Project 4: Game

Due: 7:00 PM, Dec 6, 2017

Contents

> "There is no finite game unless the players freely choose to play it. No one can play who is forced to play. It is an invariable principle of all play, finite and infinite, that whoever plays, plays freely. Whoever *must* play, cannot *play*."
>
> —*Finite and Infinite Games*, James P. Carse

> "In chess too," he said at last, "there's a limit to the forecasts one can make. The best possible move, or the most probable one, is the one that leaves one's opponent in the least advantageous

> position. That's why one way of estimating the expediency of the next move consists simply in imagining that move has been made and then going on to analyze the game from your opponent's point of view. That means falling back on your own resources, but this time putting yourself in your enemy's shoes. From there, you conjecture another move and then immediately put yourself in the role of your opponent's opponent, in other words, yourself. And so on indefinitely, as far ahead as you can. By that, I mean that I know where I've got to, but I don't know how far he's got."
>
> —Muñoz, *The Flanders Panel*, Arturo Pérez-Reverte

1 Introduction

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

- A **two-player** game has two players. Tic-tac-toe is a two-player game; hearts is not.
- 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 his or her 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 play a game using the minimax algorithm, together with a helper procedure that estimates the values of intermediate game states.

In lab, you will have implemented two simple games: GoFirst and Nim. This project entails implementing a more complex game, with a more sophisticated `estimate_value` procedure and AI player.

The large-scale structure of the completed project is this: First, there's a "referee" module, which references a Game module and two Player modules. The main procedure in the referee module creates a game (using the Game module), and then alternately asks the two players to make moves in that game, keeping track of whether one of them wins.

The two Player modules describe two kinds of players: humans, or AI players. The referee can run a game between two human players (in which case each human is asked to provide a move in the form of some typed data, with the two players alternating moves, of course), or between a human and an AI player, where the HumanPlayer gets its move from an actual human typing, as before, but the AI player gets its moves by looking at the current game state and figuring out the best possible move to make in this state, and returning that move. The referee can also start a game between two AI players, and you can watch to see what each one does.

2 Pick A Game!

To start this assignment, you must pick a game.

We suggest implementing one of the following "official" games:

- Connect 4
- Frogs and Toads

We describe our version of these games at the end of this handout. If you choose to implement one of these official games, please be sure to read our description of that game carefully, because our rules may vary slightly from the rules you know.

If you choose one of these official games, we guarantee that at least one of the TAs will be expert enough in your game to be a good source of advice and guidance to you. Still, you may choose to implement something else, like checkers; but if you do, we cannot promise you that any of the TAs will be experienced enough with your particular game to help you extensively. If you want to implement an "unofficial" game, please seek approval from the TAs first.

Traditionally these games are played on a board of fixed size. But, in the spirit of abstraction and elegance, we are requiring that your implementation handle boards of varying sizes (e.g., larger boards, or more pieces in play, etc.).

3 Assignment

Once you've picked a game, there are three parts to this project.

1. You have to implement your game. This involves writing a module with data types capable of keeping track of the state of the game, and a few procedures that provide information and generate new states.
2. You have to complete the implementation of the human player. We've provided part of the code, i.e. that which handles receiving input, but you need to handle the rest.
3. You have to build an AI player capable of playing your game. AI stands for artificial intelligence. Your AI player should play intelligently, where "intelligent" is understood to mean rational.

3.1 Game Signature and Module

For the first part of this project, you will need to understand a game abstract data type (ADT). This ADT captures (much of) what is common to two-person, sequential, finite-action, deterministic, zero-sum games of perfect information.

Note: The content of this section is reminiscent of (if not copied verbatim from) Lab 10.

Game Signature Here is the `GAME` signature:

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

  (* INITIAL GAME STATE *)
  val initial_state : state

  (* TYPE CONVERSIONS *)
  val string_of_player : which_player -> string
  val string_of_state : state -> string
  val string_of_move : move -> string
  val move_of_string : string -> move

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

  ...
end
```

This signature has four main components:

- Types: What does a player look like? 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.
    For instance, for checkers, this might be "What does the board look like, and whose turn is it?" For Nim, it might be "how many coins are left, and whose turn is it?"
  - `move`: Represents any possible move in the game. For instance, for tic-tac-toe, a move might be an integer from 1 to 9 indicating which square is to be marked. Note that `move` doesn't depend on the current game state: the move `1` is still a move even in the case where the upper left square of the board has an "X" in it – it's just not a *legal* move!
- Initial Game State: How do you start a game? Specifically, what is the 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 representation, like 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? For tic-tac-toe, this might produce a string that when printed actually shows a 3×3 board with lines dividing up the squares, etc., and a note at the bottom saying "Player X's turn" or "Player O's 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`: provides an estimate of the value, to player one, of a particular state in the game. This value should be positive for states where player 1 seems to be winning (and especially for those in which player 1 has actually won!), negative for those in which player 1 seems to be losing (and especially for those in which player 1 has actually lost), and have intermediate values for game-states in which the advantage to player 1 is not so clear. A very simple `estimate_value` assigns the value 1.0 to each winning state for player 1, -1.0 to each losing state for player 1, and 0.0 for all other states. Your actual `estimate_value` should be more discriminating than this. Also: the values need not be between -1.0 and 1.0: all that matters is that larger values mean that a situation is better for player 1. Having negative values for situations that are good for player 2 makes it easier to see how the game is going, but is not essential. The name `estimate_value` arises because of the way that this particular procedure gets used by the minimax algorithm.

Game Module Your next task is to write a module that implements this `GAME` signature for your chosen game. Here's a road map for how you might proceed:

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

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

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

```
  type state = ...

  type move = ...

  let initial_state = ...
  ...
end
```

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

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:state) : move list = failwith "not implemented"
   ```

4. The `TestGame` module is concrete. You should leave it concrete so that your helpers are fully visible, and you can test them.

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

   ```
   module Game = (TestGame : GAME)
   ```

   This will ensure that your `TestGame` module implements your game signature, for if it does not, then the ascription of the signature `GAME` to the module will lead to warnings/errors.

5. Gradually fill in all of the stubs you just wrote until they're working, *except* `estimate_value`.

At this point, you're almost ready for two human players to play your game.

(See Section 4 for instructions.)

3.2 Player Signature and Modules

The second major part of this project is to implement an AI player that can intelligently play any game that implements the `GAME` signature. To do this, you will implement the minimax search algorithm discussed in lecture, which looks ahead several turns and selects the move that appears to be most beneficial in the long run. But first, let's do something simpler: understand the `PLAYER` signature, and create a `HumanPlayer` that conforms to this signature.

Here's the signature for a `PLAYER`:

```
module type PLAYER =
sig
  module PlayerGame : GAME
  (* given a state, chooses a move *)
  val next_move : PlayerGame.state -> PlayerGame.move
end
```

A player's only task is to figure out its next move, given the state of the game.

As you can see in the first line, a player has a Game as a submodule. Why does it make sense for a player to have this? Because a player plays a *game*, and how can you play a game if you don't know what game you're playing? So a player is first told what game it is playing—this is the submodule—and then with that information in hand, it figures out how to play: i.e., what its next move should be.

In the support code, we partially provide a `Human` player.[1]

Your first player-related job is to complete the `Human` implementation.

Do so now; it should only take a few moments, especially since you already know (from your `GAME` implementation) what legal moves there are, and how to convert a move to a string and vice versa.

3.2.1 Parsing Human Input

There's one problem you'll encounter here (in `move_of_string`): what do you do when you ask the user to type a move (e.g., a number from 1 to 3) and the user types "andrew" rather than "1", "2", or "3"? Well, you want to ask the user to retype the move, but perhaps your `move_of_string` procedure has already tried to convert "andrew" to a number between 1 and 3, been unsuccessful, and hit a "failwith" case, or maybe `string_of_int` has failed by "raising an exception." At this point, your program has halted, and there's nothing you can do.

You can address with with something called a "try-with" expression. Here's an example. Let's start with a simple procedure:

```
let f(x) =
  string_of_int (1000 / x)
f(25);;
f(0);;
```

The results of running this program are this:

```
val f : int -> string = <fun>
- : string = "40"
Exception: Division_by_zero.
```

As you can see, when we evaluated $f(0)$, there was a divide-by-zero exception raised (i.e., there was a problem, and something in the OCaml interpreter noticed it, and did something called "raising an exception", which ended up halting the program and printing a message.)

The structure of a try-with expression is

```
try  <exp> with
| E1 -> ...
| E2 -> ...
```

where `<exp>` is some expression; if it evaluates without exceptions to some value v, then the value of the try-with expression is just v. On the other hand, if there's an exception, we can "pattern-match"

[1] Not literally, of course! We provide an implementation of *PLAYER* that determines its next move based on user (i.e., human) input.

the exception to produce a different value (and to prevent program termination). Here's a modified version of our procedure, and the new results:

```
let f(x) =
try
  string_of_int (1000 / x)
with
| _ -> "Divide by zero";;

f(25);;
f(0);;
```

```
val f : int -> string = <fun>
 - : string = "40"
- : string = "Divide by zero"
```

We've "caught" the divide-by-zero exception and provided a replacement value.

If you look at the referee code, where it asks the user to make one of four choices, you can see this same idea in practical use.

You'll need to write something very like this, to ensure that the move your user types is actually a valid move. HumanPlayer uses this to ask for input again, if there was a failure. You'll use this structure in `move_of_string` to supply more informative error messages in the case something fails or raises an exception.

`HumanPlayer.next_move` only handles `Failures`, not `Exceptions`. This is partly to enforce that we'd like you to fail with your own custom error messages for a user. Also, this way, actual `Exceptions` still fail – for instance, pressing CTRL-D (control D) while reading input raises an `Exception`, and we'd like to make sure the program still fails then, so you can safely exit the game even when asked to enter a move.

Another potential concern when implementing `move_of_string` is *parsing*. When a move consists of multiple pieces (like the (x, y)-coordinates of a token on an Othello board), it's not clear what's the best way for a human user to represent the move.

We recommend that if your game has complex moves, you try separating them with a comma. Let's say you're implementing Othello, and you want a move to consist of two numbers (the row and column in which to place the token). Then, examples of parse-friendly input might be `2,3` or `1,0`.

Note that this is easier to parse than it would be if you included parentheses, like (2,3).

The `substring` and `index` procedures in the `String` module may be especially helpful in getting the first number, second number, etc. from a string.

3.3 The referee

You're now at the point where you can look at the referee code and understand what it does.

You'll want to do that, because one of the parts of your Design Check will be to walk through that code with the TA and quickly explain what's going on.

The other thing you can do at this point is alter the referee code to actually load up *your* game and *your* human player for that game, and then run the referee code to actually manage a game between two human players. (If you ask it to manage a game with AI players, it'll fail, because the default AI player has a stub for `next_move`).

You can read about how to get the referee to work with your modules in Section 4.

3.4 The AI Player

Now we come to the interesting part: writing code that will play the game effectively.

Here is the syntax for creating an `AI` module of type `PLAYER`:

```
module AI : PLAYER =
struct
  module PlayerGame = ...
  ...
end
```

As noted above, you only have to implement one procedure, `next_move`. But implementing this procedure is not necessarily straightforward. Here's a road map for how you might proceed:

1. In the `AI` module, replace the stub for `next_move` with a procedure that simply gets the set of legal moves and returns any one of them. Verify that you can play a game against your AI player, and that everything works fine (except that the AI player makes naive moves).

2. Make a first improvement of `estimate_value`, having it return the max value, if P1 wins, the min value, if P2 wins, or 0., otherwise.

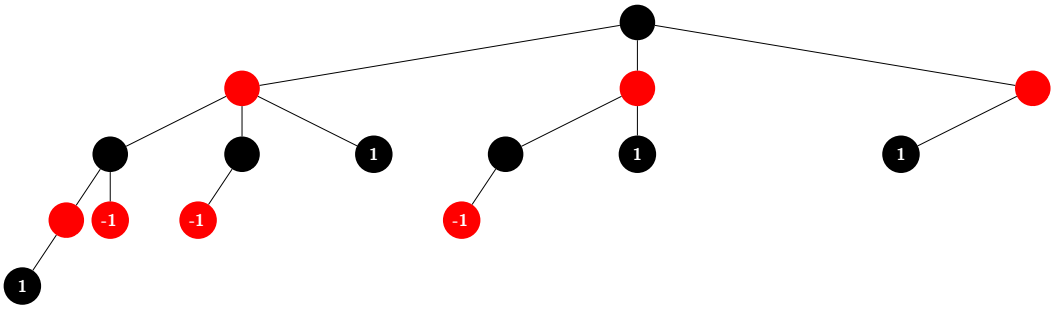
3. Rewrite `estimate_value` to do something sensible. The specifics of this procedure will depend on the game that you choose. This is the hardest part of the assignment for most games, and the one that defines how good your AI is.

4. Write a new version of `next_move` which first uses `estimate_value` to evaluate the results of each possible move, and then plays the best one. Try out this new version—it should be a little smarter.

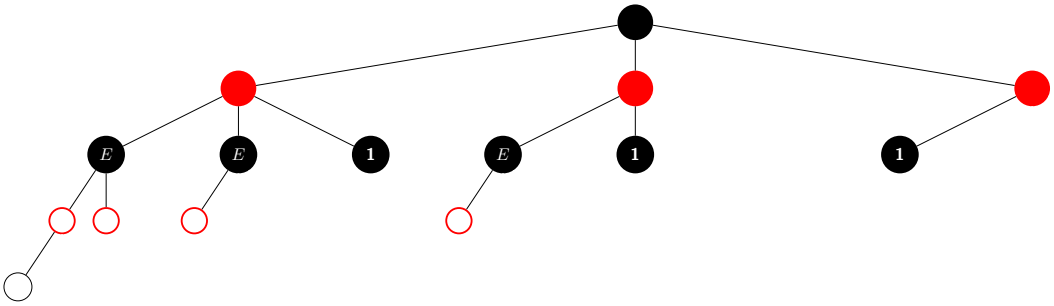
5. Implement limited-depth minimax search. You should look ahead something like 3 to 5 levels, so that you don't get bored waiting for your AI to make its next move. (If your game has many possible moves, deep lookahead will be infeasible. If it has only two or three at each state, then quite deep lookahead may be feasible, as long as `estimate_value` is reasonably efficient.) Typically, extra levels of look ahead are worth much more than an improved `estimate_value` procedure, as long as `estimate_value` is at all reasonable.

   You can imagine a huge tree, where each node is a possible state of the game, and each edge leaving a node corresponds to a move by one player or the other. A leaf in this "game tree" is a state in which the game is over, either because player 1 or player 2 has won, or because it's a draw. When the AI player has to make a move, one way to decide on a move is to look at all possible future states of the game (i.e., the subtree rooted at the current state) and see which move leads to the best possible outcome.

But looking at that whole subtree may not be feasible. Instead, if we look only at the current state, its children, and its grandchildren, we have a depth-3 tree sort of sitting "inside" the game tree. We could look at this smaller tree, and try to decide how to move based on it. To do so, we try to pick a move with the property that when the opponent picks their best move, we still end up in the best possible position. (This is the one-level version of minimax.) For this to work, we need to evaluate how much we like a particular "grandchild" state, i.e., we need to *evaluate* a *leaf* of our tiny tree to see how good a state it is. For that, we use the `estimate_value` procedure in the game.



Above is the game subtree for a NIM game in which there are four pieces left, and it is your turn. The leaves store a value corresponding to the status of the game: 1 if player 1 wins, -1 if player 1 loses.



The bolded nodes constitute the tree that your minimax algorithm will consider, i.e. not necessarily the whole tree. E is an estimate for the value of the game given that state.

Note: The AI player can only interact with the game module through the `GAME` signature. It's not allowed to access any internal representation of states or moves or anything else. So for an AI player to decide on a move, it must ask the game what are the possible moves, and the resulting next states, and then the moves from *those* states, and so on. And it must evaluate the resulting states to see which ones it likes or dislikes, and move accordingly.

3.5 How Everything Fits Together

In our support code, we have provided you with seven files: `sig_game.ml`, `sig_players.ml`, `game.ml`, `players.ml`, `referee.ml`, as well as a copy the the teachpack and an empty `README` which you should fill out. The first two code files provide the `GAME` and `PLAYER` module signatures. They contain all of the types and function type signatures that you will implement in `game.ml` and `players.ml`. We'll now walk through how the modules fit together in the next three files.

'`game.ml`' contains the skeleton for a concrete implementation of the `GAME` signature, which we have called `TestGame`, but which you are free to rename to whatever you would like. If you

were writing the game Nim, we'd recommend a name like `TestNim`. All of the your implementation details will reside in this concrete module. You will also write test cases right after the module is defined. Note that at this point we can still test our local implementation details because we have not yet made our module abstract. Following your test cases, there is the following line of code:

```
module Game = (TestGame: GAME) ;;
```

This line will abstract your concrete implementation of `GAME`, i.e. `TestGame` will now be treated as an implementation of the `GAME` signature. Having done so, we then lose the ability to access internal implementation details. Finally, we call this new abstract game `Game`, a name than can now used in any other file.

In players.ml, we use the same idea to create two concrete implementations of `PLAYER`, which we call `TestHumanPlayer` and `TestAIPlayer`. Notice that in `sig_players.ml`, we specify that `PLAYER`s contain a submodule called `PlayerGame` which needs to be of type `GAME`. Thus, both `TestHumanPlayer` and `TestAIPlayer` must contain a `GAME`, which is demonstrated in the following lines:

```
module PlayerGame = Game
open PlayerGame
```

The first line assigns the `PlayerGame` submodule to be the `Game` that you implemented in `game.ml`. The second line simply eliminates the need to prefix the `GAME` procedures with `PlayerGame`. In other words, you can now call `legal_moves`, when before you would have to write `PlayerGame.legal_moves`.

Additionally, we need to abstract our concrete player modules, which is done in the same way as it was for `Game`. However, there is an additional bit of syntax. You'll notice that instead of just `TestHumanPlayer : PLAYER`, we have

```
TestHumanPlayer : PLAYER with module PlayerGame := Game
```

for both `HumanPlayer` and `AIPlayer`. This added assertion is necessary for later use in Referee so that we will be able to be explicit in saying that `HumanPlayer` and `AIPlayer` use the *same* game. Otherwise, our two player modules could never play together.

Finally, we have `referee.ml`, which will conduct the actual game play. It has three sub-modules: `CurrentGame`, `Human`, and `AI`. We first set `CurrentGame` to be equal to the `Game` that we created in `game.ml`. The second and third sub-modules, `Human` and `AI`, are both implementation of `PLAYER` whose `PlayerGame` is set to `CurrentGame`. We then set these equal to our two abstract `PLAYER`s from `players.ml`, `HumanPlayer` and `AIPlayer`. Feel free to ask a TA if you need any further clarification about the source code!

4 Let's Play!

As we said earlier, the `Referee` module is in charge of running games. Given two players and a game, it starts playing!

```
(* The Referee coordinates the playing of a game by two players. *)
module Referee =
struct
  module CurrentGame = ...
  module Human : PLAYER with module PlayerGame := CurrentGame = ...
  module AI    : PLAYER with module PlayerGame := CurrentGame = ...

  ...

  let play_game () = game_loop (Game.initial_state)
end
```

Then, to use the referee, make sure the names of your `Game`, `HumanPlayer`, and `AIPlayer` modules are correct where they're specified in the submodules.

Run: `ocaml referee.ml`

A menu will be displayed, prompting you to input a number between 1 and 4. This'll determine which players you want to play. For example, if you want to play human vs human, choose 1 and press enter. Then, the game will begin!

5 Notes

- There are some constructs that OCaml will let you use but that we, the CS 17 instructors, will not. Specifically, you are not allowed to use mutation. (If you don't know what this means, don't worry; you won't use it by accident.) As always, if you are unsure about whether a certain OCaml construct is allowed, ask the TAs.
- Please don't "hard code constants" (i.e., have them appear all over your program). Instead, define them at the top of your module. For example, instead of using the integers 5 and 7 in the Connect Four `initial_state` procedure to describe the number of rows and columns in the game board, use an identifier called `initial_rows`, and another called `initial_cols` and assign them the values 5 and 7. Doing so makes it easy to change the board dimensions later.
- The OCaml `List` module has a number of procedures that should come in handy. Documentation can be found online at:
  `http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html`.

  Documentation for the OCaml Pervasives module (i.e., the procedures that are available in any OCaml program) can also be found online at:
  `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html`.

  Old homeworks, labs, and lecture notes may also contain useful procedures.
- Here are some things you should know about strings:
  - The infix operator `^` concatenates two strings.
  - In a string, `\n` (backslash n) represents a newline character. For example, if the string

    ```
    "hello\nworld"
    ```

is printed out,[2] it will have a line break in between "hello" and "world".

- `string_of_int` and `string_of_float` return a string representation of the number passed to them.
- `int_of_string` and `float_of_string` return a number based on the string passed to them.
- `Str.split (Str.regexp "")` returns a procedure that breaks a string into a list of strings; if you implement a game whose moves are multiple numbers, you might want to first do this and then call `int_of_string` on the elements of the list.

If you'd like to use the Str module, put this line at the beginning of your file: `#load "str.cma"`

If you want to learn more about the `Str` library module, check out:
`http://caml.inria.fr/ocaml/htmlman/libref/Str.html`

6 Handing In

6.1 Design Check

Design checks will be held on Nov 20-21, 2017. We will send out an email detailing how to sign up for design checks, so please check your inbox periodically, and sign up as soon as possible.

Reminder: You are required to pair program this project. We recommend finding a partner as soon as possible, as you will not be able to sign up for a design check until you have one.

There are three components to the design check.

First, you should have at least a partial implementation of the GAME signature. At minimum, you should fill out the type definitions for `move` and `state`, as well as writing the `initial_state` procedure.

In addition, what you have should be well written, organized, and commented. You should be able to explain everything you've written to your TA.

Second, you should be able to describe how you intend to implement minimax. Specifically, you should be able to answer the following questions:

- Which procedures in your `Game` module will the AI player use? Why?
- What should the AI player do if there are no legal moves available? If you don't think this should ever happen, tell us why not.
- How does your AI player take into account the fact that the other player wants the AI player to lose? Does your strategy work even if two AI players are playing against each other?

[2] Strings will include these special characters just as you typed them; to print them out, use the print_string operation. Unlike all the procedures we have written this semester, this operation does not only return a value; instead, it does something—namely, it prints out a message. We wrote the HumanPlayer for you because it uses operations like this one that aren't purely functional (because it has side-effects other than the value it returns; in this case the procedure returns a unit, `()`, an empty value in OCaml).

Third, make sure you understand how the support code works and fits together.

- How do you plan to complete the implementation of HumanPlayer?
- How does the Referee code work? You'll be asked to walk the TA through a brief explanation of the Referee code, to prove that you understand it.

This is the minimum that we're expecting, but feel free to do more! Remember, this is your opportunity to have the TAs look closely at your code, and to ask them questions about it.

6.2 Final Handin

The final handin is due by Due: 7:00 PM, Dec 6, 2017.

For the final handin, you are required to hand in seven files: a '`README.txt`' file, a '`game.ml`' file containing your choice of game implementation, and a '`players.ml`' file, containing your implementation of minimax, a '`referee.ml`' file, containing the Referee module, a '`sig_game.ml`' file containing the game signature, a '`sig_player.ml`' file containing the player signature, and a '`CS17setup.ml`' file.

In the '`README`' file, you should provide:

- instructions for use, describing how a user would interact with your program (how would someone play your game against a friend? against the AI?)
- an overview of how your program functions, including how all of the pieces fit together
- a description of any possible bugs or problems with your program
- a list of the people with whom you collaborated
- a description of any extra features you chose to implement

To hand in your files, first navigate to the web handin site and select the game option. Only one partner has to submit the project.

6.3 Grading

The design check counts for 20% of your grade. Specifically,

- Your draft implementation of a `Game` module: 10 points
- Your plans for implementing the `HumanPlayer` and `AIPlayer` module: 7 points
- Your walkthrough of the Referee module: 3 points.

Note: You must decide for sure which game you will implement in time for the design check.

Functionality counts for 60% of your score. Specifically, we will look for:

- An initial state with customizable parameters
- Correctly generating legal moves
- Correctly determining the status of the game
- Correctly transitioning to the next state
- Handling human input and completing a human player
- Estimating the value of a game state well

Since Frogs and Toads is slightly simpler to implement than Connect 4, the maximum functionality score on Frogs and Toads is 53 points. The maximum score on Connect 4 is 60 points. The max scores for any other games you may choose to implement will be determined on a case by case basis; feel free to reach out to the TA list and run possible games by us to determine their point value.

Your AI Player will count for 20% of your grade. Specifically, we will look for:

- A correct implementation of minimax
- Intelligent play

Partial functionality merits partial credit.

You can lose up to 20% of your grade for bad style. Remember to follow the design recipe; that is, include type signatures, specifications, and test cases for all procedures your write. And add comments to any code which would otherwise be unclear.

7 Connect 4 Tournament (Optional)

At the end of this project, the TAs will be running an automated tournament pitting your Connect 4 AIs against each other in mind-bending computerized showdown! Participation in the tournament is completely voluntary, but if you'd like to do so, you must (1) choose the game Connect 4, and (2) represent your `move` type as an integer between 1 and the number of columns in the board, 1 indicating a move into the leftmost column, and each successive integer indicating a move into next column to the right.

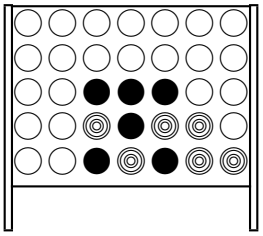
A The Games

A.1 Connect 4

Connect 4 is a game of brutal conflict. If you choose to implement this game, you'll be able to pit the skills you've honed for years against the cold heart of an AI player.

Connect 4 is played on a grid. One player is red and the other black. The players take turns dropping red and black checkers into the grid. The grid can be filled only from the bottom up. The red (black) player's goal is to line up four of red (black) checkers in a row.

Figure 1: A 5×7 Connect 4 game. Black has a guaranteed win.



Parameters While standard Connect 4 is played on a 5×7 board, you should make the size of your board configurable, allowing for play on any board with at least 4 rows and at least 4 columns. If you want an extra challenge, you can also make configurable the number of consecutive checkers required to win.

Strategy The Connect 4 strategy can be summarized fairly simply: always try to extend a line, and if possible, to leave both ends of the line open. If you can manage to make 3-in-a-row with empty slots at both ends, you have a guaranteed win. More generally, if with one move you can complete two different 3s-in-a-row, each with an empty slot at one end (and the empty slots are distinct), you're likewise guaranteed to win. As a special case of this, the trademark Connect 4 strategy is the one where you set up the two empty slots to be vertically adjacent (as in Figure 1); thus in blocking one win, your opponent sets you up for another!

A.2 Frogs and Toads

Frogs and Toads is a simple game with complex strategy and solutions.

Cribbing from the Wikipedia article on the game,

> Toads and Frogs is played on a $1 \times n$ strip of squares. At any time, each square is either empty or occupied by a single toad or frog. Although the game may start at any configuration, it is customary to begin with toads occupying consecutive squares on the leftmost end and frogs occupying consecutive squares on the rightmost end of the strip.
>
> On his turn, Left may move a toad one square to the right if it is empty. Alternatively, if a frog occupies the space immediately to a toad's right, and the space immediately right of the frog is empty, Left may move the toad into that empty space; such a move constitutes a "hop". Toads may not hop over more than one frog, nor are they allowed to hop over another toad. Analogous rules apply for Right: on a turn, he may move a frog left into a neighboring empty space, or hop a frog over a single toad into an empty square immediately to the toad's left. As usual, the first player to be unable to move on his turn loses.

It's best to play this a few times on, say, a 1×8 board that starts with 2 toads at the left and 2 frogs at the right, to get a feel for how the game goes.

Parameters The number n of squares in the strip, and the number k of toads/frogs initially at each end, are both parameters. $2k$ should be less than n.

Strategy Surprisingly little is known about general strategies for the game, although you are welcome to look at the wikipedia article and the work by Erickson mentioned there.

A.3 Other Games

Dissatisfied with the choices we've given you? Feel free to implement something else. The only restriction is: the game you pick has to be of the right type—that is, it has to be a two-player, sequential, finite-action, deterministic, zero-sum game of perfect information—and it must be nontrivial. Other acceptable examples include Chess, Gomoku, Pentago, Othello, and Dots and Boxes. What should guide your decision is your ability to code the game you choose and the corresponding AI in the time frame allotted.

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`.