "rounds" of some number of hands and check to see whether the utility values have stopped changing (actually, whether the maximum change is less than some small amount). You should write a little code to do this: a procedure that creates a list of utilities (using the `get-utility-element` procedure), one that compares two lists of utilities to find the amount of the maximum change, and one that repeatedly plays "rounds." The last procedure should terminate when the utility value change is small enough or until some maximum number of rounds have been played. You may even want to let the `alpha` value decrease as more rounds are played.

Save the resulting tables. Add any code that you wrote for this part to this file and upload it to the web-tester for this problem.

In your writeup, describe what you did to learn the utility values for nonterminal states. Make sure you say what parameter values you chose (i.e., `R+`, `Ne`, and `alpha`), and how many hands were played in each round. Briefly explain your choice of parameters. Also, how many rounds did it take for your utility values to converge?

(d) Load the tables from the previous part and test how well the following player performs.

```
(define (rl-player)
  (list "Alice" basic-rl-strategy non-learning-procedure))
```

Play 10,000 hands, and report the net winnings and total bets. Calculate the winnings as a percentage of total bets.

Implement another strategy of your choice. You can easily find blackjack strategies online, but you will probably have to modify them because you we aren't allowing splitting. Play 10,000 hands and report the results as above.

Describe the strategy you implemented, and include your code in your writeup. Chances are that your reinforcement learning player did not do as well as your manually programmed strategy. Why is this, and how would you learn a better (or optimal) strategy for your states?

B. Support code documentation

B.1 Players and strategies

A *player* is a list of three things, in order:

1. A (double-quoted) string containing the name of the player. You can use any name except `"dealer"`.
2. A *strategy procedure* of the form `(lambda (fs-num actions) ...)` where `fs-num` is a reinforcement learning state (a nonnegative integer) returned by one of your `calc-X-state` procedures, and `actions` is a list of actions that your player may take at that point in the game.
3. A *learning procedure* of the form `(lambda (fs a ts) ...)`. This procedure will be called on when there is a state transition from state `fs` after taking action `a`to state `ts`.

To get started, you can use the random player (defined in the `a7example.scm` file):

```
(define (random-strategy state-num actions)
  (list-ref actions (random (length actions))))

(define (non-learning-procedure fs a ts)
  '())

(define (random-player)
  (list "Bob" random-strategy non-learning-procedure))
```

I strongly suggest that you define players as procedures so that the most recent version of your strategy and learning procedures is used.

B.2 Playing blackjack

There are two procedures for playing blackjack with your player:

```
  (play-hand player)
  (play-match N player)
```

The first plays a single hand; the second plays `N` hands. The initial bet on each hand is 1.0. If the player doubles down, then the bet is increased to 2.0. The `play-match` procedure returns a list of two numbers: the first is your player's net winnings, and the second is the total amount your player bet. Note that for a fixed number of hands, the amount bet will vary if your player doubles down.

Note that you must have initialized the tables (see Section C of the original Assignment 7 handout) and have defined the `calc-X-state` procedures (Problem 1) in order to play blackjack — even if your player does not use them (as is the case for the random player). Again, you can use the procedures in `a7example.scm` to get started.

The operation of `play-hand` and `play-match` can be controlled by the following variables:

- `print-narration` — when `#t` (its default value) a narration of the play is printed to the screen. You can disable printing parts of this information with the following variables (all of whose default value is `#t`):
  - `print-player-play`
  - `print-dealer-play`
  - `print-bet-settlement`

  When `print-narration` is set to `#f`, no narration is printed to the screen, with one exception, controlled by the following variable:
  - `print-match-progress` — when set to, for example, 10 (its default value), a message will be printed every 10 hands. If you don't want this message to be printed, you can set it to `#f`.
- `print-learning` — when `#t` (its default value), a message will be printed before each call to your learning procedure.
- `enable-table-updates` — when `#t` (its default value), the support code records all state transitions and rewards in order to build up data for estimating state transition probabilities and average rewards. Disabling table updates will preserve the current transition probabilities and average rewards.

While testing, you may find that you want to run your code on the same sequence of cards/hands. In order to do this, I have provided procedures that let you manipulate the state of the random number generator.

- `(save-random-state` — makes and saves a copy of the random number generator state
- `(restore-random-state` — restores the random number generator state to that from the last call to `save-random-state`. The sequence of random numbers (and therefore sequence of cards) will be the same as following the last call to `save-random-state`.

You will notice that if the dealer or the player has blackjack, there is no update to the tables, nor is the player's learning function called. This is because there is no state transition in these situations.

B.3 Tables

The main procedures for creating, accessing, and printing the tables of transition probabilities, rewards, and states were described in the original Assignment 7 handout. What follows are additional features (and a few clarifications) of the support code.

- As noted in the original handout, a call to `create-tables` will discard the old tables. However, you should know that loading `a7code.com` will also discard the old tables.
- There is a procedure `print-rl` that takes zero arguments and prints all the tables by calling `print-transitions`, `print-rewards`, and `print-utilities`.
- There are several variables that control how the tables are printed to the screen:
  - `print-line-width` — default value is 80
  - `transition-decimal-places` — default value is 3
  - `utility-decimal-places` — default value is 3
  - `reward-decimal-places` — default value is 3
- Make sure your code works for any number of states. You can get the number of states by calling the procedure `(num-states)`.

- The support code counts the number of "action transitions," i.e., the number of times a given action is tried from each state. If you ran the random player lots of times to learn the transition probabilities and rewards, all its actions are recorded in this manner. However, when you first start a temporal differencing program, you want it to start exploring the states without any knowledge of what happened previously. This is necessary for your exploration function to work. To reset the action transition counts, call the procedure:
  - `(reset-action-transitions)` — all action transition counts (as returned by the procedure `get-action-transitions`) will be zeroed by a call to this procedure.
- To save your tables, you can call the procedure:
  - `(save-tables fname)` — given a double-quoted string argument, it will print definitions of the transition, utility, and rewards tables to a file of that name.

B.4 Blackjack hands

There are several procedures for evaluating cards blackjack hands that you can use:

- `(bj-value h)` — returns the value of a hand
- `(soft-hand?  h)` — returns `#t` if there is an ace is the hand that is counted as 11.
- `(blackjack?  h)` — returns `#t` if `h` is a two card hand with value 21.