Monte Carlo Tree Search with Clojure

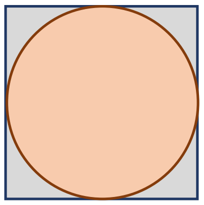
James Vanderhyde
Chicago Clojure Meetup, July 26, 2017

In March 2016, Google's parent company Alphabet made history when its Go-playing AI AlphaGo beat the world's best human Go player. At the heart of that AI is a technique called Monte Carlo tree search. We will use Clojure to explore this technique and apply it to a simpler game, Shut the Box. We will start with Monte Carlo simulation, and if time permits we will move on to Monte Carlo tree search.

Part 1: Monte Carlo Simulation with Clojure

Task 1—Monte Carlo simulation

1. Open the example.clj file. Run the program and take a look at what the code does.
2. The Clojure code shows how to do a Monte Carlo simulation for winning a simple game of flipping a coin: Flip a coin many times, count how many flips are wins, and calculate the probability of winning.
3. Let's do something a little more interesting: a simplified game of darts. We will simulate throwing darts at a circular dart board. Some darts hit inside the circle, and some hit outside.



4. Replicate the structure of the coin-tossing game to make the dart-throwing game.
5. Step 1: Define the game.
    a. To throw a dart, use the *rand* function twice to generate a random x coordinate and a random y coordinate.
    b. The *rand* function returns a number between 0 and 1, but the dart game uses numbers between -1 and 1. To get the correct range, we have to scale and shift the result of *rand*. Multiply the result of *rand* by 2 and then subtract 1 to get x and y.
    c. A game result is a pair [x y].
6. Step 2: Define how you win.
    a. You win if the dart is inside the circle. The dart is inside the circle if $x^2 + y^2 < 1$.
    b. Define a function `win?` that takes a dart throw and checks whether it is inside the circle.
7. Step 3: Create a sequence of game outcomes, as in the coin-toss game.
8. Step 4: Calculate the probability of winning, as in the coin-toss game.
9. Now multiply the result by 4 and see if the number looks familiar. Try increasing the number of trials to get a more accurate answer.

Task 2—Shut the Box

1. The game is a box with 9 levers and a pair of 6-sided dice. It is a solitaire game.
    a. Each turn the player rolls the dice. The player must then flip down any set of levers that add up to the total on the dice. If this cannot be done, the player loses.

    b. Once all the levers are flipped, the player wins.
2. Play the game once yourself:
    a. Write the numbers 1 through 9 on a piece of paper to represent the levers.
    b. If you don't have a pair of dice on you, roll some virtual dice: https://www.random.org/dice/.
    c. Cross out any set of numbers that add up to your dice roll.
    d. Repeat steps (b) and (c) until you win or lose.
3. Now make up one example play through the game that will result in a loss:
    a. You control the dice rolls for this example.
    b. Keep track of everything as you go.
    c. Start with all 9 levers up. (Write down the numbers 1 through 9.)
    d. Write down a dice roll. Write down next to it which levers you will flip down.
    e. Write down the numbers of the levers 1 through 9 again, but cross out the ones you flipped down.
    f. Repeat steps (d) and (e) until the game ends in a loss.
4. Do the same thing again, but write down a sequence of moves that will result in a win. You can do this because you are controlling the dice rolls for this example, too.
5. Compare your answers to your neighbor. See if you both think you are both right.

Task 3—Game state

1. The game state consists of a set of 9 levers, each of which can be up or down, and a given dice roll. How many game states are there?
2. How will you represent a game state in Clojure? It consists of two items: a configuration of the levers (the box) and a dice roll.
    a. How will you represent the box? Write an example.
    b. How will you represent the dice roll? Write an example.
    c. Suppose the box already has the 1, 3, and 6 levers down, and the dice roll is a 5 and 3. Write down how you are representing this particular case.
3. Create a new file called "shutthebox.clj" (or whatever you want).
4. Define a sample box configuration. Use code like this:
    ```
    (def sample-box …)
    ```
    Fill in the ellipsis (dots) with a sample box configuration using your representation from 2.c above.
5. Create a `dice-roll` function that returns a random valid dice roll. The `rand-int` function should be helpful.
    ```
    (defn dice-roll [] …)
    ```
6. Share the code with the speaker, or get help from the speaker or others as necessary.
7. Next, we need a function called `possible-moves` that does the following: Given a game state (a box and a dice roll), return a set of possible moves from that state. For example, at the beginning of the game, if an 11 is rolled, the player can flip the 8 and the 3; or the 7 and the 4; or the 1, 2, 3, and 5; etc. This is a bit complicated, so we will start with something simpler and come back to this.

Task 4—Simplified Shut the Box

1. Our simplification will be the following: When the player rolls the dice, if the lever with that number is up, the player flips it down and continues. If that lever is already down, the player loses.
    a. For example, say the box already has the 1, 3, and 6 flipped down. If the player rolls a 9, the 9 lever gets flipped down. But if the player had rolled a six, they would lose, because the 6 lever is already down.
    b. Since it's not possible to roll a 1 with two dice, let's allow the 1 lever to start flipped down.
2. It may not be a fun game, but it is still possible to win. Write down a sequence of rolls that will produce a win, and a sequence that will produce a loss.
3. Write a function called `possible-moves` that does the following: Given a game state (a box configuration and roll), return one of two things:
    a. If the roll does not result in a loss, return a set containing one item: the box configuration that results from flipping down the lever shown on the dice.
    b. If the roll is a loss, return an empty set.
    c. Note that `possible-moves` is supposed to return a set `#{…}` in either case: either a set with one item or a set with no items.
4. Test your function on the sample state you defined in the last task.

Task 5—Play the game from the REPL

1. Define a `start-box` value that describes the beginning of the game. (For now, we will start with the 1 lever flipped down.)
2. Using the REPL, call the `possible-moves` function and the `dice-roll` function repeatedly to play through the game. Use the output of `possible-moves` each time as the input to the next time. You can copy-paste the results or use `*1` for the last result.
3. Share the code and REPL interactions with the speaker or a neighbor.
4. The next thing we need is a function that repeatedly uses the `possible-moves` function and the `dice-roll` function to play through the game to a loss or a win. We will do this in the next task.

Task 6—Play the game automatically

1. Write a function `play-game` that plays the game from beginning to end. The function should return `:win` if the game is a win, and `:loss` if the game is a loss. (In our simplified game, there are no choices to make, so the computer can play the game without making any decisions.) The function should take one parameter: a starting box configuration. The function should play the game starting there, and go until the game is lost or won. The result will be random, based on the dice rolls, but wins and losses should both be possible. The procedure is as follows:
    a. If all the levers are down in the given box, the game is a win.
    b. Otherwise, roll the dice and proceed:
    c. Get the possible moves for the current box and the dice.
    d. If no possible moves, the game is a loss.
    e. Otherwise call the function recursively with the new box state.

2. Remember rolling a 1 on two standard dice is impossible, so winning the game will be impossible unless we start with the 1 lever already flipped.
3. Add a print statement to display the box and the dice roll at each step. Run the program a few times and check the console to see what is printed. You should see the game played through to a win or a loss (usually a loss).
4. Wins are pretty rare, so the winning case is difficult to test. I suggest testing the `play-game` function using a starting board with only one lever left up (maybe the 7). From this configuration, losses are still more likely than wins, but wins should come up fairly regularly. Run the test several times, and see if you're getting losses with an occasional win. A 7 should be rolled one out of 6 times, so if you run the test 20 times or so the odds are pretty good (> 97%) a win will occur eventually.
5. When it works and you've seen both a win and a loss, show the result and code to the speaker or a neighbor.
6. Remove all print expressions before the next task.

Task 7—Monte Carlo simulation of Shut the Box

1. Now we want to run the game from beginning to end many, many times to try to estimate the probability of winning. We did this in Task 1 with coin flipping. You can go back and look to remember how to do this.
2. As we did with the coin flipping, you can use `repeatedly` to play several games of Shut the Box, and then use `count` and `filter` to calculate how many were wins. Alternatively, you can just keep track of the number of wins and losses, since we don't really need a whole sequence of all the game results: `{:win 0, :loss 0}`
3. Finally, define a function called `simulate` that takes an integer `n` and plays the game `n` times. Return the probability of winning.
4. Simulate with $n = 1{,}000{,}000$ and write down the probability.

Part 2: Monte Carlo Tree Search

Task 8—Shut the box with a choice

1. Let's introduce a choice for the player. (The computer is the player, for now.) In the real game of Shut the Box, when the dice are rolled, the player can choose to flip any combination of levers that add up to the dice roll. For example, if the player rolls a 6, then the player can flip down levers 2 and 4; 5 and 1; 1, 2, and 3; or just the 6.
2. The simplest way I could think to generate all the choices for a given roll is to generate all combinations (the power set) of the 9 levers and filter them based on which add up to the roll. You can use `clojure.math.combinatorics/subsets` or this code, which I got from Stack Overflow:
```
(defn power [s]
   (loop [[f & r] (seq s) p '(())]
      (if f (recur r (concat p (map (partial cons f) p))) p)))
(def lever-combinations (power [1 2 3 4 5 6 7 8 9]))
(defn levers-for-roll [roll] …)
```
3. It will help to store this in a vector for easy reference later. It is too slow to filter the power set over and over.
```
(def lever-combinations (into [] (map levers-for-roll (range 13))))
```

4. Now that we're playing the real game, given a box configuration and a roll, there may be several options, so the `possible-moves` function has to be adjusted. Modify it so that it returns a set of all the possible moves for a given box and roll. There may be zero, one, or more possible moves. (Remember the possible moves are the box configurations resulting from flipping down levers.)
5. The computer player has no strategy—it makes every choice randomly. Modify the `play-game` function so that it chooses a possible move at random.
    a. If there are one or more possible moves, choose one at random using `rand-nth`.
    b. If there are no possible moves (the set is empty), then the game is a loss.
6. Simulate with $n = 1{,}000{,}000$ again and write down the probability.

Task 9—Shut the Box with user control

1. We will make an interactive Shut the Box game. You can use console interaction (`read` and `print`).
2. Make a copy of the `play-game` function called `play-game-user`.
3. We will modify the `play-game-user` function for human player control.
4. First, display the current box and the dice roll at each turn.
5. Second, whenever there is a choice, do the following:
    a. Prompt with the available choices.
    b. Ask the user to choose one.
    c. Use that choice and proceed. (If you have a recursive call, make sure it's calling the right function.)
6. If there is no possible move (the `possible-moves` function returns the empty set), display a message saying the player has lost.
7. Play the game and see if you can win. If you can't, see if you think it's fun anyway.

Task 10—Look-up table

1. Our ultimate goal with this project is to find the best strategy for playing the game. At any point in the game, we roll the dice. When there is choice of possible moves, which should the player choose? The one with the best probability of winning. So we will build a table of the probabilities of winning from any given box state.
2. Define a function that given a box state, calculates the probability of winning, assuming random play. This is very similar to the `simulate` function of Task 7, but it requires a box state as input.
3. Next, we need a table of box states with their associated probabilities of winning. To generate this table, we need the following:
    a. A list of all possible box states.
    b. A map in which each state is associated with its probability.
4. For part (a), we already know how many there are (2^9). Now generate them.
5. Part (b) needs to construct a map where each key-value pair is a box state and a probability. You can combine the list from part (a) with the associated probabilities using `zipmap` or any other technique.
6. Modify the `play-game-user` function so that it looks up the probability in the map and displays it next to each possible move. Try playing the game. It is giving you hints about what move to make by showing the probability of winning.

Task 11—Monte Carlo Tree Search

1. At this point we only know the probabilities of winning assuming random play. Presumably, we could win more often if we followed an actual strategy. To develop a strategy, we have to learn about the probabilities of winning from each game state. This is known as the “multi-armed bandit problem,” a reference to playing with several slot machines (one-armed bandits) with differing and unknown payouts. In a sense, each choice between possible moves is like choosing which slot machine to play, so this metaphor applies.
2. For Monte Carlo tree search, we build up a tree of states. Each node in the tree is a state of the game, just like the table entries in Task 10. The node keeps track of a record of wins and losses seen so far when starting from that node.
3. MCTS works by repeatedly diving into the tree looking for good outcomes. Each dive adds one new leaf node to the tree. We establish the record for the new node by simulating one random play of the game starting at the new node, as in Task 10. The result back propagates to the parent node, all the way to the root. (If the new node ends the game, we don’t need to simulate, but instead back propagate a win or a loss, as appropriate.)
4. During the dive, the play is not random. We use an upper confidence bound (UCB) strategy. When looking at several choices (the multi-armed bandit), with some knowledge about the choices, we want to pick the one with the best probability, but we also need to explore the ones we haven’t picked much yet. For example, suppose we’ve pulled one lever 100 times and we’ve won 23 times, but we’ve pulled another lever 10 times and we’ve won 1 time. Which should we pull? The first one has better odds, but on the other hand we’re not too sure about the second one since we haven’t pulled it much yet. So each choice is given a rating that tries to balance high probability with how well known our estimate of that probability really is. The choice with the highest rating is chosen.
5. We compute the rating by establishing confidence bounds on our probability estimates. The choice with the highest possible value within the bounds is the choice we make. One formula for confidence bounds is based on the standard deviation of the sample:
$$\frac{1}{n}\left(w + z\sqrt{\frac{wl}{n}}\right)$$
The number of trials is n, and the number of wins and losses are w and l, respectively. Note that w/n is our estimate of the probability of winning. The other term is the standard deviation. The parameter z comes from the confidence interval theory; we’ll use $z = 2$ because it corresponds to 95% confidence. For our purposes, it can be thought of as a tuning parameter, balancing exploration against exploitation of known good choices. Two issues with this formula:
    a. When $n = 0$, we are dividing by 0. This corresponds to a new node in the tree, one we haven’t explored yet. We can safely use infinity for the UCB value, since we definitely want to explore this one.

   b. When $n > 0$ but $w = 0$, the UCB value is 0. This is a big problem because once a node has a losing record with no wins, it will never be selected again. To get around this, we use the "rule of 3." We amend the formula so that if $w = 0$, we use $3/n$ as the upper confidence bound.

6. Define a function `ucb` that takes a record of wins and losses (as in Task 7) and returns the upper confidence bound rating just described. We will need this function when building the tree.
7. Define a function `uct` (upper confidence tree) that takes an existing tree and a box configuration (a node) as parameters.
   a. The "tree" is actually just a table (map) of box configurations with their win/loss records. The initial tree is just an empty map.
   b. The function adds to the tree by running the MCTS algorithm starting at the given box and back propagates the new result.
   c. If the given box does not exist in the tree yet, use the `play-game` from Task 6 to simulate 1 random play from that box configuration. Add the box as a new node to the tree.
   d. Otherwise, use a random dice roll and the `possible-moves` function to calculate the children. Call the function recursively on the child that has the highest UCB rating.
   e. I found it useful to make the function return a vector with two items: the updated tree and the most recent simulation result. That makes the backpropagation easier.
8. Run the algorithm 1000 times (retaining the tree from run to run) and see what the probability of winning is. It should be better than random play. Run it again 10000 times. The probability should improve. Learning has occurred! We have achieved AI!

Part 3: The complete game tree

Task 12—Construct the full tree

1. Because this game has a pretty small number of distinct states ($2^9 \times 12$), we can construct the entire tree in memory, to solve the game perfectly. (It is actually a DAG, because you can reach the same state in different ways.) Therefore, we can compute all the probabilities exactly and don't actually need Monte Carlo tree search or simulation after all.
2. For the perfect search, we build up a tree of states. Each node in the tree is a state of the game. We can think of this as a 2-player alternating game, where the player modifies the state by flipping levers, and the adversary modifies the state by choosing the dice.
3. The root of the tree is the starting box configuration. The children of a box node are the 11 combinations of that box with a dice roll. The children of a box-roll combination node are all possible moves from that box with that roll.
4. Leaves of the tree are loss and win conditions. The probabilities are 0 and 1, respectively. These are the base cases for the recursive construction of the tree. (Also, if

you find that a node already exists in the tree because it was reached from a different path, skip it.)
5. The probability of winning given a box and a roll is the probability of the best child. The probability of winning given just a box is the sum of the probabilities of the 11 children, weighted by the probabilities of the rolls themselves. For example, the probability of rolling a 4 is 3/36.
6. Generate the tree and then write down the probability of winning from the starting box.
7. Modify the `play-game-user` function from Task 9 so that it looks up the probability in the optimal play table instead of the random play table.

Task 13—Display the table of optimal moves

1. Using the tree from Task 12, create a table suitable for export. For each box configuration and dice roll, write the optimal choice of move. In case of ties, just output any optimal choice.
2. Save the table as a text file.
3. Bring your table to a pub and rake in the cash. (Don't tell them I told you to do it.)

References

http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html
http://www.cameronius.com/research/mcts/about/index.html
https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/
http://banditalgs.com/2016/09/18/the-upper-confidence-bound-algorithm/
https://en.wikipedia.org/wiki/Binomial_proportion_confidence_interval
http://blog.thedataincubator.com/2016/07/multi-armed-bandits-2/