1 Fifty Shades of Winning

“It’s not just about winning and losing... It’s about the point spread.”

So far we have considered a game to be a win for one player, a win for the other player, or a draw. In fact, the GAME signature requires that. However, in some games it’s not just about winning and losing—there are degrees to which one wins and loses. This is modeled by defining payoffs. In a zero-sum game, the payoff to one player plus the payoff to the other player must equal zero. You can imagine that the result of the game is that one player pays money to the other player—the amount of money one player gets equals the amount the other player loses. For such games, it suffices to define just Player 1’s payoff because Player 2’s payoff is always the negative of Player 1’s. (Note that Player 1’s payoff could be a negative number.)

In this scenario, Player 1 wants to maximize the payoff and Player 2 wants to minimize it.

2 Game-Tree Game:

Consider a game defined by a rooted tree. Each nonleaf node is labeled by + or -. The children of a + node are - nodes and vice versa. Each leaf node is labeled by a number.

Here is how the game is played. The state of the game is a node of the tree. The initial state is the root. If the current node is a + node, it is the turn of the + player. If the current node is a - node, it is the turn of the - player. Whoever’s turn it is gets to choose which child to move to. If the current node is a leaf, the game is over and the payoff is the label of the node.
Now, you can try playing this game with a bigger tree:

```
+  
/ \
-   -
  / \
+/  +  +  +
10 25 45 55 12 15 5 20 -10 -1 -90 10 45 30 -10 30 26 -1
```

**Strategy:** first figure out the value of each node. + player goes to child with largest value, and - player goes to child with smallest value.

- **Value** means the payoff if a player were to play optimally.
  - Value of a + node is the maximum value of any of the children
  - Value of a - node is the minimum value of any of the children
  - value of a leaf is just the label

Now suppose the tree is represented by a procedure `successors : node -> node list`

Define type `node_label = Plus | Minus | Leaf of float`  
Also suppose that you have a procedure `get_label : node -> node_label` which, given a node, will return the node’s label.

**Quiz:** Write a procedure `compute_value : node -> float` with the following I/O spec:

- **input:** a node in a game tree
- **output:** value of that node
Answer:

```ocaml
let rec compute_value : node -> float = function (a_node) ->
    match get_label(a_node) with
    Leaf(x) -> x
  | Plus -> (List.fold_right (fun x y -> (max x y))
              (List.map compute_value successors(a_node))
              min_float)
  | Minus -> (List.fold_right (fun x y -> (min x y))
              (List.map compute_value successors(a_node))
              max_float) ;;
```

2.1 Strategy for playing any game:

It is also worth noting that every game is a game-tree game. As discussed in previous lecture, each node represents a state of that game. Thus a potential strategy for estimating the value of these states can be done as follows:

- Choose one player, say Player 1, to be the + player
- For state that is a win for Player 1, use +1.0 as the payoff
- For state that is a win for Player 2, use -1.0 as the payoff
- For state that is a draw, use 0.0 as the payoff

Then, we use this value to decide which move to make:

- For Player 1, choose whichever move brings game to state with largest value
- For Player 2, choose whichever move brings game to state with smallest value

Will the above strategy work? Unfortunately, not quite. The reason for this is because most games will have game trees that are extraordinarily large. Thus, it is not realistic to search the entire tree to find winning and losing states. For this reason, we must modify our approach.

Instead, we can use a **Static game-state evaluator**. The algorithm (minimax algorithm) is correct but would take way too long. So, we can instead estimate the value of a game given its state. Then, we terminate the searching of our tree after 3-5 ply.

**Note:** A *ply* is what we ordinarily call a *turn*.

There is a large part of the tree that the evaluator procedure is never going to explore, so will treat certain nodes as leaves.

The payoff estimator is called `estimate_value` in our project. Usually it is called a *static evaluator*. The 'evaluator' portion of this is because it computes a value, and the 'static' portion of this is because it does not consider moves or different states over time; it’s just taking a fixed look at the current state. Then, the minimax algorithm does the work of looking at the different moves and the tree, potentially using our static evaluator as a helper procedure.
The static evaluator depends on the game. Traditionally, these are designed by experts (e.g., chess grandmaster).

2.1.1 There’s more to game-tree search than minimax

In graph search, we used `successP/existsP`. This procedure used short-circuiting, meaning that the search stopped after finding the first true result. You may remember that in the first attempt at game-tree search, we used `existsP` and `allP`. These procedures used short-circuiting. `existsP` stopped after finding the first true result and `allP` stopped after finding the first false result.

Let’s think about how we can short-circuit using the minimax algorithm. What is the equivalent here? How can we stop early when computing the maximum or minimum?

**Quiz:** Choose the best move for the + player and compute the value, given the tree below.

![Game Tree](image)

**Key idea:** You don’t need to know values of all nodes to choose a move.

How can we say that you don’t need to know the values of all the nodes to choose a move? For this answer: look at the following graph:
In each of the right subtrees, the opposite player would only choose one of the missing nodes if the value were less than the one furthest to the left. Because both of the values of the leftmost subtrees are less than 27, we don’t need to explore the remaining trees. This is a concept known as *alpha-beta pruning*.

### 2.2 Alpha-Beta Pruning

#### 2.2.1 Alpha Pruning for *minimum*

This is a way of creating a short-circuiting *minimum* procedure. The trick here is that the procedure will take an additional argument, traditionally called $\alpha$.

- **Input:** float $\alpha$, procedure $\ell$ list $[x_1; x_2; \ldots; x_k]$.
- **Output:** minimum among $[\ell x_1; \ell x_2; \ldots; \ell x_k]$ if minimum is greater than $\alpha$, else $-\infty$.

**Note:** In OCaml, you can use `min_float` as a value for $-\infty$.

The logic behind this is that if you already know that overall value is at least $\alpha$, then any node with value less than $\alpha$ is irrelevant. This value can help you prune other nodes.

#### 2.2.2 Beta Pruning for *maximum*

This is a way of creating a short-circuiting *maximum* procedure. The trick here is that the procedure will take an additional argument, traditionally called $\beta$. 
• **Input:** float $\beta$, procedure $f$ list $[x_1; x_2; \ldots; x_k]$.

• **Output:** maximum among $[f x_1; f x_2; \ldots; f x_k]$ if minimum is greater than $\beta$, else $\infty$.

**Note:** In OCaml, you can use `max_float` as a value for $-\infty$.

The logic behind this is that if you already know that overall value is at least $\beta$, then any node with value less than $\beta$ is irrelevant. This value can help you prune other nodes.

### 2.2.3 Programming Strategy

In order to implement alpha-beta pruning, you must keep track of both $\alpha$ and $\beta$ and short-circuit both the max and min.

### 2.3 Iterative Deepening

This is a very simple concept which answers the question: how many plys should we search?

Often times, we have a time limit which we are dealing with, and we want to search as many plys as possible without going over the time.

The strategy for iterative deepening is as follows:

- Try depth 1
- If you haven’t run out of time, try depth 2
- If you still haven’t run out of time, try depth 3
- and so on

In terms of the game project, this will look like manually changing your value for the amount of plys your minimax algorithm should search, and seeing the maximum plys you will be able to search without your algorithm taking an absurd amount of time.

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](http://cs.brown.edu/courses/csci0170/feedback)