Lecture 31: Games, Round 2
10:00 AM, Nov 17, 2017

Contents

1 Review from Last Class

2 Finishing the Code for Yucky Chocolate

3 Other Game Components
   3.1 Human Players
   3.2 Referee
   3.3 AI Players

4 The Minimax Algorithm
   4.1 Implementing Minimax

5 Summary

Objectives

By the end of this lecture, you will know:

- a more advanced and more efficient algorithm to determine a player’s best move, which you will utilize within your Game project
- the structure of Human and AI player modules, as well as the Referee module, for your Game project

1 Review from Last Class

Last time I introduced to you the notion of a two-person, finite, complete-information, alternating-move, zero-sum game. Everything about the game is clearly visible to all players; there’s no random drawing of cards from a deck. I talked about strategies for figuring out how to win such a game. You may recall, I drew a “game tree” for a 2x2 Yucky Chocolate and from values at the terminal nodes (i.e leaves), I filled in “values” at the other nodes. One thing we worked out is that when you know the values of all the leaves, you can figure out all the values at the nodes above the leaves. And, a smart player would go to the leaf that is best for them. If I’m player 1, I want to maximize the values below. If I’m player 2, I want to minimize the values below.

We wrote a good chunk of code for Yucky Chocolate, which we’ll go through as a review:
As you can probably guess, the first type, `which_player`, is used to keep track of players. The second type, `state`, is used to keep track of a state. As the state of Yucky Chocolate depends on how many rows and columns are left, as well as whose turn it is, we define a `state` as holding all this information. The `initial_state` simply keeps track of how the game should start. For our implementation, it should be a 2x2 game, and player 1 should go first. Finally, a `move` is simply the number of rows or columns a player eats, and is represented as such.

Our `next_state` code would look something like this:

```plaintext
let next_state ((n,k, w): state) (m:move) : state =
  match m, w with
  | Row p, P1 when p <= n -> (n-p, k, P2)
  | Row p, P2 when p <= n -> (n-p, k, P1)
  | ...  
```

Your `next_state` will behave differently, but should take in the same thing - a state and a move. This procedure produces the state of the game after the move `m` has been applied to the current state. For Yucky Chocolate, this is simply changing whose turn it is and subtracting the number of rows/columns.

We also wrote `game_status`:

```plaintext
type status = Win of which_player | Draw | Ongoing of which_player
let game_status (s:state) : status = match s with
  | (0,0,w) -> Win of w
  | (_,_,w) -> Ongoing of w
```

Your `game_status` will be very similar. It takes in a state `s` and produces the status of the game: whether a player has won, and if so, which player, whether the game is a tie game, or whether the game is ongoing, and if so, whose turn it is.

Finally, we wrote a `value` procedure that produces the value of terminal nodes.

```plaintext
let value (s:state) : float= match s with
  | (0,0,P1) -> 1.0
  | (0,0,P2) -> -1.0
  | _ -> failwith "value undefined for nonterminal states"
```

The procedure takes in a state, `s`, and produces a value - a higher positive number is desired by player 1, and a more negative number is desired by player 2. This `value` procedure only works for terminal nodes. We’re going to need more information to figure out the best moves to make, and doing so is the subject of the next lecture and a half.

## 2 Finishing the Code for Yucky Chocolate

There are a few more things we need to do before we finish writing the code for Yucky Chocolate.
First up, we need to write a procedure called string_of_player. This function should take as input an argument, \( w \), of type which_player, and output a string representing the current player. For a game like Tic-Tac-Toe, this may output "X" for player one and "O" for player two. For other games, perhaps the strings "Player 1" and "Player 2" work just fine.

Next, we should write string_of_state. This procedure takes in a state, and returns a string that represents the state. This typically means returning a string representation of the game board. For example, string_of_state might return 

```
[ ][ ]
[X][ ]
```

to represent the starting state of a 2x2 Yucky Chocolate game.

The third additional piece is string_of_move, which returns the string representation of an input move. For Yucky Chocolate, string_of_move(row 3) might return "3 rows", which could be used to print out: "Player 1 makes the move: 3 rows."

The final procedure we need to write is move_of_string, which takes in a string as input, and returns a move. It’s used to transform human input into the internal representation of a move. For example, for Connect 4, move_of_string("4") might produce Col 4, which represents a move in which the player puts a game piece in the fourth column. If the input string is nonsense, then this procedure should fail.

3 Other Game Components

Our game needs to have players, and we’re going to implement both human and artificial intelligence players. Human players can type in their next move, where AI players select their move based on the approach we just talked about. We’ll also implement a referee, who can start the the game, manage plays, update game status, and report who won. (AI players will be covered later).

3.1 Human Players

The signature for a PLAYER is as follows:

```ml
module type PLAYER =

sig

  module PlayerGame : GAME

  val next_move : PlayerGame.state -> PlayerGame.move

end
```

We’ve implemented a struct that partially extends this signature below.

The human module only has two things in it - one of them is the game that this is a player for. You cannot just have an arbitrary player, it has to be a “chess player” or a “tic-tac-toe” player. The human module also has a next_move procedure that takes the current state of the game and produces the next move that this player wants to make. For a human player, we read a line of input from the keyword, and call the move_of_string procedure which produces a move from the entered string.
You may wonder, what is the try thing in the code below? What if the player, rather than entering something nice and clear like “r 3”, which could represent eating three rows in Yucky Chocolate, has typed something like “Hello!”, thinking they are working with Eliza. If you try to call move_of_string on “Hello!” it’s going to fail. So, try says “try” to execute this code, and if there are any problems, do something special.

Assuming move_of_string did not fail, we then needs to check if \( m \) is a legal move. If so, we should return the move \( m \). If it isn’t, we should output a message saying it was an illegal move, and re-call next_move to re-prompt the user to enter a move.

If something went wrong while doing this, the try block will catch the errors and try to execute the code after the word “with”. One possible error is an End_of_file error which will be thrown when CTRL+D is pressed - this is a good way to exit your program. If there is some other failure, such as a problem with move_of_string we can match on Failure message, print out the failure message, and re-prompt.

```
module TestHumanPlayer =
struct
  module PlayerGame = Game
  open PlayerGame

  let rec next_move s =
    try
      let m = move_of_string (read_line ()) in
      (* TODO: replace the below expression (between the if and then) *
       * with the proper functionality *)
      if m is a legal move then m
      else
        let () = print_endline "Illegal move."
        in next_move s
    with
      | End_of_file -> failwith "exiting."
      | Failure message -> print_endline message ; next_move s
end;;
```

### 3.2 Referee

The referee is also a module. It’s what sets up and runs the game. It has a notion of the game being played. It also has the notion of two modules, each of which implement a player module - one of which being a human module, one being an AI module (the AI module will be covered later). Don’t worry too much about the code below - it’s a bit convoluted.

```
module Referee =
struct
  (* Change these module names to what you’ve named them *)
  module CurrentGame = Game
  module Human : PLAYER with module PlayerGame := CurrentGame = HumanPlayer
  open CurrentGame
  let play_game():unit =
    let rec game_loop (s: state): unit =
      print_endline (string_of_state s);
```

match game_status s with
| Win p-> print_endline ((string_of_player p) ^ "wins!")
| Draw -> print_endline "Draw"
| Ongoing p ->
  print_endline ((string_of_player p)^"'s turn.");
  let move = Human.next_move s in
  print_endline ((string_of_player p) ^ " makes the move " ^
    (string_of_move move));
  game_loop (next_state s move) in
  try game_loop initial_state with
    | Failure message -> print_endline message
  end;;

The function play_game runs the game_loop procedure over and over again. game_loop prints out the state of the game, and checks to see if the game is over. If one of the players won, it prints out which player won. If there was a draw, it prints out "Draw." If the game is ongoing, it prints out the current player. Then, it asks the current player to calculate their next move for the given state, s. After that, it prints out the move that the player made, and run the game_loop with the game state that results from the player making that move.

The last line of code, Referee.play_game();, is what actually runs the game.

The actual referee that you will be implementing will be more complicated than this. It allows the person running the program to chose whether they want the game to run with two human players, two AI players, or an AI and a human player.

3.3 AI Players

Like a human player, an AI player has to have a game associated with it. It also has to have a way, given a state, to choose a next move. It decides what move to make by looking at the game state, itself!

module TestAIPlayer =
  struct
    module PlayerGame = Game
    open PlayerGame

    (* TODO *)
    let next_move s =
      failwith "not yet implemented"
  end ;;

4 The Minimax Algorithm

Our value procedure determines values at terminal nodes. When the game is over, it checks how good a game was for Player 1. For Yucky Chocolate, it’s +1 or -1. For checkers, it might be “how many more pieces did player 1 capture than did player 2?” Player 1’s goal is generally to maximize
the “value” at the end of the game, because the value represents how happy player 1 is at the end of the game. So Player 2 should try to minimize the value.

We also worked out a naive algorithm last time: we said we could assign “values” to each node that isn’t terminal and call this “nvalue.” Player 1 can look at child nodes and pick the maximum of them. Likewise, player 2 can look at child nodes and pick the minimum of them. To do so, player 1, in a given state \( s \), looks at the value of all states that arise from taking each legal move starting at \( s \), and picks the move that leads to the highest value.

This algorithm will require a few procedures:

1. `legal_moves s` - given a state, \( s \), produce a list of all legal moves at that state.

2. `next_state m s` - given a move, \( m \), and a state, \( s \), produce the state you get to by making move \( m \).

3. `(map (fun s -> next_state m s) (legal_moves s))` - given a state \( s \), produce a list of all possible next-states.

4. `argmax` and `argmin` - which of the legal moves led to the best outcomes?

Player 1 would select the move that produces a state with the highest nvalue:

\[
\text{move} = \text{argmax} (\text{map} (\text{fun} \ s \to \text{next_state} \ m \ s) \ (\text{legal_moves} \ s)) \ \text{nvalue}
\]

Player 2, on the other hand, would select the move that produces a state with the lowest nvalue. This procedure uses `argmin`.

\[
\text{move} = \text{argmin} (\text{map} (\text{fun} \ s \to \text{next_state} \ m \ s) \ (\text{legal_moves} \ s)) \ \text{nvalue}
\]

With this approach, we can associate a computed value to any node. A terminal node has its given value. All other nodes’ values are the maximum value of its children, if it’s player 1’s turn, or the minimum value of its children if it’s player 2’s turn.

However, we can take this a step further: as we move up the theoretical ”tree” of game states, we know that whenever we move up a level, the players switch. We know that player 2 will make the move that leads to the game state with the lowest value possible, and that player 1 will make the move that results in a game state with the highest value possible. So, as we move up the tree, we can switch between storing the minimum and the maximum value of the child nodes in the current node. In doing so, we can take into account the fact that the each player has different goals.

The process of switching between the min and max of the child node values is called ”minimax.”

### 4.1 Implementing Minimax

For small games, we can run minimax until we reach the terminal nodes of the game, and plan moves accordingly. However, for larger games, this simply isn’t possible.

Not to mention, knowing the value of a node simply tells you whether you can win or not - it doesn’t give you any information about how to get to the win state.
To address the former, we need to give our minimax implementation a maximum depth to analyze. To address the latter, we modify the procedures argmin and argmax so that they output a pair, where the first element of the pair is the move to make, and the second element is the value of the game state that results from that move.

Overall, our improved implementation of minimax is as follows:

- **Input**: a game state to start at
- **Output**: a (value, move option) pair where the "value" is the value of the game to P1 if everyone moves optimally at each state, and the move option is the optimal move (if any) for whichever player is supposed to move at the input state.
- **Algorithm**:
  - If the input state is final, return the input state.
  - Otherwise, calculate the game state for each valid move from the current state, and pass each game state in as recursive input to minimax. Thus, you’ll get a value for each child state (relative to the input state).
  - If it’s P1’s turn, calculate the max of the child states’ values, and return that.
  - If it’s P2’s turn, calculate the min of the child states’ values, and return that.

However, we mentioned earlier that it’s not feasible to calculate the game state values for every terminal state in a complicated game. So, what we can do is create another function called estimate_value that estimates the value of a nonterminal node. Then, after the minimax function reaches a given depth, we call estimate_value on the child nodes, and pass those values back up (as opposed to calling minimax until we reach terminal nodes).

5 Summary

Ideas

- We introduced the idea of a “subtree search” algorithm to efficiently implement an AI player.
- We’ve discussed the benefits and tradeoffs of going deeper in the subtree, and writing a more efficient or accurate eval_leaf procedure.

Skills

- We discussed implementation details for several procedures that you will write in the Game project.
- We discussed implementation details of a Human module, an AI module, and a Referee module.
Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback).