(Provisional) Lecture 31: Games, Round 2

10:00 AM, Nov 15, 2019

Contents

1 Tree Game Warmup

2 Review from Last Class

3 Summary

Objectives

By the end of this lecture, you will know:

- a more advanced and efficient algorithm to determine a player’s best move, which you will utilize within your game project.

1 Tree Game Warmup

Tree game is a simple game where blue and red switch off taking their turn. On a turn they can decide which side of the tree to go down that eventually leads to a number. Each number in a leaf is the amount of money blue pays to red. So negative numbers are better for blue and positive ones are better for red. Spike walked through a few examples in class, check out the slides to see them visually.

In order to decide what move each player should do I have to maximize when it is blue’s turn and maximize when it is red’s turn. The value of each turn is sent up the tree until to decide your next move you can just maximize or minimize depending on the children. The purpose of this game was to introduce the minimax algorithm. Keep in mind that we don’t want the final value of the game but also the path we need to take to get to that value.

Every game in the game project looks like this but often with many moves not just two. More move options leads to more children that creates complicated trees. We cannot build a whole game tree with our resources but we don’t actually have to.

In an actually game we will have an arbitrary tree. Each node can have any number of children depending on the state. If you can only afford to look from a certain place in the tree forward 3 or 4 places, you would go down depth 4 and you would have hundreds of options for your next move. If they were all terminal states then you could run minimax to decide. However, they aren’t all going to be terminal states. In this case we need to guess what each value will be. You hallucinate the value of the state and if you do so wisely, you can assign that state a helpful value. For Connect 4, if you have 3 blue pieces in a row with empty slots on both sides, you know that state should have a high value for a blue and a low value for red.
We can use this information to write an estimate value procedure. If we reach a terminal state than that value should be the stored value. If it is not terminal, we look at the state of the board and use our knowledge of the game to estimate a value. A naive way to do this is to look through the rows and columns of the board to look for 3 pieces in a row and then assign a high value to the corresponding player. This would create an AI that would play fairly well but could do much better. We should at least be checking the diagonals and seeing if there are open spaces next to the ones 3 in a row.

2 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 will now go back to code for Yucky Chocolate

```ocaml
let initial_state = (2, 2, P1);

type which_player = | P1 | P2;
type state = (int, int, which_player);

let legalMoves: state => list(move) = (n, k, w) => ...
```

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:

```ocaml
let next_state ((n, k, w): state) (m:move) : state =
switch (m, w){
| 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(which_player) | Draw | Ongoing(which_player)
let game_status (s:state) : status =
switch (s) {
| (0,0,w) => Win (w)
| (_,_,w) => Ongoing (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= switch (s){
| (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.

There are additional pieces of code needed:

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 "[ ] [ ] \n [X] [ ] \n", which prints out as:

```
[ ] [ ]
[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.'
let string_of_move: move => string = fun
| Row(n) => string_of_int(n) ++ " rows"
| Col(n) => string_of_int(n) ++ " cols";

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 and ask the user to input a move again.

All of these procedures go into a yucky chocolate Game module. In lab, you’ll actually go through this for the game 'Nim' Which is good practice for the more substantial game you’ll be writing later.

3

Summary

Ideas

- Minimax is a game-tree search algorithm that returns the best move the current player can make given some depth of the game tree to search through from the current state.
- Game trees are represented implicitly through the game signature instead of actually creating a tree.

Skills

- We discussed implementation details for several procedures that you will write in the game project.
- Understand how estimate value works

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/csci0170/feedback