Nim (Tree search)

Help Center

Overview

As we saw in a previous practice activity, Nim is a mathematical strategy game in which two players take turns removing items from one of several heaps. As before, we will focus on a simple variant of the game (sometimes called 21) in which a single heap starts with 21 items and the players then alternate having a choice of removing up to three items from the heap. The winner is the player to remove the last item from the heap.

In this activity, we take a different approach to analyzing Nim. In particular, we will develop a simple recursive method that searches the tree of all possible games of Nim and determines whether a game, given a specified number of starting items, is either `"won"` or `"lost`. Once we have developed our solution, we will then reconsider this method as an instance of a Minimax algorithm. Finally, we will consider how to improve the performance of our recursive method via memoization.

A recursive solver for Nim

Our goal for this section is to implement a function `evaluate_position` that takes the current number of items in the heap and returns `"won"` or `"lost"` depending on whether the current game of Nim can be won from this position using optimal play. Before we reveal any further hints, you may wish to attempt this problem using the following program template. Our recursive solution requires adding only four lines of code to the template. Note that it is, strictly speaking, **not** an instance of a Minimax algorithm.

If you need more help, consider the following English description of our solution method. When it is my turn to move, my goal is to find a move that causes my opponent to be `"lost`. If I can find such a move, the current position is `"won"` for me. Other the other hand, if all of my moves lead to a position that is `"won"` for my opponent, then my current position is `"lost"`. With this hint, we suggest that you attempt to implement `evaluate_position`. Once you are satisfied with your attempt, feel free to consider our solution.

Before we proceed, we note two important features of this method. First, the base case for the recursion occurs when `current_num` is zero. In this case, the range for the `for` loop is empty and no recursive calls to `evaluate_position` are made. When `current_num` is zero, the method correctly returns `"lost"` to indicate the your opponent has just made the winning move of removing all of the remaining items.

Second, `evaluate_position` immediately returns `"won"` whenever a move that places the opponent in a losing position is detected. As result, `evaluate_position` correctly determines whether the game is won or lost for the current number of items. However, it does not search all possible games of Nim from the current position.

A Minimax interpretation of our algorithm

One interesting question about our implementation of `evaluate_position` is its relationship to the Minimax approach. Our recursive algorithm uses `"won"` and `"lost"` to characterize the state of the game from the viewpoint of the player **who is currently moving**. Using Minimax, the state of the game is usually characterized by a score (either $+1$ or -1) from the viewpoint of the player **who moved first**

in the game.

We can reinterpret our implementation of `evaluate_position` as a Minimax method as follows. Assuming that I went first, consider whether it is either my turn to move or my opponent's turn to move when `evaluate_position` is called. First, consider the case when it is my turn to move. If I find a move that puts my opponent in a `"lost"` position ($+1$ score from my viewpoint), I select this move immediately since it is the best possible move for me. On the other hand, if all of my possible moves lead to a `"win"` for my opponent (-1 score from my viewpoint), this position is `"lost"` for me (-1 score from my viewpoint). In both of these case, taking the maximum of the scores returned by the recursive calls yield the appropriate score.

On the other hand, consider the case when it is my opponent's turn to move. If my opponent finds a move that puts me in a `"lost"` position (-1 score from my viewpoint), they select this move immediately since it is the best possible move for them. On the other hand, if all of their possible moves lead to a `"win"` for me ($+1$ score from my viewpoint), this position is `"lost"` for them ($+1$ score from my viewpoint). In both of these case, taking the minimum of the scores returned by the recursive calls yield the appropriate score.

With this interpretation, our recursive solution uses the move with maximal score when it is my turn to play and uses the move with minimal score when it is my opponent's turn to play. This property is characteristic of a Minimax method. Our solution has just reduced these two cases to a single case for the game of Nim.

Memoizing `evaluate_position`

The program templates includes a global counter that can be used to compute the total number of calls generated by a single initial call to `evaluate_position`. To conclude this activity, you should experiment with different initial values for `evaluate_position` and estimate the growth in the number of calls generated by a single initial call.

In practice, the number of calls grows exponentially as function of the number of initial items, making `evaluation_position` a candidate for memoization. As your last activity, we suggest that you implement a memoized version of `evaluate_position` called `evaluate_memo_position`. This function should take two inputs: the number of initial items in the heap and a dictionary that contains previously computed values for this function. If you need a hint, we suggest that you review problem #5 on homework 5.

Once you have attempted this problem, you are welcome to review our solution. As final activity, you might compare the number of calls for `evaluate_position` and `evaluate_memo_position`. You should notice a dramatic difference in the number of calls.

Created Wed4 Jun 2014 11:26 PM BST

Last Modified Mon 21 Jul 2014 5:08 AM BST