The Why and How of Deductive Program Verification

Jean-Christophe Filliâtre

Marktoberdorf Summer School 2026

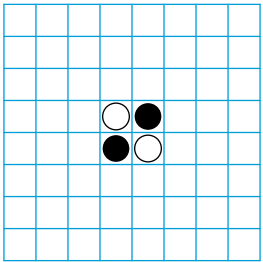
lecture 4

if we need some verified piece of software, it is much easier to implement it in a programming language that was designed with verification in mind

Why3 can also be used this way

let us implement a two-player game, of the kind human vs computer, where the machine implements a verified alpha-beta pruning

for instance to play Othello

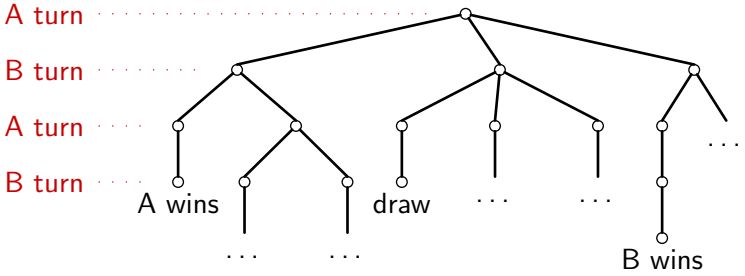


players A and B play in turn

for each game position,

- either the game is over
- or there is a finite number of legal moves

we can see the game as a tree

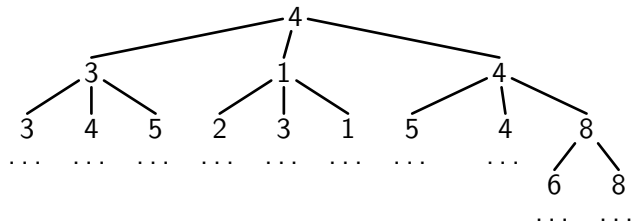


if the tree is finite, and not too big (e.g. tic-tac-toe),
we can evaluate each position (as winning, losing, or draw)
and derive from that an optimal strategy

for a realistic game, we can't do that

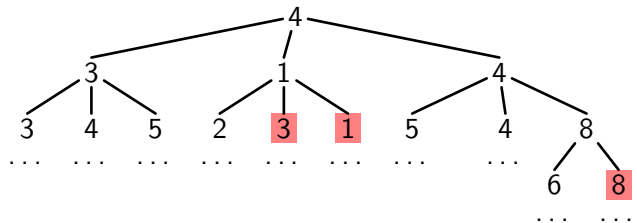
we stop our descent at some point, and heuristically evaluate the position

player A is maximizing values and assumes player B to minimize them
(all values here from the point of view of A)



this is called min-max algorithm

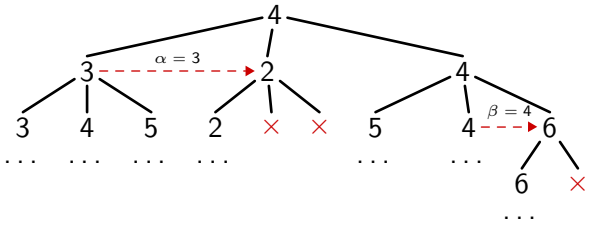
player A is maximizing values and assumes player B to minimize them
(all values here from the point of view of A)



the subtrees in red can be safely ignored

principle: maintain an interval $[\alpha, \beta]$ out of which
there is no need to compute the value of the position

see Donald E. Knuth, Ronald W. Moore
An analysis of alpha-beta pruning
Artificial Intelligence, 6(4), 1975


4
3
$\alpha = 3$
2
4
3
4
5
2
×
×
5
4
$\beta = 4$
6
6
×

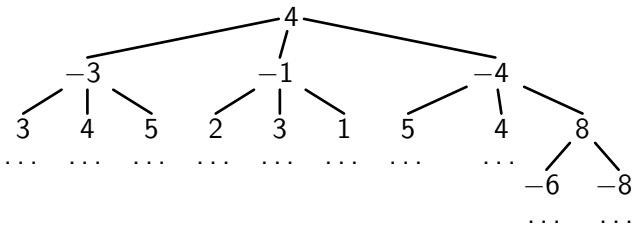
Theorem

Alpha-beta pruning computes the same value as min-max for the root of the tree.

let us verify this with Why3!

in practice, it is convenient

- to evaluate a position from the point of view of the player who is going to play
- to assume the evaluation by the other player to be the negation of our evaluation



this way, we don't alternate between max and min; we just use max

demo

```
type position

val constant inf: int
  ensures { 0 < result }

val function eval_position (_: position) : (v: int)
  ensures { -inf <= v <= inf }
```

```
type game = Node position (list game)
```

the specification is the min-max evaluation

```
let rec function eval (g: game) : int
= match g with
  | Node p Nil -> eval_position p
  | Node _ gl  -> eval_list gl
  end
with function eval_list (gl: list game) : int
= match gl with
  | Nil       -> -inf
  | Cons g gl -> max (- eval g) (eval_list gl)
  end
```

we intend to show that alpha-beta is sound wrt min-max

```
let alpha_beta (g: game) : (v: int)
  ensures { v = eval g }
= ...
```

```
let rec alpha_beta_g (a b: int) (g: game) : (v: int)
= match g with
  | Node p Nil -> eval_position p
  | Node _ gl  -> alpha_beta_l a b gl
  end
with alpha_beta_l (a b: int) (gl: list game) : (v: int)
= match gl with
  | Nil ->
      a
  | Cons g gl ->
      let a = max a (- alpha_beta_g (-b) (-a) g) in
      if a >= b then a else alpha_beta_l a b gl
  end

let alpha_beta (g: game) : (v: int)
= alpha_beta_g (-inf) inf g
```

```
let rec alpha_beta_g (a b: int) (g: game) : (v: int)
  requires { -inf <= a < b <= inf }
  ensures  { -inf <= v <= inf }
  ensures  { eval g < a -> v <= a }
  ensures  { a <= eval g <= b -> v = eval g }
  ensures  { b < eval g -> v >= b }
= match g with
  | Node p Nil -> eval_position p
  | Node _ gl  -> alpha_beta_l a b gl
  end
```

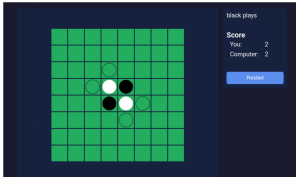
```
with alpha_beta_l (a b: int) (gl: list game) : (v: int)
  requires { -inf <= a < b <= inf }
  ensures  { a <= v <= inf }
  ensures  { if eval_list gl <= b
             then v = max a (eval_list gl) else b <= v }
= match gl with
  | Nil ->
      a
  | Cons g gl ->
      let a = max a (- alpha_beta_g (-b) (-a) g) in
      if a >= b then a else alpha_beta_l a b gl
  end
```

once verified, the Why3 code can be translated to OCaml, automatically

```
why3 extract -D ocaml64 alpha_beta.AlphaBeta
```

linked with unverified OCaml code (game logic, user interface, etc.)

and then even translated to JavaScript



play online

the web site points to two larger case studies

- a nontrivial subset of GMP's algorithms
  (Melquiond & Rieu-Helft, 2020)

  verified with Why3, then translated to C

- a checker for a railway information system
  (Boyer, Canva, Manighetti & Marché, 2025)

  verified with Why3, then translated to OCaml

conclusion

deductive verification means designing

- programming languages
  - semantics
  - type systems
- specification languages
  - logic
- libraries
  - programming libraries
  - logical theories

deductive verification means implementing

- compilers
- logical transformations
- a lot of plumbing

is a lot of fun!