Lab 11: GoFirst and Nim
12:00 PM, Nov 18, 2018

Contents

1 Prologue 1
2 Game Theory 1
3 Game Signature 2
4 GoFirst, A Game Module 3
5 Nim, A Game Module 5
6 Let’s Play! 6
7 AI 7
8 Generalizing Nim 7

1 Prologue

Bored in the Sunlab, Allan and Sally want to play some snazzy terminal games. Unfortunately for them, they are CS15 students and don’t know OCaml. Allan asks you to help them by writing two games for them to play: GoFirst and Number in the Middle (Nim).

Before we begin please copy over all of the files from /course/cs017/src/lab11

2 Game Theory

For your Game project, you will implement a two-player, sequential, finite-action, deterministic, zero-sum game of perfect information. Let’s define what that means:

• A **sequential** game is one in which only one player moves at a time. Monopoly is a sequential game; *Rochambeau* (i.e., rock-paper-scissors) is not.

• A **finite-action** game is one in which there is a finite number of legal moves available to a player when it is their turn to move. Battleship is a finite-action game; soccer is not.

• A **deterministic** game is one that does not depend at all on chance. Its progress is entirely a function of selected moves. Checkers is deterministic; backgammon is not.
- A **zero-sum game** is one in which what is good for one player is equally bad for the other, and vice versa. All the examples in this section are zero-sum games.

- A game of **perfect information** is one in which both players witness the entire progression of the game. Chess is a game of perfect information; poker is not.

In addition to implementing a game, you will also write a general purpose, artificially intelligent (AI) player that can play any game of this sort. Your AI player will solve a game using the minimax algorithm, together with a helper procedure that estimates the values of intermediate game states. GoFirst and Nim are both examples of two-player, sequential, finite-action, deterministic, zero-sum games of perfect information.

GoFirst is essentially Tic-tac-toe on a one-by-one grid. On each turn, a player has the option to either go or pass. As you may have guessed, because there is only one space to fill, the first player to play go will win.

For the CS 17 version of Nim, the players take turns removing matches from a pile of 21 matches. On each turn, the player whose turn it is takes one, two, or three matches. The player who takes the last match loses.

In this lab, you will implement simple versions of GoFirst and Nim, with an AI player that doesn’t even need to use minimax to play intelligently.

## 3 Game Signature

First, you will design a game abstract data type (ADT). Your ADT will capture (much of) what is common to two-person, sequential, finite-action, deterministic, zero-sum games of perfect information.

**Game Signature** Here is a start at a GAME signature:

```ocaml
module type GAME =
  sig
    (* TYPES *)
    type which_player = P1 | P2
    type status =
      | Win of which_player
      | Draw
      | Ongoing of which_player
    type state
    type move

    (* PRINTING FUNCTIONS *)
    val string_of_player : which_player -> string
    val string_of_state : state -> string
    val string_of_move : move -> string

    (* GAME LOGIC *)
    val initial_state : state
    val legal_moves : state -> move list
```
val game_status : state -> status
val next_state : state -> move -> state

(* SPECIFIC TO HUMAN PLAYERS *)
val move_of_string : string -> move

(* SPECIFIC TO AI PLAYERS *)
val estimate_value : state -> float

end

This signature has four main components:

• Types: How is a player represented? What about a state, or a move?
  
  – which_player: A type with a variant for each player.
  
  – status: Represents the status of the game.
     Either the game is over, and a player has won (or there’s a draw), or the game is still
going on, and it’s a player’s turn.
  
  – state: Represents the state of the game.
     How is the current board represented, and whose turn is it?
  
  – move: Represents any possible move in the game.

• Start state: How do you start a game? Specifically, what is the initial state of the game before
any moves have been made?

  – initial_state: The initial game state.

• Type Conversions: How do you go from your internal representations to an external representa-
tion, strings, and vice versa?

  – string_of_player: Produces the string representation of a player.
  
  – string_of_state: Produces a string representation of a game state: what information
do both players need to know in a turn?
  
  – string_of_move: Produces a string representation of a move.
  
  – move_of_string: Produces an internal representation of a move, given a string. This
involves parsing a string and failing if the input is not valid.

• Game logic: These are the operations that define your game.

  – legal_moves: Produces a list of all the legal moves at a given state.
  
  – game_status: Produces the status of the game at a given state.
  
  – next_state: Given a state and a move, produces the state that results from making
that move.
  
  – estimate_value: Associated with each state in a zero-sum game is a value. However, it
is not always possible to compute such a value. For example, the value of the initial state
in a two-player, zero-sum game is either $+\infty$ (a win for P1) or $-\infty$ (a win for P2), but
in Chess, this value is unknown. The procedure estimate_value computes an estimate of how good each state is.

You won’t have to worry about this for this lab; you can give a dummy representation that goes straight to a failwith. However, we’ve left it in the signature to keep the lab consistent with the upcoming project, Game.

4 GoFirst, A Game Module

Now that you’re familiar with the Game signature, take a look at our implementation of the Game signature for GoFirst.

```ocaml
#use "/course/cs017/src/ocaml/CS17setup.ml" ;;

module GoFirst =
struct
  type which_player = P1 | P2

  type status =
    Win of which_player
  | Draw
  | Ongoing of which_player

  type state = State of (status * bool)

  type move = Move of bool

  let other_player = function player ->
    match player with
    P1 -> P2
  | P2 -> P1

  let string_of_player = function player ->
    if player = P1 then "Player 1" else "Player 2"

  let string_of_state = function State(s, b) ->
    match s with
    Win(p) -> string_of_player p ^ " wins!"
  | Draw -> " Draw"
  | Ongoing(p) -> "It is " ^ string_of_player p ^ "+'s turn"

  let string_of_move = function Move m ->
    if m then "go!" else "pass!"

  let initial_state = State ((Ongoing P1), false)

  let game_status = function
    State (p, b) -> p

  let next_state = function (State (p, b), Move m) ->
    match p, m with
    Win(_, _ | Draw, _) -> State(p, b)
```
Ongoing player, true -> State((Win player), true)
| Ongoing player, false -> State((Ongoing (other_player player)), false)

let legal_moves = function State (p, b) ->
  if b then [] else [Move true; Move false]

let move_of_string = function str ->
  match str with
  "go" -> Move true
| "pass" -> Move false
| _ -> failwith "not a legal move"

let estimate_value = function State (p, b) ->
  match p with
  Win(P1) -> 1.
| Win(P2) -> -1.
| Ongoing(_) | Draw -> 0.

end ;;

Think about how our implementations of the procedures and types from the signature reflect the rules of GoFirst. In particular we’ve defined state as

type state = State of (status * bool)

because at any point a game of GoFirst can be completely described by the status and whether the the board has been filled in. The boolean represents whether the board has been filled in.

Similarly, a player always has a choice of only two moves, to go or pass. So we can represent a move as a boolean:

type move = Move of bool

5 Nim, A Game Module

Now that you are done with GoFirst, Allan wants you to move on to Nim! Follow the same steps as in the section directly above to write another module that implements the GAME signature. This time however, you will implement the game of Nim in the provided /course/cs017/src/lab11/nim.ml file.

Here’s a road map for how you might proceed:

1. Write a template for a module that implements a testable version of Nim, starting like this:

module Nim =
struct
  type which_player = P1 | P2
  type status =
Leave off the signature ascription, meaning : GAME. You’ll see why it’s unnecessary later.

2. Fill in the types for state and move. We have filled in the player and status types for you.

What are the states in a game of Nim? To answer this, think about what you would need to record if you were in the middle of a game of Nim, and you intended to take a break and finish the game later. Your state type should encapsulate just these things.

Once you have determined what is needed to represent a state in Nim, you can then go on to determine how to describe legal moves. That is, what will transition the game from one state to the next?

3. Add stubs for all of the procedures in the GAME signature. A stub is just a procedure header plus a body which does nothing useful, or reports an error. For example, here’s a stub for the legal_moves procedure:

   let legal_moves a_state = failwith "not implemented"

4. Gradually fill in all of the stubs you just wrote until they’re working. Remember, you don’t have to complete estimate_value!

At this point, two human players should be able to play your game. (See Section 6 for instructions.)

You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

6 Let’s Play!

Here is a player’s signature. All a player does is choose its next move.

module type PLAYER =
  sig
    (* given a state, and decides which legal move to make *)
    val next_move : Nim.state -> Nim.move
  end ;;

There are two kinds of players in this world: human players and AI players. For now, we’ll worry about human players, which can be found in the /course/cs017/src/lab11/human_player.ml file.
The HumanPlayer makes its moves based on user input. We’ve written most of this player for you because it uses some imperative language constructs.

**Task:** Make the changes described in the comments to get it working!

You’ll notice that HumanPlayer is a functor. A functor takes in a module and uses it to produce another module. In this case, the module the HumanPlayer takes in is the Game that it’s playing. This means that, to create a HumanPlayer for a game of GoFirst, you’d have to write something like `module HumanGoFirstPlayer = HumanPlayer(GoFirst).` Note that you can’t use “let” to create a new Player; you have to use the “module” keyword to create one.

You may also notice that in the first line, we write `Game : GAME`. This means we’re applying the GAME signature to whatever module we’re given as a Game. If it doesn’t match the signature, this will result in an error; if it does, the module the Player can access will have the GAME signature applied to it, so the user will only be able to access the fields declared in the signature. This is why we don’t need to put `: GAME` on the GoFirst or Nim modules. When we actually go to use these modules, the signatures will be applied.

The Referee is also a functor. It takes in a Game and two Players that both play the Game given, and allows those Players to play the Game. To let a HumanPlayer face off against another HumanPlayer at Nim, you’d write `module HumanHumanNimReferee = Referee(Nim)(HumanPlayer(Nim))(HumanPlayer(Nim)).`

The following line of code starts a game:

```ocaml
HumanHumanNimReferee.play_game()
```

You’ll want to run things from the command line to get a good, interactive game going. Start up the OCaml interpreter by typing the command `ocaml`. Then bring the definitions from files `human_player.ml` and `nim.ml` and `referee.ml` into the environment using the `#use` directive, i.e. type into the REPL the command `#use "human_player" ;;`, etc.

Then create `HumanHumanNimReferee` and call `HumanHumanNimReferee.play_game()`.

**Task:** Play a few games of Nim against your partner.

## 7 AI

Annoyed by Allan winning all their games, Sally wants a program that will help her always win. It is now up to you to write an AI that can defeat Allan.

One nice thing about our simplified version of Nim is that there is a way to determine whether the current player will win or lose based solely on the number of matches left in the pile. What is this formula?

**Task:** Play Nim a few times and try to discover this formula. (Call over the TAs if you need help.) Once you have the formula, you can define a perfect `next_move` procedure: i.e., one that will always win if possible.

**Task:** Write an `AINimPlayer` module that implements the `PLAYER` signature and always plays an optimal move. A stencil for this module can be found in the

---

1By the end of CS 18, you will be able to write a human player all by yourself!
Hint: Don’t worry! This is way easier than it seems.

Note: While the HumanPlayer in human_player.ml isn’t Game-specific, AINimPlayer is going to be specific to Nim. Because of this, it’s not a functor; it already knows what Game it’s playing. In the project, you will design a generic AIPlayer that uses estimate_value to play any Game it’s given.

Task: Play some games with your AI! See if you can beat it, and if so, under what circumstances.

Note: Your referee should look like module HumanAINimReferee = Referee(Nim)(HumanPlayer(Nim))(AINimPlayer) Since AINimPlayer already knows the game it’s playing, you don’t have to provide Nim to AINimPlayer when making the referee.

8 Generalizing Nim

In standard Nim, the initial number of matches has size twenty-one. However, Sally might want to play a longer or shorter game. You will now write a functor Nim_generalized that takes as argument a module agreeing with the signature

```
sig val initial: int end
```

You should be able to apply this functor to get a Nim game with, say, 15 matches to begin with. Once your functor is written, you should be able to write

```
module Nim15 = NimGeneralized(struct let initial = 15 end);;
```

and then use Nim15 in place of Nim in an actual game.

Task: Edit nim.ml to define the functor NimGeneralized.

Task: At the bottom of the file, create the game Nim15 using the functor.

Task: In ai_nim_player.ml, create a new Player, AINim15Player, that plays Nim15 instead of Nim.

Task: Use Referee and some combination of HumanPlayer and AINim15Player to create a module on which you can call play_game().

Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

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/csci10170/feedback](http://cs.brown.edu/courses/csci10170/feedback)