CSCI–4150 Introduction to Artificial Intelligence, Fall 2004
Assignment 7 (135 points), out Monday November 22, due Thursday December 9

Learning to play blackjack

In this assignment, you will implement a reinforcement learning method that learns to play blackjack. Your program will be somewhat generic in that it will not have the rules of blackjack built-in — it will learn how to play as it goes along! This assignment is structured only in that you will need to compute the utilities of states. The support code provides the "performance element" that will actually play blackjack, making decisions on what action to take based upon the state, observed transition probabilities, and your learned utilities.

A. Rules of blackjack and Scheme representations

Blackjack is card game commonly found in casinos. A number of players may be seated at a a blackjack table, but players are each playing against the dealer, not each other. The basic objective is to have a "hand" with a value higher than the dealer's, but not over 21. The players can choose how they play their hands, but the dealer's strategy for playing his or her hard is fixed.

A.1 Terminology

- **cards** — a standard deck of playing cards is used, i.e., there are four suits (clubs, diamonds, spades, and hearts) and 13 different cards within each suit (the numbers 2 through 10, jack, queen, king, and ace)

  We will be using a *shoe* of 4 decks. This means that four complete decks of cards are shuffled together and placed in the shoe. Cards are drawn from this shoe. The shoe will be reset after approximately 3 decks of cards have been played. (The shoe is reset in between hands, not during a hand.)

- **card values** — the numbered cards (2 through 10) count as their numerical value. The jack, queen, and king count as 10, and the ace may count as either 1 or 11.

- **hand value** — the value of a hand is the sum of the values of all cards in the hand. The values of the aces in a hand are such that they produce the highest value that is 21 or under (if possible). A hand where any ace is counted as 11 is called a *soft* hand. The suits of the cards do not matter in blackjack.

- **blackjack** is a two-card hand where one card is an ace and the other card is any value 10 card.

A.2 Rules of play

There are some slight variations on the rules and procedure of blackjack. Below is the simplified procedure that we will use for this assignment — we will not be using "insurance" or "splitting" which are options in the standard casino game.

1. Each player places a bet on the hand.

2. The dealer deals two cards to each player, including him- or herself. The players' cards will be face-up. One of the dealer's cards is face-up, but the other is face-down.

3. The dealer checks his or her face-down card. If the dealer has blackjack, then the dealer wins the bets with all players unless a player also has blackjack. If this happens, this is called a *push*, meaning that the player and dealer have tied, and the player keeps his/her bet.

4. If a player has blackjack (but the dealer does not), then that player wins immediately. The player is paid 1.5 times his or her bet.

5. Each of the remaining players, in turn, is given the opportunity to receive additional cards. The player must either:

   - *hit* — the player receives one additional card (face up). A player can receive as many cards as he or she wants, but if the value of the player's hand exceeds 21, the player *busts* and loses the bet on this hand.
   - *stand* — the player does not want any additional cards
   - *double-down* — before the player has received any additional cards, he or she may double-down. This means that the player doubles his or her bet on the hand and will receive only one additional card. The disadvantage of doubling-down is that the player cannot receive any more cards (beyond the one additional card); the advantage is that the player can use this option to increase the bet when conditions are favorable.

6. The dealer turns over his or her face-down card. The dealer then "hits" or "stands" according to the following policy:

   - If the value of the hand is less than 17, the dealer must "hit."
   - If the hand is a "soft 17," the dealer must "hit."
   - Otherwise, the dealer must "stand."

   Most casinos force the dealer to hit a soft 17, but there is some variation in this regard.

   If the dealer busts, then he or she loses the bets with all remaining players (i.e., those players that did not bust or have blackjack).

7. The dealer then settles the bets with the remaining players. If the dealer has a hand with a higher value than the player, then the player loses his or her bet. If the values are equal, then it is a "push" and the player keeps his or her bet. If the player has a hand with a higher value, then the dealer pays the player an amount equal to the player's bet.

In this assignment, there will only be one player; if the player busts, then the dealer's hand will not be played.

A.3 Scheme representation

- A cards is represented by a list of two elements:
  - the first element is either an integer between `2` and `10` or one of the symbols: `jack`, `queen`, `king`, and `ace`.
  - the second element is one of the symbols: `diamonds`, `spades`, `clubs`, and `hearts`.
- A hand is represented by a list of cards.
  - A player's hand will always be a list with at least two elements. The first two elements will always be the cards that you were initially dealt, i.e., any additional cards you receive will be appended to the end of the list.
  - A dealer's hand will consist of a list of one card (the face-up card) before the dealer has played his or her hand. After the dealer's hand is played, it will be a list of all the dealer's cards; the first element will still be the dealer's original face-up card.

B. Reinforcement learning states

A reinforcement learning method will calculate utility values for states. You will need to write two procedures to transform the "game state" (i.e., the dealer's and player's cards) into a "reinforcement learning state" (represented by a nonnegative integer). You will have to decide how to do this transformation. I will give a simple example below.

The support code will provide procedures for storing utility values for reinforcement learning states as well as for keeping track of transition probabilities between reinforcement learning states.

> **Problem 1** (15 points) Write the following 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.

Here are details on each of these procedures:

- `(calc-initial-state player-hand dealer-hand)`

  The argument `player-hand` will consist of a list of two cards that the player was initially dealt; `dealer-hand` will consist of a list of one card, the dealer's face-up card. This procedure must return a valid reinforcement learning state (a nonnegative integer between zero, inclusive, and the number of states declared to `create-tables`, exclusive).

- `(calc-new-state previous-rl-state action terminal? reward player-hand dealer-hand)`

  The arguments are as follows.

  - `previous-rl-state` will be a nonnegative integer (returned by either your `calc-initial-state` procedure or by the previous call to your `calc-new-state` procedure).
  - `action` will be one of the symbols `hit`, `stand`, or `double-down`
  - `terminal?` will be `#t` or `#f` depending on whether the action resulted in a terminal state or not. The `stand` and `double-down` actions always result in a terminal state. The `hit` action may or may not result in a terminal state (depending on whether you bust or not).
  - `reward` is the reward for the new state. For nonterminal states, the reward will always be zero, but for terminal states, this will be the amount that you win or lose. (Note that this might be 0 if there is a "push."
  - `player-hand` will be the player's hand *after* the action is taken. For the `hit` and `double-down` actions, there will be a new card added to the end of this list. For the `stand` action, there will be no change.
  - `dealer-hand` will be the dealer's hand after the action is taken. For the `stand` and `double-down` action, the dealer's hand will have been played, so this argument will contain all the dealer's cards. For the `hit` action, the dealer's hand will consist of only the face-up card, even if the player has busted.

  It must return a valid reinforcement learning state.

- `(terminal-state? rl-state)`

  This procedure should return `#t` if `rl-state` (a nonnegative integer) is a terminal state and returns `#f` otherwise.

- `(init-tables)`

  For now, the main thing to know is that you will have to declare how many reinforcement learning states you have. Details of the transition probability and utility tables appear in later sections.

C. Transition probabilities

The support code will keep track of all observed state transitions and rewards in order to build up a model of blackjack for your states. It will also provide storage for the utility values for nonterminal states.

To initialize the tables for storing this information, your `(init-tables)` procedure must call:

```
(create-tables num-states . utility-init-proc)
```

where `num-states` is the number of states you will use; the state numbers are 0 through (`num-states` - 1). The optional argument `utility-init-proc` must be a procedure that takes zero arguments. This procedure will be called to initialize the utility value for every nonterminal state (as indicated by your `terminal-state?` procedure). If not given, the utilities will be initialized to zero.

Each time this procedure is called, it will create new, empty tables; any information in the old tables will be lost.

Transitions

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 in the support code to access the transition probabilities:

- `(print-transitions)` — prints the transition table to the screen
- `(get-transition-probability fs-num action ts-num)` — returns the probability of transitioning from state `fs-num` to state `ts-num` under `action`. `fs-num` and `ts-num` are reinforcement learning states (nonnegative integers), and `action` is one of the symbols `hit`, `stand`, or `double-down`.
- `(get-transition-alist fs-num action)` — returns an association list of all transitions from state `fs-num` under `action`. The result is a list of the form:

  ```
  ((ts-num1 tprob1) (ts-num2 tprob2) ...)
  ```

  where the `ts-num`'s are states for which there was an observed transition, and the `tprob`'s are the fraction of times (i.e., probability) of that transition.
- `(get-action-transitions fs-num action)` — returns the number of times that `action` has been taken from state `fs-num`

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 in the support code to access the rewards:

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

Utilities

For nonterminal states, your reinforcement learning algorithm will compute the utility values. For terminal states, the utility will be the average reward received in that state.

The following procedures are available in the support code to access the utilities:

- `(print-utilities)` — prints a table of the utility values to the screen
- `(get-utility-element state-num)` — returns the utility of the reinforcement learning state `state-num`
- `(set-utility-element state-num u)` — changes the utility value for the state `state-num` to `u`.

D. Reinforcement learning

If the utility values are known, then the best action is the one that maximizes the expected utility.

> **Problem 2** (15 points) Write the procedure `(basic-rl-player 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:
>
> $$a = \text{argmax}_{a_i \in A} \sum_{s'} T(s, a_i, s') U(s')$$

However, we are interested in learning the utility values as our agent plays the game. To do this, you will implement temporal differencing. There are two parts:

> **Problem 3** (15 points) Write the procedure `(create-exploring-rl-player R+ Ne)` that returns a player procedure that incorporates the simple exploration function described in our text (page 774) where `Ne` and `R+` are parameters in the exploration function.

> **Problem 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.

E. Conclusion

Additional documentation and some examples will be on the assignment web page. And as usual, there will be a writeup. You will find that this assignment does not require that much code, and so the writeup and your final blackjack player are worth most of the points for this assignment.

> **Problem 5** (75 points) I will ask you to try learning a blackjack player with different parameters for an exploration function and for temporal differencing. You will turn in your final player (support code will be provided that lets you save the transition tables to a file), and write up details of your implementation such as how you decided to transform the game state to reinforcement learning states and what parameters you used for your final player, how many hands it played while learning, etc. Details will be available on the assignment web page.