

## Problems

1. (30 points, E) Write the procedure (`rl-strategy game-state actions`) which uses the utilities to pick one of the actions (a list containing the symbols `hit`, `stand`, and sometimes `double-down`). This should be able to work with anyone's `transform-state` procedure given the transition probabilities, rewards, and utilities. You can use the example in `a7example.txt` to test your procedure.
2. (20 points, E/W) Write the procedure (`transform-state game-state`) procedure as described in the original assignment handout (bottom of page 2). You should devise your own method for transforming the game state into a state number — think about the problem and what states would be useful to a reinforcement learner. You must also write the associated (`init-tables`) procedure that will initialize the tables for transition probabilities, rewards, and utilities to the proper size.

For the written part of this problem, turn in a clear and detailed but concise description of how you change the game state into a numerical state for reinforcement learning. Explain your reasoning behind this transformation. Be sure to specify how many states it uses.

3. (40 points, E/W) Use some reinforcement learning method to learn utilities for the states under your `transform-state` procedure. See the discussion later in this handout for more details. Turn in the code for whichever approach you choose.

For the written part of this problem, describe the method you implemented and any significant details of your implementation. Also describe the testing/verification you did while developing your implementation.

4. (30 points, E/W) Learn transitions, rewards, and utilities using your reinforcement learning method. Save the tables to a file (a procedure in the support code will be provided for this), and add your `transform-state` and `init-tables` procedures to this file. The result should be something like the `a7example.scm` file. We should be able to measure the performance of your learned strategy by loading this file and using our own utility-based player (without any exploration function). Part of your score on this problem will be based on the performance of your learned strategy.

In the written portion of this problem, describe the steps you took to learn this player, including details such as the number of hands played in various phases of the learning, how much was lost (or won) during the learning, and approximately how much time it took.

## The blackjack simulator

The support code will simulate playing blackjack with a player. A *player* is a list where the first element is the name of the player, the second is its *strategy* procedure, and the third is its *learning procedure*. For example, you can use the following player to get started:

```
(define (random-player)
  (list "Bob" random-strategy (lambda (fs a ts) '())))
```

The `random-strategy` simply picks one of the actions randomly, without looking at the game state; you can replace this with your `rl-strategy` once you have learned utility values for it to

use. Here, the learning procedure here is just a `lambda` procedure that does nothing. You could later replace this with a procedure that does reinforcement learning through temporal differencing.

Before you can play blackjack, you must have a `transform-state` procedure and you must have initialized the tables. (Details on these things appear later in this handout.) You would normally do this by calling your `init-tables` procedure.

To get started, you can use the example in the file `a7example.scm`. Do the following:

```
(load "a7header")
(load "a7example")
(play-hand (random-player))
```

A call to `init-tables` is not necessary here because the `a7example.scm` file loads a set of tables.

If you want to play more than one hand, you can do the following:

```
(play-match 100 (random-player))
```

This will play 100 hands. Right now, these procedure print out a narration of every hand. The support code will soon include variables for you to turn this off.

Note that these procedure return the net result of the player's betting — this is useful for evaluating how good your player is.

## Blackjack states

Your `transform-state` procedure needs to translate the game state into a nonnegative integer. (See the original assignment handout for details and some examples.)

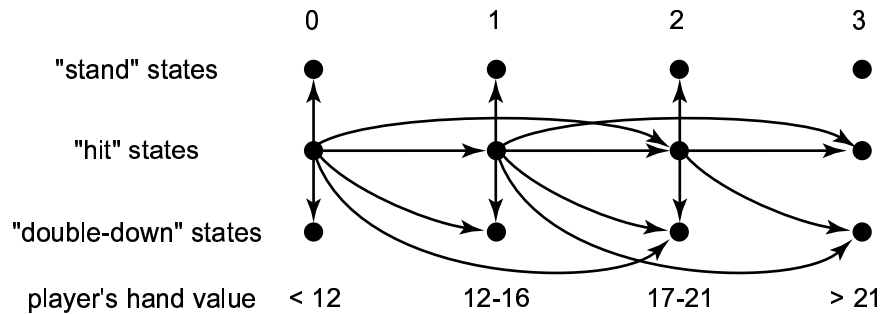
There's something funny about blackjack states, at least for the way I have structured this assignment. The "hit" action is normal — after taking this action, the game state changes, and the player has a choice of actions from this new state. Sometimes, the "hit" action leads to a terminal state, i.e. when the player "busts."

However, the "stand" action does not lead to a change in the game state — the cards in the player's hand do not change, and the dealer's face-up card doesn't change. However, after a player "stands," there is something that is different — the player can then receive a reward depending on actual value of the dealer's hand. Essentially, the "stand" action transforms the state from a nonterminal state to a terminal state, even though the game state (as defined for this assignment) is the same.

The "double-down" action is similar. The player receives an additional card, which changes the game state, but this is a terminal state. If the player had received the same card by taking a "hit," the player could still take another action ("hit" or "stand"). Unlike the "stand" action, the "double-down" action can lead to a number of different terminal game states, depending on what card is received.

The way we will handle this situation is, if your `transform-state` procedure classifies the game state into one of  $n$  states, we will actually create  $3n$  states. For each state number  $j$ , there will be one terminal "stand" state, one terminal "double-down" state, and one "hit" state. Note that one "hit" state (or more, depending on how you classify the game state) will be a terminal state.

For example, suppose we classify the game state into one of 4 states according to whether the value of the player's hand is less than 12, between 12 and 16, between 16 and 21, and over 21. The `transform-state` procedure would return an integer between 0 and 3, but there would actually be the following 12 states:



“Hit” actions lead from one “hit” state to another, “stand” actions lead from a “hit” state to a “stand” state, and “double-down” actions lead from a “hit” state to a “double-down” state. Directed arcs are shown for all transitions with nonzero probability.

Although your reinforcement learning implementation will not encode the rules of blackjack, you will use your understanding of the game to devise reinforcement learning states that make sense for the game. As one example, you should not have any state that represents hands that have busted and hands that have not. Also, since the valid actions will be given to the `r1-strategy` procedure to select from, you do not need to worry about whether some action, `double-down` for example, is valid in all hands represented by a state.

## Transition probabilities, rewards, and utilities

The support code, while playing blackjack, will keep track of all observed state transitions and rewards in order to build up a model of blackjack for your states. To initialize the tables for storing this information, your `(init-tables)` procedure must call:

```
(create-tables n-states . utility-initialization)
```

where `n-states` is the number of states that your `transform-state` procedure will produce; the actual state numbers are 0 through `(- n-states 1)`. This procedure takes one optional argument for initializing the “hit” state utilities. If given, this argument must be a procedure of zero arguments; it will be called repeatedly to initialize each utility value. If not given, the utilities will be initialized to zero.

By calling this procedure again (or by calling your `init-tables` procedure again), the old tables will be thrown away, and new empty tables will be created.

### Transition probabilities

The transition probabilities describe the probability of going from state to state when a given action is executed. In our text’s notation, these are the  $T(s, a, s')$ . They are estimated by simply dividing the number of state transitions from  $s$  to  $s'$  when action  $a$  is executed by the total number of times action  $a$  was taken from state  $s$ .

The following procedures are available to access the transition probabilities:

- `(print-transitions)` — prints the transition table to the screen
- `(get-transition-element fs-num action ts-num)` — returns the value of transitioning from “hit” state `fs-num` (the “from state” number) to the “action” state `ts-num` (the “to state” number) under action, which must be one of the symbols: `hit`, `stand`, or `double-down`.

- `(get-transition-vector fs-num action)` — returns a list (not a Scheme vector) of all the transition probabilities from state `fs-num` under `action`. The first element is the probability of transitioning to state 0, the second, to state 1, and so on.

## Rewards

The support code also keeps a running average of what rewards are received in every state. Note that, in general, the actual reward is random because it depends the value of the dealer’s hand. The following procedures are available to access the rewards:

- `(print-rewards)` — prints a table of the reward values to the screen
- `(get-reward state-num action)` — returns the average reward received in the “action” state `statenum`.

## Utilities

Since the “stand” states and the “double down” states are terminal states, we do not need to keep a utility for these states. Recall that utility is “expected reward to go,” so for a terminal state, we only need to look at the reward received for reaching that state.

Therefore, there is only storage for utilities for the “hit” states. Note that at least one of the “hit” states will be a terminal state. However, the support code cannot know how many or which one(s) a priori, so it has storage for a utility value for all “hit” states.

The following procedures are available to access the rewards:

- `(print-utilities)` — prints a table of the utility values to the screen
- `(get-utility-element state-num)` — returns the utility of the “hit” state `state-num`
- `(get-utility-vector)` — returns a list (not a Scheme vector) of all the “hit” states where the first element corresponds to state 0, the second to state 1, etc.
- `(set-utility-element state-num u)` — changes the utility value for “hit” state `state-num` to `u`.

## A utility-based strategy for blackjack

Once a model of the world is known, i.e. the transition probabilities and rewards, and the utility values have been calculated, a utility-based strategy for blackjack (which you will implement in your `rl-strategy` procedure) simply needs to calculate its action:

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

This is essentially equation 17.4 in our text. This simply says that in any given state  $s$ , the policy should pick the action that has the highest expected utility.

For the `rl-strategy` procedure that you turn in, you should calculate all the expected utilities and select an action accordingly. Since we will test your `rl-strategy` procedure with other utility values, rewards, and transition probabilities, you cannot hardcode your own values. (You also can’t hardcode your own `transform-state` procedure for this reason.) For your own testing and development, you could calculate a policy once (from your final utility values) and simply put that policy in a “strategy” procedure. You will probably want to do this if you are implementing reinforcement learning with policy iteration.

Calculating the expected utility for the “hit” action follows the above equation. However, the way this assignment is set up, we do not have utilities for the terminal “stand” and “double-down” states, so expected utility for “stand” and “double-down” actions must be calculated as follows:

$$\sum_{s'} T(s, a, s') R_a(s')$$

This is then the expected reward for taking action  $a$  since we know this action leads to a terminal state. Note that the the reward  $R_a(s')$  is a function of the action  $a$ , reflecting the fact that for a given state number  $s'$ , we need to use the rewards appropriate to that action.

## Reinforcement learning

For this assignment, you must implement some form of reinforcement learning. You basically have three choices. Although in the original handout I suggested temporal differencing as a default, I now think that value iteration may be simpler. I haven’t yet implemented all of them, so I can’t say for sure. You’ll have to make your own decision!

Whichever approach you take, you should test your learned player to see how well it does and whether you can improve it further.

### Value and policy iteration

You can learn a model of blackjack by playing a lot of games on the simulator using the random strategy and a learning function that doesn’t do anything. This will give you the transition probabilities and the average rewards for all the states. You can then apply either the value iteration algorithm or the policy iteration algorithm (described in Chapter 17 of the text) in order to learn the utility values.

You may need to test the utility values by simulating additional games using your utility values with your `rl-strategy` procedure. You might find that you need to repeat this process — as you put a better strategy in place, the probabilities and rewards in the more commonly used parts of the state space become more accurate, and this may necessitate a new policy.

If you choose one of these approaches, implement one of the following procedures:

```
(value-iteration)
```

```
(policy-iteration)
```

When called, your procedure should be able to learn utility values for any set of states, i.e. under anyone else’s `transform-state` procedure, assuming that the model has already been learned. (Of course, you might set parameters, such as termination conditions, that work well only on your reinforcement learning states. However, your procedure should be able to run on any number of states.

If you choose to implement policy iteration, you can try to find code that will solve linear systems of equations, invert matrices, etc. This is an exception to the general rule that students should not receive code from any other source for an assignment.

### Temporal differencing

Another approach to reinforcement learning is to use temporal differencing, described in Chapter 21 of the text. The basic idea is that utilities are updated for every observed state transition. You will have to pick a learning rate  $\alpha$  and decrease its value as the learning proceeds.

In order to implement temporal differencing, you will need to write a learning procedure. To understand where this procedure is called (and what its arguments are), I suggest using a procedure like the following:

```
(define (learning-procedure fs-num action ts-num)
  (print "Learning on transition under: " action "\n"
        "      from: " from-state "\n"
        "      to: " to-state "\n"))
```

A normal call to the learning procedure indicates that there was a transition from the “hit” state `from-state` to the “action” state `ts-num`. When a terminal state is reached, this procedure will be called with a null `to-state`. The terminal state in this case will be the “action” state `from-state`. Note that the action refers to the `to-state` in the former case and to the `from-state` in the latter.

You will need to write a `rl-strategy` that uses an *exploration function*, described in Chapter 21 of the text. It is this function that allows reinforcement learning to explore the state space instead of repeatedly following the first path it finds to some positive reward.

If you use temporal differencing, implement a learning procedure:

```
(td-learning fs-num action ts-num)
```

and an exploration function:

```
(exploration-fn u n)
```

A strategy that uses the utilities to make decisions must be used in conjunction with temporal differencing. You can do this by making a copy of the `rl-strategy` and modifying it to use the exploration function.

You will need to know how many times a state has been visited in order to use the exploration function. You can get this information using the procedure:

- `(get-visits-element state-num action)` — returns the number of times the “action” state `state-num` has been visited.

The advantage of temporal differencing (well, technically any active reinforcement learning method), is that it should be more efficient at learning the game. For value iteration and policy iteration, you would need to play for a while to learn a model. For active reinforcement learning, this exploratory play would be directed towards parts of the model that provide greater rewards. In this manner, the utilities are also refined in areas of greatest use to the player; states that are not so useful will be explored initially to see if there is anything good there, but otherwise seldom, if at all, visited after that.