Optimal Strategies for Two-Player Games

The goal of this project is to formalize two-player games for which optimal strategies can be computed using the classic alpha-beta algorithm[1], and to instantiate the formalization on a particular variant of the game of Nim, called the game of "Num". This project is to be carried out using the Why3 tool (version 0.85), in combination with automated provers (Alt-Ergo, CVC3, CVC4 and Z3). You may use Coq for discharging particular proof obligations, although the project can be completed without it.

The project must be done individually—team work is not allowed. In order to obtain a grade for the project, you must send an e-mail to `claude.marche@inria.fr` and `arthur.charguéraud@inria.fr`, no later than **Monday, February 16th, 2015** at 19:00 UTC+1. This e-mail should be entitled "Num", be signed with your name, and have as attachment an archive (`zip` or `tar.gz`) storing the following items:

- The source file `game.mlw`, completed with your specifications, implementations, and proofs.
- The content of the sub-directory `game` generated by Why3. In particular, this directory should contain the proof session files `why3session.mlw` and `why3shapes.gz`, and your Coq proof scripts, if any.
- A PDF document named `report.pdf` in which you report on your work.

The PDF report should consists of 4 to 8 pages and should follow the sections of the present document. For each question, detail your approach, focusing in particular on the design choices that you made regarding the implementations and specifications. In case you are not able to fully complete a definition or a proof, carefully describe which parts are missing and explain the problems that you faced. Note that the report counts for at least half of the final grade.

To get started, you need to install Why3 and the automated provers, and to download the file `game.mlw`, which contains the template that you need to complete. Both the installation procedure and the file may be found on the webpage of the course.[2]

1 Two-player games

The two players are called *White* and *Black*. They play one after the other, until the game ends. White plays first. The game can reach various configurations, which we call *positions*, starting from the initial position.

1.1 A Variant of Nim Games: the "Num" Game

A classic version of the *Nim games*[3] plays as follows: given a set of identical objects (e.g., 21 of them), each player can in turn pick either one, two or three objects; the player who manages to pick the last object wins. In this project, we consider a variant that we call the "Num" game. This game is played with a sequence of numbers. At his turn, a player may take either the first number or the first two numbers from the head of the sequence. The game ends as soon as there is zero or one number left in the sequence. At this point, the player whose numbers have the greatest sum wins the game.

For example, consider the following sequence of numbers.

$$1, 10, 2, 100, 3$$

If white takes the first two numbers (1 and 10), and Black then take the next two (2 and 100), then the game ends with Black winning 102 against 11. In this case, we say that the final *score* is -91 (i.e. $11 - 102$). As another example, if White takes only the first number (1), then whatever the next move of Black, White is able to pick up the number 100 and therefore White ultimately wins the game.

[1] http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

[2] https://www.lri.fr/~marche/MPRI-2-36-1/

[3] (http://en.wikipedia.org/wiki/Nim)

1.2 General Description of Two-Player Games

We start by giving a general interface for two-player games, although, for the sake of simplicity, we will assume that there are always exactly 2 possible moves from every non-final position. The module `Num` of the file `game.mlw` provides definitions that implement this interface for the game of Num.

- The type `player` represents players: either White or Black.
- The type `position` represents a configuration of the game. In particular, for a position `p`, the field `p.turn` indicates which is the next player to play, and the field `p.score` gives the score of the position, the score being defined as the points of White minus the points of Black.
- The predicate `inv` is an invariant that constraints reachable positions.
- The predicate `is_final_position` characterizes positions for which the game has ended.
- The type `move` is used to represent the possible moves that a player can make on a given position.
- The function `next_positions` computes, given a position, the set of possible moves, together with the corresponding positions that these moves would lead to.

1.3 Specification and Verification of Move-Related Functions

We are interested in specifying functions for manipulating moves in a way that is abstract with respect to the game, that is, using only the logical definitions introduced in the previous section. In particular, the constructors `PickOne` and `PickTwo`, specific to the Num game, should not appear in the specifications.

We consider two functions: `legal_moves` and `do_move`. Given a position, the function `legal_moves` returns the pair of the two possible moves from this position, or raises the exception `GameEnded` if the position is final. Given a position and given a move, the function `do_move` returns the position obtained by playing this move from this position. The function `do_move` should not be called on final positions.

1. *Give a pre-condition and a post-condition to the function `legal_moves`.*
2. *Prove that the implementation of `legal_moves` satisfies your specification.*
3. *Give a pre-condition and a post-condition to the function `do_move`.*
4. *Prove that the implementation of `do_move` satisfies your specification.*

2 Search for Optimal Moves: Generic Approach

2.1 Tree of Moves, Minimax Values, and Optimal Strategies

Given a position in a two-player game, we can draw the set of all reachable positions as a tree. Figure 1 shows the game tree of the Num game associated with our example sequence $(1, 10, 2, 100, 3)$. Each box corresponds to a position, and contains the remaining numbers from the sequence at this point in the game. Light-gray boxes indicate White's turn to play, whereas dark-gray boxes indicate Black's turn. The arrows correspond to the possible moves.

Each box is decorated with two values, one on each side. The value on the left-hand side of each box corresponds to the score of the corresponding position. These values can be computed from top to bottom. The value on the right-hand side of each box corresponds to the *minimax* value of the corresponding position: it is equal to the maximal final score that can be achieved when starting from the position.

The minimax values can be computed from bottom to top. For a final position (i.e. a leaf), the minimax value is equal to the score of the position. For a position at which White plays, the minimax value is equal to the *maximum* of the minimax values of the two children boxes. Symmetrically, for a position at which Black plays, the minimax value is equal to the *minimum* of the minimax values of the two children boxes. Indeed, during the game, White's goal is to maximize the score whereas Black's goal is to minimize it.

The minimax value associated with the root of the tree corresponds to the final score obtained when both players play optimally. The optimal sequence of moves can be viewed by going down the tree, playing

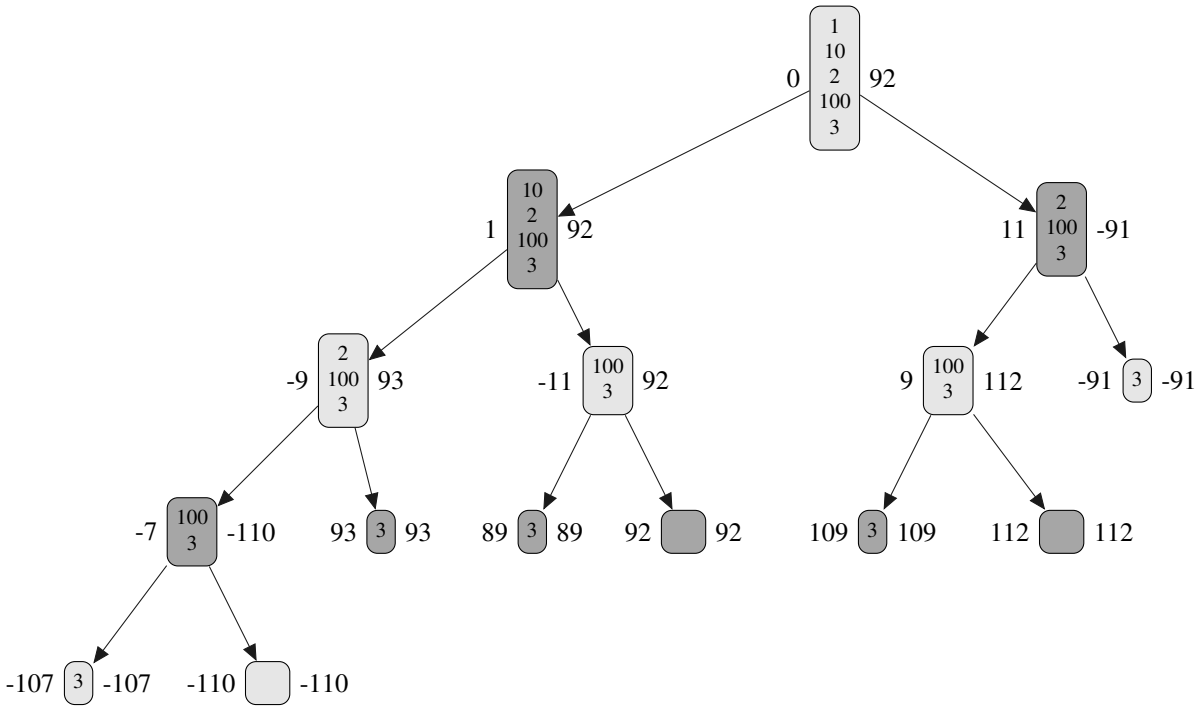

Figure 1: The complete minimax tree for the Num game on the sequence $1, 10, 2, 100, 3$.

at each position the move that reaches a children box whose minimax value is equal to that of the current position. For example, on Figure 1, the optimal moves correspond to the path made of the boxes labelled 92. White takes one number from the sequence, then Black takes two, then White takes the last two.

Unlike the Num game, two-player games in general may allow for infinite games. For this reason, the general definition of minimax values is parameterized by a depth parameter, which bounds how far the tree should be developed. If, at the depth considered, the game has not terminated, then its minimax value is defined to be the score of the current position. In the example, the minimax value of the root at depth 2 is -11, and its minimax value at depth 3 or more is 92.

5. *Complete the axioms that define the logical function* `minimax`*.*
6. *Complete the assertions in the test function* `test1`*, and prove them. Comment on the time needed by the provers in function of the depth parameter of* `minimax`*.*

2.2 Alpha-Beta Algorithm

The alpha-beta algorithm is able to compute the minimax value of a position at a given depth with necessarily exploring all the nodes in the tree. Given a position p and a depth d, and given two bounds α and β, the algorithm computes the minimax value m of position p at depth d, *within* the interval $]\alpha, \beta[$. This means that if the actual value m is less than or equal to α, then returning any value smaller than or equal to α is acceptable. Symmetrically, if m is greater than or equal to β, then returning any value greater than or equal to β is acceptable. As first approximation, the alpha-beta algorithm can be implemented as follows.

```
alpha_beta (alpha beta:int) (p:position) (d:int) : int =
  if d = 0 or (legal_moves p) = ∅: return p.score
  if (legal_moves p) = {m1, m2}:
    let v1 = alpha_beta alpha beta (do_move p m1) (d-1)
    if turn is White:
       if v1 >= beta then return v1       (* no need to explore second branch *)
       let v2 = alpha_beta (max alpha v1) beta (do_move p m2) (d-1)
       return (max v1 v2)
    if turn is Black:
```

```
if alpha >= v1 then return v1      (* no need to explore second branch *)
let v2 = alpha_beta alpha (min v1 beta) (do_move p m2) (d-1)
return (min v1 v2)
```

The actual minimax value for a given position and a given depth can be computed by providing to `alpha_beta` the arguments $-\infty$ for α and $+\infty$ for β. In the code given, the possibly-infinite values α and β are represent as options, with `None` denoting the "infinite" values. (None indicates that no bound applies.)

7. *Give a formal specification to the alpha-beta function, specifying its value result in terms of the minimax value m, distinguishing the case where m lies in the interval $]\alpha, \beta[$ and the case it does not.*
8. *Complete the code of the `alpha_beta` function, and prove it correct with respect to your specification.*

We are now interested in programming a function that effectively chooses the best move in a given position. Assume $\{m_1, m_2\}$ to be the two possible moves. First, we compute the minimax value v of the position obtained if playing move m_1. If White is playing, then we call `alpha_beta` with $\alpha = v$ and $\beta = v + 1$. This suffices to decide which move is the best. Symmetrically, if Black is playing, then we call `alpha_beta` with $\alpha = v - 1$ and $\beta = v$, and this suffices to find the best move. The function `best_move`, given a position and a depth, should return the best move from this position. The function should raise the exception `GameEnded` if there are no legal moves.

9. *Give a specification for the function `best_move`.*
10. *Complete the implementation for `best_move`, and prove it correct with respect to your specification.*

3 An Optimal Dynamic Algorithm for the Num Game

We now consider an efficient algorithm for computing optimal strategies for the Num game. In this section, implementations and specifications will refer directly to the sequence of numbers associated with the game.

3.1 Independence With Respect to Past Moves

An important property of the game of Num is that, for any given position, the best move depends only on the sequence of numbers remaining to take; it does not depend on the player in turn, nor on the scores accumulated so far. This property can be formalized as follows: the minimax value of a position p at depth d is equal to the score of the position p plus the minimax value of the same position with the score set to 0.

11. *State the property above as a lemma.*
12. *Prove this lemma by induction on the depth, either using a lemma function or using Coq.*

3.2 Dynamic Programming Algorithm

The property above leads to an efficient *dynamic programming* algorithm for computing optimal scores. Let s be the initial sequence and n be its length. We build an array a of size $n + 1$, such that $a[i]$ stores the best score that the player in turn can obtain when playing on the subsequence of s starting at index i. The table a can easily be filled backwards, as follows. For terminal positions, we let $a[n - 1] = a[n] = 0$. Otherwise, $a[i]$ can be expressed as a simple formula depending on $a[i + 1]$, $a[i + 2]$, $s[i]$ and $s[i + 1]$.

13. *Implement the algorithm described above as a function called `dynamic`, which takes as argument the initial sequence and returns the optimal score, i.e. the value $a[0]$.*
14. *State an invariant `dynamic_inv` relating, for any valid index i, the values $a[i]$ with the minimax value of any position whose current index is i.*
15. *State and prove a post-condition for the function `dynamic`, asserting that the return value is the same as the minimax value of the initial position associated with the sequence s.*
16. **Bonus** *Implement and prove correct an optimized version of this algorithm that requires only constant space usage. (The input array s should not be modified.)*