Level 2

Optimizing Minimax

Riccardo Pucella

September 30, 2014

Summary

Round-based two-player games

- Human (or computer) versus computer

Game trees

- Records possible plays
- Decide best move from any position

Minimax algorithm

- utility of terminal states
- propagate up the game tree

Summary

Round-based two-player games

- Human (or computer) versus computer

Game trees

- Records possible plays
- Decide best move from a

Minimax algorithm

- utility of terminal states
- propagate up the game tree

Maximizing for A

Minimizing for B

Big problem: size!

Minimax works great, but it's slow

10^4 nodes/sec (not unreasonable, not great)

- 100 secs (~2 minutes): 10^6 nodes
- 1000 secs (~15 minutes): 10^7 nodes
- 10000 secs (~2.5 hours): 10^8 nodes
- . . .

4x4 Tic-Tac-Toe ~ 10^{12} nodes

Four solutions

In increasing order of effectiveness

- Local optimizations
- Caching
- Pruning
- Giving up on finding best move

Solution 1: Local Optimizations

Pareto Principle (variant):

90% of the time is spent in 10% of the code

Find that 10%

Make sure it run fast

Optimize the deeper into the loops

(loops may be implicit via recursion)

Profiling

The best way to determine where your code is spending time is to run a profiler

For Python: `cProfile` module

(There are others. Note that a profiler has overhead. `cProfile` has low overhead.)

Example: counting primes

```
def prime (n):
    for i in range(2,n):
        if i > math.sqrt(n):
            return True
        if n % i == 0:
            return False
    return True

def main ():
    count = 0
    for i in range(20000):
        if prime(i):
            count += 1
    print count
```

Example: counting primes

```
>>> import primes

>>> import cProfile

>>> cProfile.run('primes.main()')
2264
         343540 function calls in 3.380 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.380    3.380 <string>:1(<module>)
        1    0.013    0.013    3.380    3.380 primes.py:12(main)
    20000    1.535    0.000    3.366    0.000 primes.py:4(prime)
   303536    0.053    0.000    0.053    0.000 {math.sqrt}
    20001    1.778    0.000    1.778    0.000 {range}
```

Example: counting primes

```
def prime (n):
    if n % 2 == 0:
        return False
    for i in range(3,n,2):
        if i > math.sqrt(n):
            return True
        if n % i == 0:
            return False
    return True

def main ():
    ...
```

Example: counting primes

```
>>> import primes

>>> import cProfile

>>> cProfile.run('primes.main()')
12262
         177336 function calls in 0.920 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.920    0.920 <string>:1(<module>)
        1    0.011    0.011    0.920    0.920 primes.py:14(main)
    20000    0.431    0.000    0.907    0.000 primes.py:4(prime)
   147332    0.025    0.000    0.025    0.000 {math.sqrt}
    10001    0.452    0.000    0.452    0.000 {range}
```

Example: counting primes

```
def prime (n):
    if n % 2 == 0:
        return False
    for i in range(3,n,2):
        if i*i > n:
            return True
        if n % i == 0:
            return False
    return True

def main ():
    ...
```

Example: counting primes

```
>>> import primes

>>> import cProfile

>>> cProfile.run('primes.main()')
12262
         30004 function calls in 0.840 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.840    0.840 <string>:1(<module>)
        1    0.011    0.011    0.840    0.840 primes.py:14(main)
    20000    0.387    0.000    0.828    0.000 primes.py:4(prime)
    10001    0.442    0.000    0.442    0.000 {range}
```

Example: Tic-Tac-Toe

Many functions are called at every node of the game tree:

- check if board is terminal (done)
- compute possible moves
- apply a move to the board

Let's analyze `has_win` (part of done)

Function has_win

```
# board is array of 0, 1 (for O), 10 (for X)

def has_win (board):
    for positions in WIN_SEQUENCES:
        s = sum(board[pos] for pos in positions)
        if s == 3:
            return 'O'
        if s == 30:
            return 'X'
    return False
```

```
WIN_SEQUENCES = [
    [0,1,2],
    [3,4,5],
    [6,7,8],
    [0,3,6],
    [1,4,7],
    [2,5,8],
    [0,4,8],
    [2,4,6]
]
```

Function has_win

```
# board is array of 0, 1 (for O), 10 (for X)

de

         3228033 function calls in 1.826 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    1.826    1.826 <string>:1(<module>)
   100000    0.590    0.000    1.766    0.000 done.py:166(has_win_2)
  2502424    0.561    0.000    0.561    0.000 done.py:168(<genexpr>)
        1    0.060    0.060    1.826    1.826 done.py:27(test)
   625606    0.615    0.000    1.176    0.000 {sum}
```

Function has_win

```
# board is array of 0, 1 (for O), 10 (for X)
# checks only rows that involve last move

def has_win ((board,last_move)):
    for positions in CHECKS[last_move]:
        s = sum(board[pos] for pos in positions)
        if s == 3:
            return 'O'
        if s == 30:
            return 'X'
    return False
```

```
CHECKS = {
    0: [[1,2],[3,6],[4,8]],
    1: [[0,2],[4,7]],
    2: [[0,1],[5,8],[4,6]],
    3: [[4,5],[0,6]],
    4: [[1,7],[3,5],[0,8],[2,6]],
    5: [[3,4],[2,8]],
    6: [[7,8],[0,3],[2,4]],
    7: [[6,8],[1,4]],
    8: [[6,7],[2,5],[0,4]]
}
```

Function has_win

```
# board is array of 0, 1 (for O), 10 (for X)
# checks only rows that involve last move
```

```
      1164907 function calls in 0.789 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.789    0.789 <string>:1(<module>)
   100000    0.307    0.000    0.730    0.000 done.py:175(has_win_3)
   798678    0.199    0.000    0.199    0.000 done.py:177(<genexpr>)
        1    0.058    0.058    0.789    0.789 done.py:27(test)
   266226    0.225    0.000    0.424    0.000 {sum}
```

Function has_win

```
# board is array of 0, 1 (for O), 10 (for X)

def has_win_0 (board):
    if (board[0] == board[1] and board[0] == board[2]):
        return board[0]
    if (board[3] == board[4] and board[3] == board[5]):
        return board[3]
    if (board[6] == board[7] and board[6] == board[8]):
        return board[6]
    if (board[0] == board[3] and board[0] == board[6]):
        return board[0]
    if (board[1] == board[4] and board[1] == board[7]):
        return board[1]
    if (board[2] == board[5] and board[2] == board[8]):
        return board[2]
    if (board[0] == board[4] and board[0] == board[8]):
        return board[0]
    if (board[2] == board[4] and board[2] == board[6]):
        return board[2]
    return False
```

Function has_win

```
# board is array of 0, 1 (for O), 10 (for X)

def

    return False
```

```
        100003 function calls in 0.158 seconds

  Ordered by: standard name

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       1    0.000    0.000    0.158    0.158 <string>:1(<module>)
       1    0.035    0.035    0.158    0.158 done.py:27(test)
  100000    0.123    0.000    0.123    0.000 done.py:80(has_win_0)
```

About optimizations

Effectiveness of an optimization depends on

- algorithmic choices
- data representation choices
- programming language choices
- details of the implementation of the language

In Python, copying is expensive, update is not

- In other languages, update is expensive, copying is not

Solution 2: Caching

Minimax will compute the minimax value of a board every time it encounters it during traversal of the game tree

The same board may appear as the result of a different sequence of moves

E.g.   moves 0 (for X), 1 (for O), 2 (for X)
       and 2 (for X), 1 (for O), 0 (for X)
both produce the same board

Basics of caching

Caching (or memoization) is an approach to remembering previous results of a function.

```
def foo (x):

    code for foo(x)

    return v
```

Basics of caching

Caching (or memoization) is an approach to remembering previous results of a function.

```
def foo (x):
    if x has been seen before, return value[x]
    code for foo(x)
    save value[x] = v
    return v
```

Basics of caching

Caching (or memoization) is an approach to remembering previous results of a function.

```
def foo (x):
    if x has been seen before, return value[x]
    code for foo(x)
    save value[x] = v
    return v
```

Lookups should be reasonably fast

They get performed at every node of the game tree

Basics of caching

Caching (or memoization) is an approach to remembering previous results of a function.

```
def foo (x):
    if x has been seen before, return value[x]
    code for foo(x)
    save value[x] = v
    return v
```

Data structure:
hash tables (dictionaries)

Lookups and saves:
constant time (mostly)

Minimax application

Keep a dictionary associating with seen boards their computed minimax value

Shortcut minimax computation when a board has been seen

Technical problem: arrays cannot be used as keys in Python dictionaries

- Need a way to associate with every board a key by which to refer to it in the dictionary

Caching problems

The caching table can get large

If the caching table gets too large, it can overflow memory, and then the system starts swapping to disk

- Any time advantage is lost because memory swapping is SUPER SLOW

Exploiting symmetry

The caching table can be used to remember more than just seen boards

It can be used to remember boards that have not been seen yet

Observation: two symmetrical boards must have the same minimax value

(Why?)

Exploiting symmetry

Two approaches:

1. When saving a minimax value for a board, save that minimax value for all symmetric boards *(space expensive)*

2. When looking up a board in the table, also look for any symmetry of that board *(time expensive)*

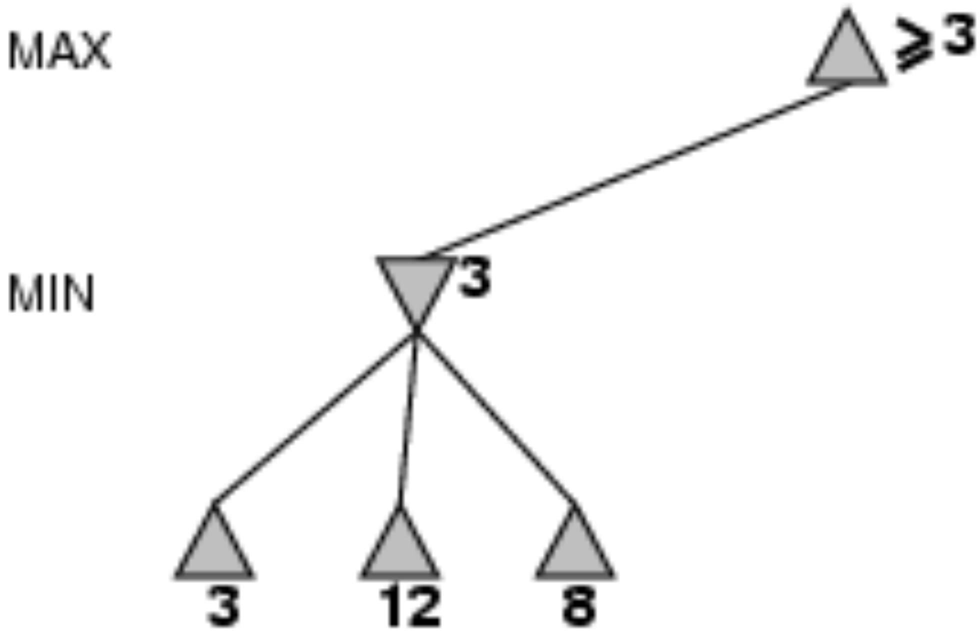
Solution 3: Pruning

When doing minimax, at a maximum node, you compute (recursively) the minimax value of the children.

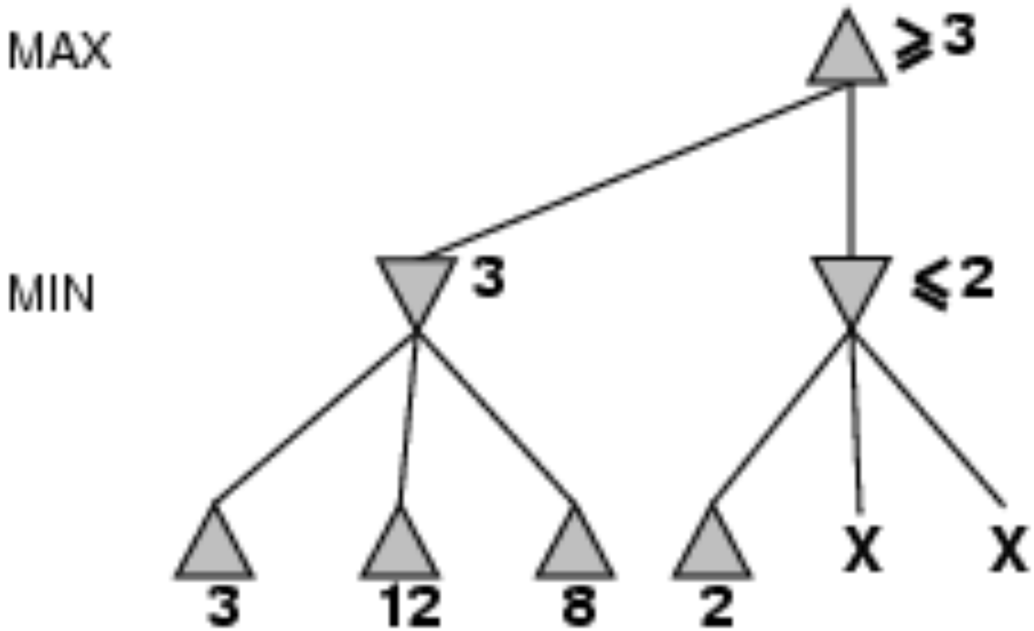
If at any point you can tell that the best value you will get for a child is less than your current max, you can get stop computing the minimax value of that child (prune the subtree)

Similarly when at a minimum node

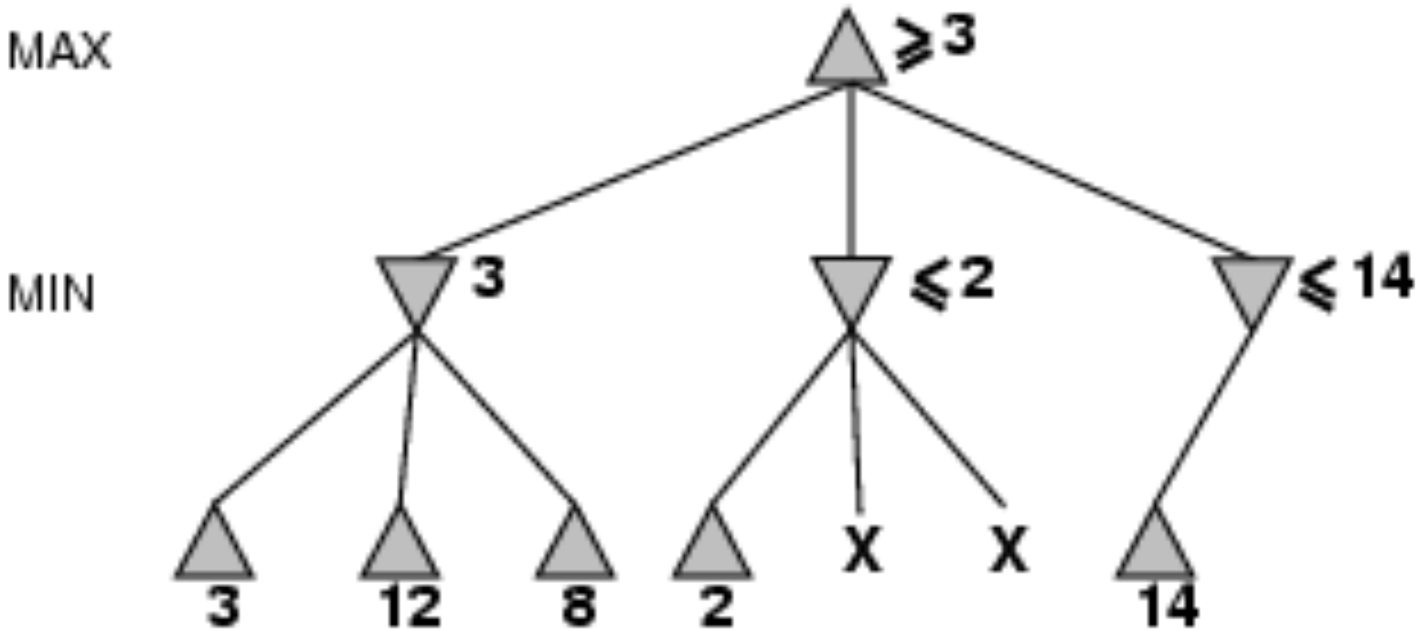
α-β pruning example



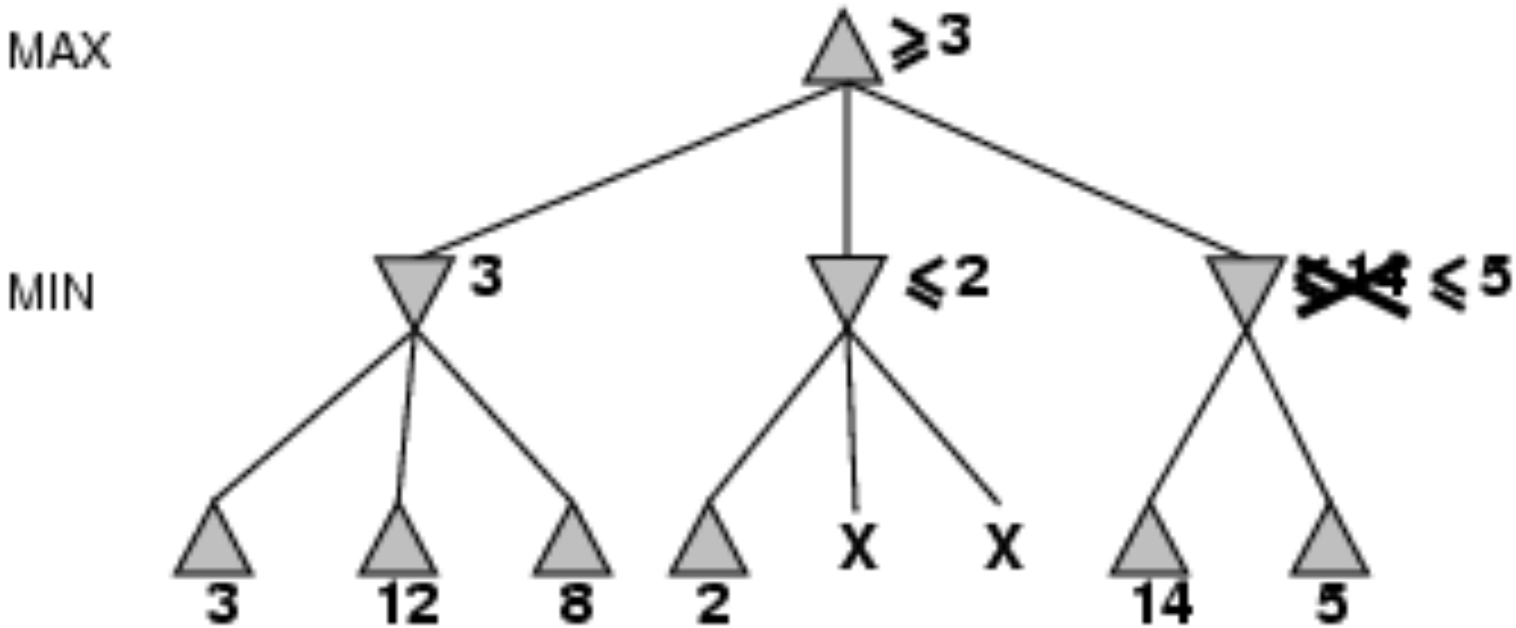
α-β pruning example



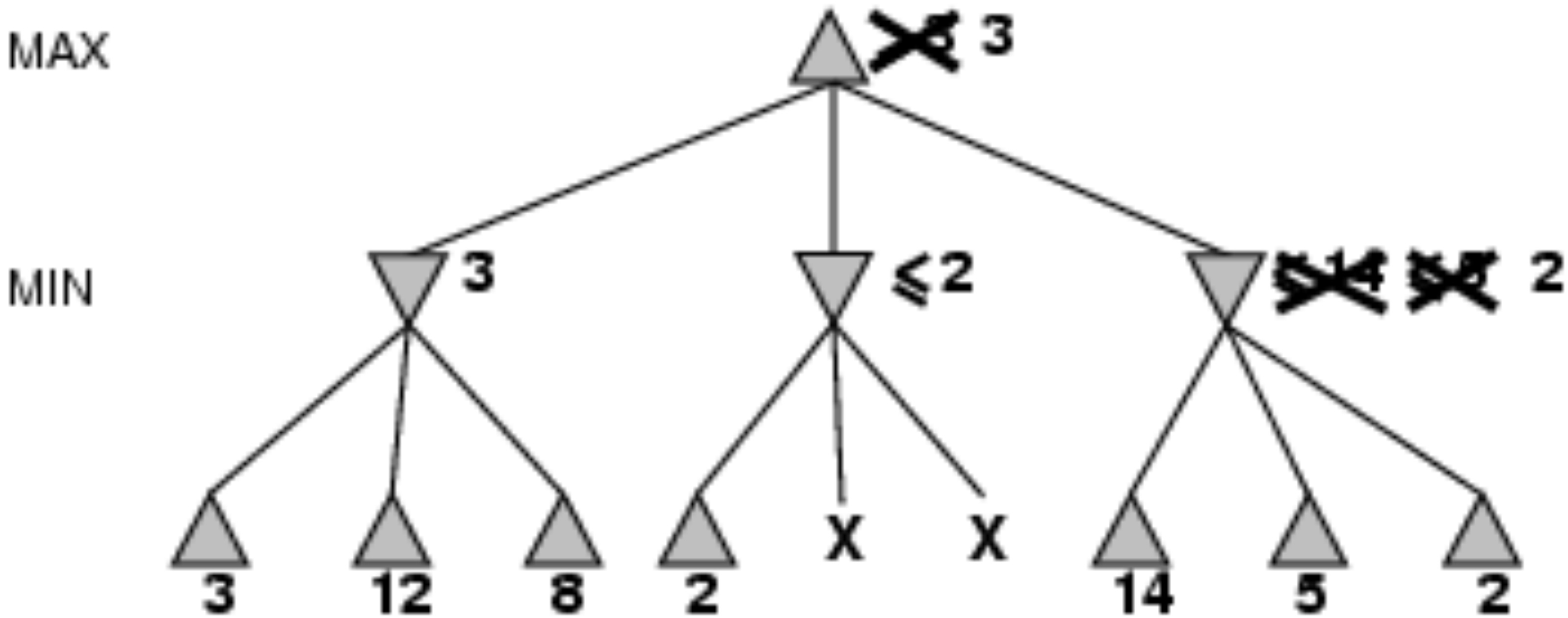
α-β pruning example



α-β pruning example



α-β pruning example



Properties of α-β pruning

Returns the same result as standard minimax

Effectiveness depends on move ordering

- Best case: can double the search depth

Can be tricky to get right in the presence of caching   (why?)

The α-β algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game

    v ← MAX-VALUE(state, −∞, +∞)
    return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state

    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

Something similar for `Min-Value`

Solution 4: Cutting-off search

For some games, none of the above is enough to make minimax manageable.

Chess, Go, Ultimate Tic-Tac-Toe.

Give up on searching the whole game tree

- Search only to a limited depth
- Possibly some branches deeper than others

What's the issue?

Evaluating nodes

Minimax works by propagating up the utility of final states

- Utility = is final state a win or not?

If you don't search to the final states, what do you propagate up?

Evaluation function:

- Associate a value that tries to capture how good the position is when cutting off

Evaluation functions

For chess, typically linear weighted sum of features

$$\text{Eval}(s) = w_1\, f_1(s) + w_2\, f_2(s) + \ldots + w_n\, f_n(s)$$

E.g., $w_1 = 9$ with

$f_1(s)$ = (# white queens) – (# black queens)

etc…

Minimax with cutoff

Minimax/cutoff is similar to Minimax:

- `Terminal?` is replaced by `Cutoff?`
- `Utility` is replaced by `Eval`

In practice, for chess, searching 10^6 moves:

$$br^{depth} = 10^6 \text{ , } br = 35 \rightarrow \text{ depth} = 4$$

4-ply lookahead is a hopeless chess player!

- 4-ply $\approx$ human novice
- 8-ply $\approx$ typical PC, human master
- 12-ply $\approx$ Deep Blue, Kasparov