Two-Player Games, Alpha-Beta and Dynamic Programming

The goal of this project is to formalize two-player games and algorithms for finding best moves, such as the classical alpha-beta algorithm[1], and to instantiate the formalization on a particular variant of the game of Nim, called the game of "Min".

This project is to be carried out using the Why3 tool (version 1.7.x), in combination with automated provers (Alt-Ergo 2.5.x, CVC4 1.8, CVC5 1.0.x and Z3 4.12.x or higher). You can use other automatic provers or versions if you want, if they are freely available and recognized by Why3. You may use Coq for discharging particular proof obligations, although the project can be completed without it. The installation procedure may be found on the web page of the course at URL `https://marche.gitlabpages.inria.fr/lecture-deductive-verif/install.html`.

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 `Jean-Marie.Madiot@inria.fr`, no later than **Thursday, February 8th, 2024** at 23:00 UTC+1. This e-mail should be entitled "MPRI project 2-36-1", be signed with your name, and have as attachment an archive (`zip` or `tar.gz`) storing the following items:

- The proposed canvas file `min_game.mlw` completed by yours.
- The content of the sub-directory `min_game` generated by Why3. In particular, this directory should contain session files `why3session.xml` and `why3shapes.gz`, and Coq proof scripts, if any.
- A PDF document named `report.pdf` in which you report on your work. The contents of this report counts for at least half of your grade for the project.

The report must be written in French or English, and should typically consist of 3 to 6 pages. The structure should follow the sections and the questions 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 particular, loop invariants and assertions that you added should be explained in your report: what they mean and how they help to complete the proof.

A typical answer to a question or step would be: *"For this function, I propose the following implementation: [give pseudo-code]. The contract of this function is [give a copy-paste of the contract]. It captures the fact that [rephrase the contract in natural language]. To prove this code correct, I need to add extra annotations [give the loop invariants, etc.] capturing that [rephrase the annotations in english]. This invariant is initially true because [explain]. It is preserved at each iteration because [explain]. The post-condition then follows because [explain]."*

The reader of your report should be convinced at each step that the contracts are the right ones, and should be able to understand why your program is correct, *e.g.*, why a loop invariant is initially true, why it is preserved, and why it suffices to establish the post-condition. It is legitimate to copy-paste parts of your Why3 code in the report, yet you should only copy the most relevant parts, not all of your code. In case you are not able to fully complete a definition or a proof, you should carefully describe which parts are missing and explain the problems that you faced.

In addition, your report should contain a conclusion, providing general feedback about your work: how easy or how hard it was, what were the major difficulties, was there any unexpected result, and any other information that you think is important to consider for the evaluation of the work you did.

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

1 Two-player games

A two-player game is any kind of game, like say chess, were each of the two players, called here *Alice* and *Bob*, play in turn a *move* that changes the current game *configuration*. They start from a given *initial* configuration and Alice plays first by convention. The game continues until it reaches some final configuration, where some rule tells which player wins.

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

Nim games[2] are a family of two-player games. A classical version runs as follows: given a set of identical objects (say, 21 matches), 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 of the above we call the "Min" game. This game is played with a sequence of numbers. At her/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 picked have the smallest sum wins the game.

For example, consider the following sequence of numbers.

$$20, 17, 24, -30, 27$$

If Alice takes the first two numbers (20 and 17), and Bob then takes the next two (24 and -30), then the game ends with Bob winning -6 against 37. In this case, we say that the final *score* is 43 (i.e. $(37 - (-6))$. The score is by definition the difference between Alice's numbers and Bob's numbers, so that Alice wants to minimize the score whereas Bob wants to maximize it. As another example of run of this game, assume Alice takes only the first number (20). Then, whatever the next move of Bob, Alice is able to pick up the number -30 and therefore Alice ultimately wins the game, either with final score -51 (i.e. $(20 - 30) - (17 + 24)$) or -3 (i.e. $(20 + 24 - 30) - 17$). In other words, the best strategy for Alice is to pick only the first number for her first move.

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 two possible moves from every non-final configuration. The module `MinGame` of the file `min_game.mlw` provides definitions that implement this interface for the game of Min.

- The type `player` represents players: either Alice or Bob.
- The type `config` represents a configuration of the game. In particular, for a configuration `c`, the field `c.turn` indicates which is the next player to play.

1. *The predicate* `inv` *is an invariant that constraints the set of well-formed configurations. Fill-in its definition (your answer to this question might be definitive only after solving the other questions).*

1.3 Specification and Verification of Move-Related Functions

The type `move` is used to represent the possible moves that a player can make on a given configuration.

The function `legal_moves` (defined both in logic and as a program using **`let function`**) is supposed to return the set of possible moves in a given configuration. There are 0 or 2 of them. A final configuration of a game is by definition a configuration where there are no more moves.

2. *Fill-in the definition of* `legal_moves`

The function `do_move`, given a configuration c and a move m, returns the configuration obtained by playing the move m from configuration c. The function `do_move` should not be called on final configurations.

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

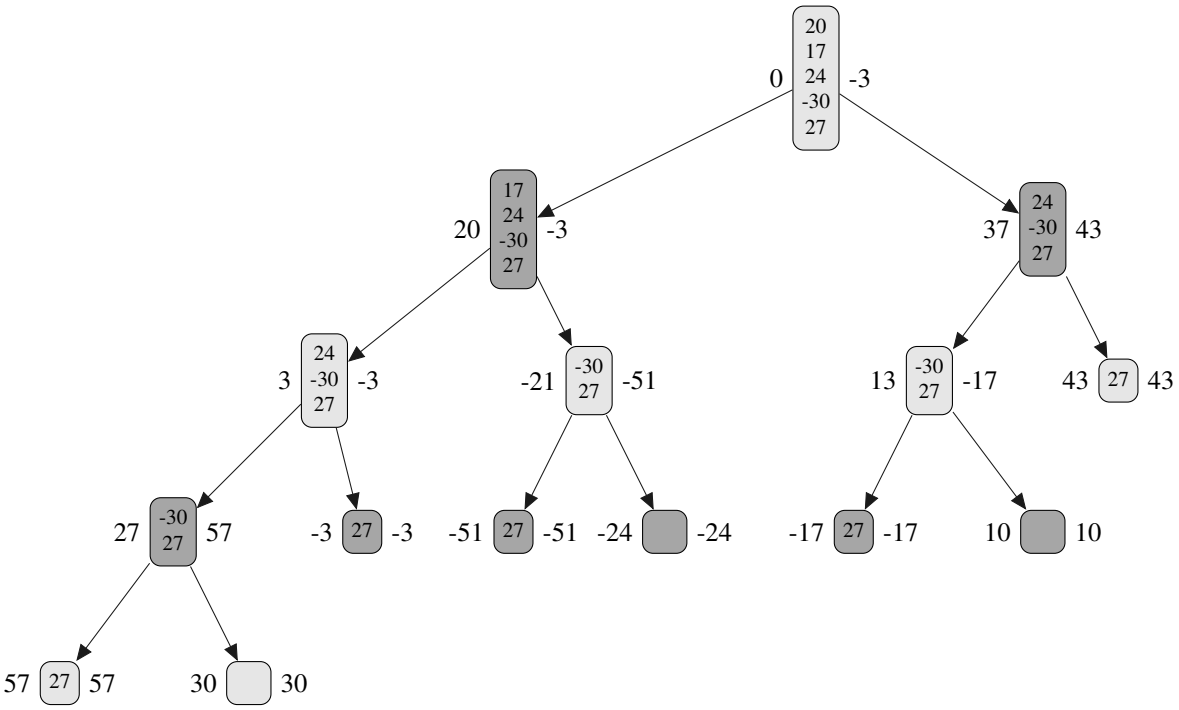

Figure 1: The complete minimax tree for the Min game on the sequence $20, 17, 24, -30, 27$.

3. *Add a pre-condition to the contract of function* `do_move` *to forbid calling it on final configurations. Complete the body of that function according to the rules of the Min game.*
4. *The program* `config0` *is a simple test for your definitions. You can run it using the Why3* interpreter*, using command* `why3 execute min_game.mlw --use MinGame "config0 ()"`*. Alternatively, you can use the command* `make tests` *from the given* `Makefile`*.*

2 Search for Optimal Moves: Generic Approach

In this section, we are interested in specifying and proving programs for searching optimal strategies for a two-player game, in a way that is abstract with respect to the game. In particular, the specifications and definitions of functions below should use only the type `config` and the functions `legal_moves` and `do_move`, and never the items specific to Min game, such as the constructors `PickOne` and `PickTwo` for moves, or the fields of the `config` type except the `turn` field.

2.1 Tree of Moves, Minimax Values, and Optimal Strategies

Given a configuration in a two-player game, we can draw the set of all reachable configurations as a tree. Figure 1 shows the game tree of the Min game associated with our sample sequence $(20, 17, 24, -30, 27)$. Each box corresponds to a configuration, and contains the remaining numbers from the sequence at this point in the game. Light-gray boxes indicate Alice's turn to play, whereas dark-gray boxes indicate Bob's turn. The arrows correspond to the possible moves: pick one number for arrows going on the left, pick two numbers for arrows going on the right.

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 configuration. 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 configuration: it is equal to the optimal final score that can be achieved when starting from the configuration.

The minimax values can be computed from bottom to top. For a final configuration (i.e. a leaf of the tree), the minimax value is equal to the score of the configuration. For a configuration at which Alice plays, the minimax value is equal to the *minimum* of the minimax values of the two children boxes. Symmetrically,

for a configuration at which Bob plays, the minimax value is equal to the *maximum* of the minimax values of the two children boxes. Remind that, during the game, Alice's goal is to minimize the score whereas Bob's goal is to maximize 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 at each configuration the move that reaches a children box whose minimax value is equal to that of the current configuration. For example, on Figure 1, the optimal moves correspond to the path made of the boxes labelled -3. Alice takes one number from the sequence, then Bob takes one, then Alice takes the last two.

Unlike the specific case of Min 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 configuration. In the example, the minimax value of the root at depth 2 is 3, and its minimax value at depth 3 or more is -3.

5. *Complete the recursive definition of the logical function* `minimax` *that defines the minimax value of a given configuration at a given depth. Prove its termination.*
6. *Test your definition of* `minimax` *with the Why3 interpreter, by running the* `test0` *program. Test also your definition by* proving *the* `test_minimax` *program. Comment on the results.*

2.2 The Classical Alpha-Beta Algorithm

The alpha-beta algorithm is able to compute the minimax value of a configuration at a given depth without necessarily exploring all the nodes in the tree. Given a configuration c and a depth d, and given two bounds α and β, the algorithm computes the minimax value m of configuration c at depth d, *within* the interval $]\alpha, \beta[$. This means that if the actual minimax 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. The alpha-beta algorithm can be described using the following rules. When there are no legal moves in c then `alpha_beta`(α,β,c,d)=`config_value`(c). Otherwise, when the legal moves are m_1 and m_2, then let v_1 be the evaluation of the m_1 move, that is `alpha_beta`(α,β,`do_move`(c,m_1),$d-1$). Assuming the turn is to Alice:

- if $v_1 \leq \alpha$ we don't need to explore the other branch and return v_1
- Otherwise we can compute v_2=`alpha_beta`(α,min(v_1,β),`do_move`(c,m_2),$d-1$) and return the minimum of v_1 and v_2

If it is to Bob to play, the rules are similar by exchanging the role of α and β and replacing minimum by maximum.

The actual minimax value for a given configuration 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.

7. *Give a formal specification to the alpha-beta function, specifying its result value in terms of the minimax value m, distinguishing the case where m lies in the interval $]\alpha, \beta[$ and the case it does not. Explain carefully why you pretend your specification is the one intended for that algorithm.*
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 configuration. Assume m_1 and m_2 are the two possible moves. First, we compute (using `alpha_beta`) the minimax value v of the configuration obtained when playing move m_1. We then consider the configuration c_2 obtained from playing m_2. If Alice is playing, then we call `alpha_beta` on c_2 with $\alpha = v$ and $\beta = v+1$. This suffices to decide which move is the best. Symmetrically, if Bob is playing, then we call `alpha_beta` on c_2 on a small interval sufficient to find the best move. The function `best_move`, given a configuration and a depth, should return the best move from this configuration. 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.*
11. *Test your program by running the interpreter on the program* `test_alpha_beta`*. Comment on the results.*
12. *Test your program by compiling and running tests from the given file* `test_min_game.ml`*, using* `make test_min_game`*. Comment on the results.*

3 An Optimal Dynamic Algorithm for the Min Game

We now consider an efficient algorithm for computing optimal strategies for the Min game. We are no longer generic with respect to an arbitrary two-player 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 Min is that, for any given configuration, 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 is indeed a consequence of a property of the minimax: the minimax value of a configuration c at some depth d is equal to the score of the configuration c plus the minimax value of the same configuration with the score set to 0.

13. *State the property above on minimax as a lemma, under the form of a lemma function* `minimax_independence` *with an appropriate contract. Prove this lemma.*

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, using the following rules. For terminal configurations, we let $a[n - 1] = a[n] = 0$ since no more moves can be done. Otherwise, $a[i]$ can be expressed as a simple formula, to be determined, depending on $a[i + 1]$, $a[i + 2]$, $s[i]$ and $s[i + 1]$.

14. *Propose a formula relating* $a[i]$ *to* $a[i + 1]$, $a[i + 2]$, $s[i]$ *and* $s[i + 1]$.
15. *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* $a[0]$*. Test your program by running the* `test` *function with the interpreter. Additionally, you can test your program by running* `make test_min_game`*. Comment on the results.*
16. *Complete the definition of the predicate* `dynamic_inv` *relating, for any valid index* i*, the values* $a[i]$ *with the minimax value of any configuration whose current index is* i*.*
17. *Complete the post-condition of the program* `dynamic`*, stating that the return value is the same as the minimax value of the initial configuration associated with the sequence* s*. Prove the program.*
18. *Implement and prove correct an optimized version* `dynamic_opt` *of the previous program, with a code that do not allocate an array whose size is linear in function of the size of the input stack, but only uses a constant memory space. The input array* s *should not be modified.*

4 Conclusions

Don't forget to end your report with a conclusion that summarizes your achievements, explain the issues you couldn't solve if any, and comment about what you learned when doing this project.