1 Estimated Value and Search Subtrees

For the Game project, we need to write an AI player that will be able to determine which move to make at any given state, s, in the game.

To do so, we need to look at the possible legal moves at this state s, and the states that these moves lead to. For each of these subsequent states, we need to look at the values of these states. How can we find the value of a state?
For each terminal state, \( t \), there’s some definite value to P1 (player 1), that we could find using the function \( \text{value}(t) \). This gives positive values to states that are good for P1, and negative values to states that are bad for P1 (and good for P2).

Additionally, for any nonterminal state, \( n \), we can estimate the value of that state to P1, using the function \( \text{estimate_value} \). In fact, we can extend \( \text{estimate_value} \) to apply to terminal states as well. We can write \( \text{estimate_value} \) such that for terminal states, it returns the value for P1 for that state, and for nonterminal states, it estimates the value of that state to P1.

We’re going to look at a “subtree” of the entire game tree, which is a a big picture representation of the entire game and all possible states of the game, starting at the state \( s \) and going down a certain number of levels, \( d \). For each leaf of the subtree (that is, the states of the subtree that are \( d \) levels down), we will assign a value to this leaf using \( \text{estimate_value}(n) \).

Keep in mind that the leaves of the subtree are not necessarily actually leaves of the game tree, because the states might not be terminal states. However, even if they are not terminal states, we can still get a value of the node at that point by estimating the value at that state.

Then, using the minimax algorithm, we propagate the values of these nodes up the tree to our current state, \( s \), to get which move we should make at the current state \( s \).

The reason we do not look at the entire game tree and instead look at a subtree is because the entire game tree could have millions of nodes, so looking at all of them would take too long.

Additionally, the reason why we don’t just call \( \text{estimate_value} \) on all the possible legal next moves instead of using minimax to go down some layers and then calling \( \text{estimate_value} \) is because it’s really hard to write a good \( \text{estimate_value} \) procedure. Errors in \( \text{estimate_value} \) can lead to bad moves. Using the minimax algorithm and going down a number of layers tends to mask these errors.

How do we actually write the \( \text{estimate_value} \) procedure? If you’re playing a game where you accumulate points, one way of writing \( \text{estimate_value}(n) \) would be by calculating P1’s points minus P2’s points at state \( n \). Essentially, \( \text{estimate_value} \) is pretending that the game stopped right here, and seeing how much P1 is winning by.

How would you write \( \text{estimate_value} \) for Connect 4?

Some ideas could be to see if it’s P1’s turn, and P1 has a row of three with both ends able to be played on the next turn. If so, you could return +100.

If not, does P1 have two intersecting rows of 2 each, so that playing at the intersection will give two rows of three? If so, +50.

If not, does P1 have any rows of 2 that can be extended to 3? If so, +5 for each of these. If it’s P2’s turn, we can do a similar set of computations but for P2 instead of P1. \( \text{estimate_value} \) should look at the current state of the game and calculate how good it is for P1 - how good it is for P2.

\( \text{estimate_value} \) estimates the value of the current situation without doing any look ahead. The look ahead part comes in the minimax algorithm. Instead, the point of \( \text{estimate_value} \) is to add something above and beyond that - the actual knowledge of the game and what states are good.

Writing \( \text{estimate_value} \) is often the hardest part of Game for many games. It can be helpful to play the game multiple times with your partner to get a sense of which states are good and which states are bad.
2 Minimax

The naive version of minimax is a one move look ahead, where we look at the value of the state resulting from each possible move, and pick the one with the best value for you. The better way of doing it is to look ahead about 4 moves, and use minimax to find the best move.

The true minimax looks all the way to the bottom of the game tree and bring the values and best moves back up. Finite-horizon minimax only looks a certain number of layers deep. The input to a finite-horizon minimax procedure would be a state \( s \) in a game tree, and a depth \( d \) to search.

If \( s \) is nonterminal, and \( d > 0 \), the procedure would output a (value, move option) pair telling you the best move for the current player to make, and the value of making the move to P1. If \( s \) is terminal or \( d = 0 \), the procedure would return a (value, None) pair telling you the value of state \( s \).

The algorithm for finite-horizon minimax is recursive. If \( d = 0 \), the procedure returns (estimate_value(s), None).

If \( s \) is terminal, return (estimate_value(s), None). estimate_value here just returns the actual value of \( s \) since \( s \) is terminal.

Otherwise, if it is P1’s turn, then among all the possible legal moves at this state, we find the move \( m \) with the largest next-state value, \( v \), and return \((v, \text{Some } m)\).

If it is P2’s turn, then then among all the possible legal moves at this state, we find the move \( m \) with the smallest next-state value, \( v \), and return \((v, \text{Some } m)\).

3 AI Player

In order to create a smart AI player, we need to be able to write estimate_value, which takes in a state, \( s \), and outputs a float that tells how good the state appears to be for P1.

A good estimate_value procedure is important for the AI Player to be good. However, there is often a tradeoff between a good estimate_value procedure and a fast estimate_value procedure. Sometimes it’s more important for the estimate_value procedure to be faster, because this allows the minimax procedure to look one level deeper. This procedure is part of the Game signature, so it’s available for the AI player to use.

We use this estimate_value procedure in the procedure next_move, which for a given state \( s \), selects the move the computer player wants to make.

The method for writing next_move is to use a finite-horizon minimax to choose the move. We also have the legal_moves procedure in order to find all the possible moves at \( s \), and the next_state procedure, which can be used to find all the possible next states. Then, the estimate_value procedure can be used to find the values of these next states.

Note: You actually don’t need to know what game you’re playing - the code for next_move is exactly the same. This is because the procedures estimate_value, legal_moves, and next_state are all in the GAME signature. These procedures might be implemented differently for different games, but the way next_move uses these procedures will be the same for every game. In other words, you only need the GAME signature, not the game itself.

The general approach for writing next_move, where you are given a state \( s \), and it’s your move is
let (m, v) = minimax(s, d)
return m

4 Game Trees and Trees

You might have noticed that nowhere in the Game code do we construct an actual “tree”. This is because the tree structure is implicit in the GAME signature. For example, the root of the tree is represented by the initial_state. The edges from a node to its children are represented by legal_moves. The node’s children are represented by next_state. Finally, game_status is able to tell whether the child is a leaf or a node (a terminal state vs. a nonterminal state).

The reason we encode this tree without actually constructing a tree is because the game tree for these games would be too big to construct the whole tree, so instead, by encoding it implicitly, we can see the parts of the tree that we want without having to construct the whole tree.

5 Functional Lists

This same idea can be illustrated with the idea of functional lists.

We’re going to build a new kind of list: functional lists. As a reminder, a list is either empty, or cons item lst. where lst is a ‘a list and item is of type ‘a. The rules for lists are that (first (cons a b))=> a and (rest (cons a b))=> b

We will be representing lists as functions from \( N \rightarrow int\) option. We can start with a function \( f: N \rightarrow int\) option, and treat \( f\) as a list. For example, in order to write first,

```ml
let first f = match (f 0) with
| None -> failwith `Can't compute first of an empty list'
| Some x -> x
```

Whenever you enter a number into \( f\) and get Some \( x\), \( x\) is the list item, and when you get None, that’s an empty list. In this way, the function \( f\) is able to represent a list.

In order to check whether the list \( f\) is empty, we can check whether \( f\) with 0 as an input returns None. In order to check whether the list \( f\) is a cons list, we can compute the not of checking whether it is empty

```ml
let emptyP f = ((f 0) = None);;
let consP f = not emptyP f;;
```

How do we compute the rest of the list? We need to return something that’s a function. We can do this by returning a function that is the same as \( f\), but on \( x + 1\) instead of \( x\). So \( rest f\) applied to 0 would be the same as \( f\) applied to 1.

```ml
let rest f = fun x -> f (x + 1);;
```

However, this isn’t completely right because \( rest\) should not operate on empty lists.

Instead, we can check to see if \( f\) is empty by matching.
let rest f = match f 0 with
| None -> failwith "Empty lists have no rest''
| _ -> fun x -> f (x + 1);

How do we write the cons procedure? This procedure takes in an item, hd, and a list, tl that we
want to cons hd onto. We also need to return a function, g, such that g(0) = hd. Additionally,
g(1) = tl(0), and g(2) = tl(1), and so on.

So we can output a function such that if 0 is inputted, we return hd, and if a number n greater
than 0 is inputted, we return the same thing that tl would output on n − 1.

let cons hd tl = fun x -> match x with
| 0 -> hd
| n -> (tl (n -1))

How do we compute the length of this list? We can compute it the same way we compute length
of regular lists.

let rec flist_length f = if (emptyP f) then 0 else 1 + flist_length (rest f)

Here, we can’t use cond or match cases, because with our previous list representation, we were
using the structure of the data to use cond or match. Here, we don’t have a nice way to do that, so
we have to use if expressions.

What would happen if we had a functional list

let f(n) = Some 0;;

This represents a list whose first, second, and third item up to infinity are all 0. The length of this
list would be infinite. Because of this, functional lists could possibly include infinite length lists,
which requires us to rethink many of the procedures that operate on lists.

This same idea goes for game-trees, because a game like checkers has an infinite game tree! That’s
why we restricted to finite games, so that next_state would eventually reach a leaf.

This generalizes to “streams”, which is an abstraction of data that keeps arriving forever, and is a
topic that is covered in future CS courses.

6 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.
- We can similarly represent lists as functions.
Skills

- Write the minimax algorithm
- Understand how estimate value works

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)