Lab 11: GoFirst and Nim

12:00 PM, Nov 19, 2017

Contents

1 Prologue

While their owners are off learning magic and battling Voldemort, Hedwig and Pigwidgeon decide to pass the time by playing some snazzy terminal games developed just for them. Unfortunately, they are owls, and most owls do not know how to program. Hedwig 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:

```
module type GAME =
sig
  (* TYPES *)
  type which_player
  type state
  type move

  (* INITIAL VALUE *)
  val initial_state : state

  (* GAME LOGIC *)
  val legal_moves : state -> move list
  val next_state : state -> move -> state
  val estimate_value : state -> float

  ...
end
```

This signature has three main components:

- Types: What does a player look like? What about a state, or a move?

  - player: Names the two players.
  - state: Expresses the states of the game.
    What does the board look like, and whose turn is it?
  - move: Describes the game's moves.

- Constructor: How do you construct a game? Specifically, what is the initial game state, before any moves have been made?
  - initial_state: The initial game state.

- Game logic: These are the operations that define your game.
  - legal_moves: Produces all the legal moves 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. The lecture notes will more clearly explain this concept and how to approach writing this method.

Your first task in this lab is to extend this signature to include other useful features, such as a way of prettily printing a game.

For instance, in order to print out a specific player you could use `val string_of_player : which_player ->string`.

Task: Add string_of_player, string_of_state, string_of_move, and move_of_string to your game signature with the correct types.

4 GoFirst, A Game Module

Your next task is to write a module that implements this signature. Specifically, you should implement the game of GoFirst.

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

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

```
module ConcreteGoFirst =
struct
  type which_player = P1 | P2

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

  type move = ...

  ...
end
```

For now, leave off the signature ascription, meaning `: GAME`, so that your code will compile.

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 GoFirst? To answer this, think about what you would need to record if you were in the middle of a game of GoFirst, 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 GoFirst, 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. The `ConcreteGoFirst` module is concrete. You should leave it as such so that your helpers are fully visible, and you can test them.

   But in addition, you should create a fully abstract `GoFirst` module, right below the definition of `ConcreteGoFirst` as follows:

   ```
   module GoFirst = (ConcreteGoFirst : GAME) ;;
   ```

   This will ensure that your `GoFirst` module implements your game signature.

5. Gradually fill in all of the stubs you just wrote until they're working.

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

| You've reached a checkpoint! Please call over a lab TA to review your work.

5 Nim, A Game Module

Now that you are done with GoFirst, Hedwig 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.

Hint: When writing `estimate_value`, think about in what cases you are guaranteed to either win or to lose. For instance, if you have 4 matches, is there any case when you can win?

| You've reached a checkpoint! Please call over 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. Both can be found in the `/course/cs017/src/lab11/players.ml` file.

1. 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.[1] Make the changes described in the comments to get it working!

2. The AIPlayer makes its moves based on artificial intelligence, which you will code soon.

The `Referee` module is in charge of running games. It does so by constructing a `HumanPlayer` and a `AIPlayer`. It then lets you choose which kind of game to run.

```
(* The Referee coordinates the other modules.
 * Invoke play_game to start playing a game. *)
module Referee =
struct
  (* Change these module names to what you've named them *)
  module Game = SimpleGame
  module Human : PLAYER with module Game := Game = HumanPlayer
  module AI    : PLAYER with module Game := Game = AIPlayer

  ...

  let play_game () = play ...
end
```

The following line of code starts a game:

```
Referee.play_game()
```

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

[1] By the end of CS 18, you will be able to write a human player all by yourself!

7 AI

Annoyed by Hedwig winning all their games, Pigwidgeon wants a program that will help him always win. It is now up to you to write an `AI` that can defeat Hedwig.

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.

Hint: Don't forget that you can use `estimate_value` to estimate how good a specific state within the game is.

Task: Write an `AIPlayer` module that implements the `PLAYER` signature and always plays an optimal move.

Hint: Don't worry! This is way easier than it seems. See the comments in the players.ml file to see exactly what you have to do.

> 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 CS17document by filling out the anonymous feedback form: `http://cs.brown.edu/courses/cs017/feedback`.