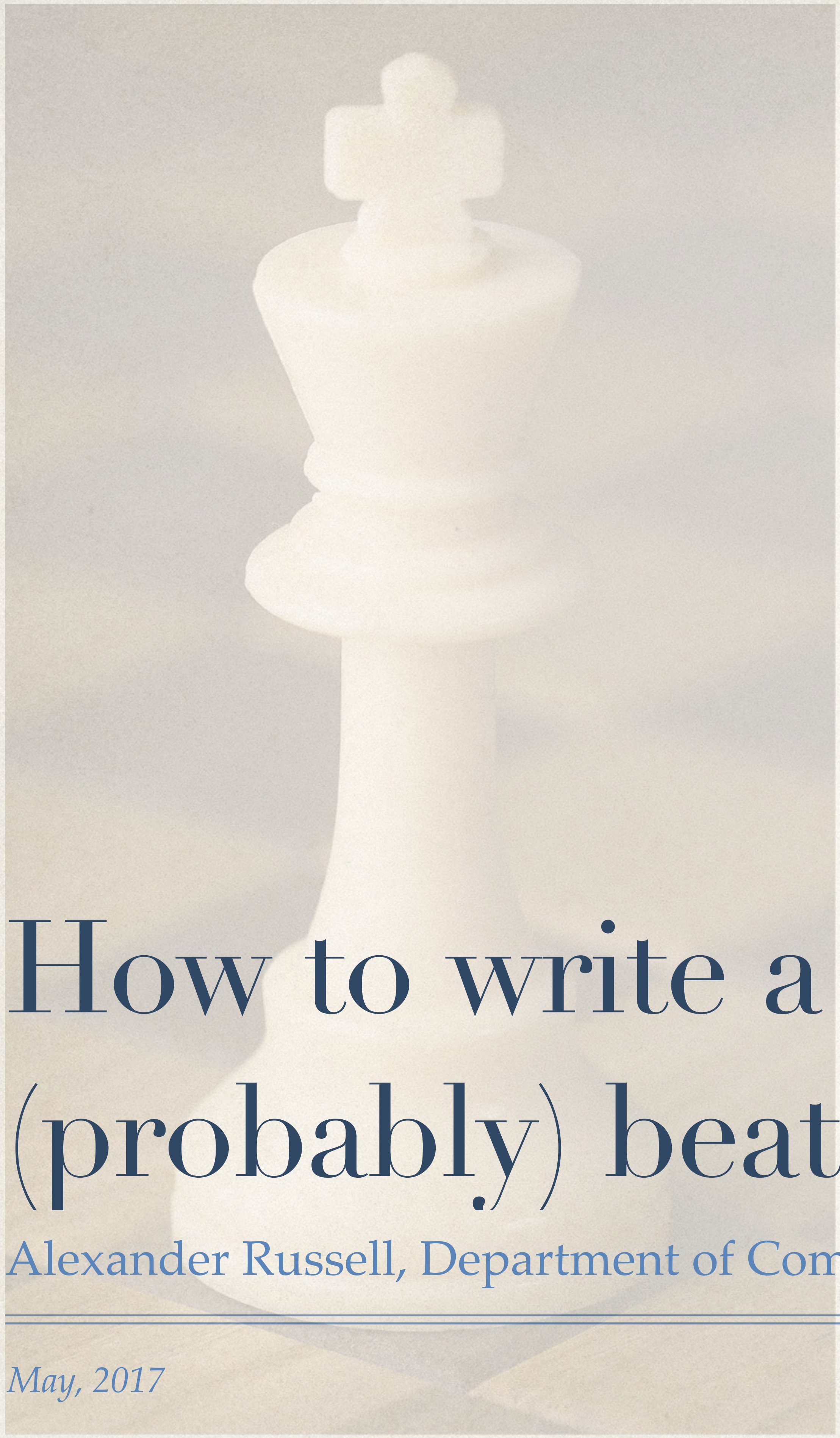


# How to write a program that can (probably) beat its author at Chess

Alexander Russell, Department of Computer Science and Engineering, UConn

*May, 2017*

# Chess, Checkers, Othello, …

- Two player, zero-sum games of full-information.
- **Zero-sum**: What's good for player 1 is bad for player 2, and *vice versa*. In particular, if player 1 wins, then player 2 loses.
- **Full information**: State of game is completely known to both players at all times. (Contrast with, e.g., Stratego.)
- **Deterministic**: No dice, spinners, coin-flips.

## Game tree search

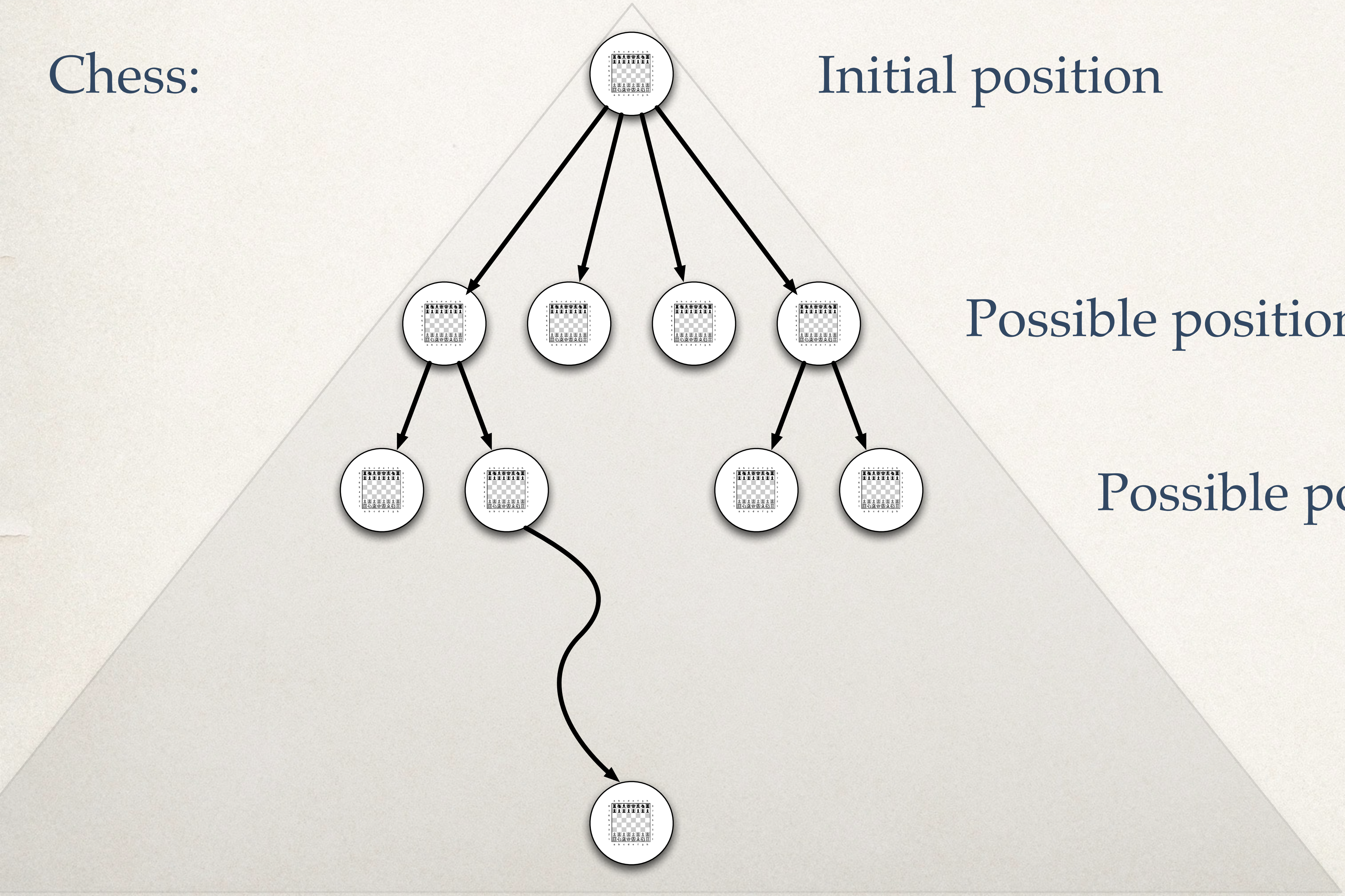
- Today: We'll discuss...
  - **Game tree search**: a basic computational paradigm which can be used to tackle all such games (and other phenomena as well). The foundation of most competitive game playing software.
  - Makes rigorous the notion of "look-ahead."
  - Can be naturally optimized and parallelized.
  - Often coupled with fancy learning algorithms.

## With some extra work & optimization

- Current world-champion for Reversi & Checkers is a computer. In 1980, *The Moor* beat the reigning Reversi world-champion. In 1996, *Chinook* won the USA National Checkers Tournament by a wide margin.
- Current world-champion for Chess is a toss-up (though things don't look too rosy for the humans).
- Current world-champion for Go is a toss-up; there has been a major advance in computer performance over the last few years.

# The big idea: Game Tree

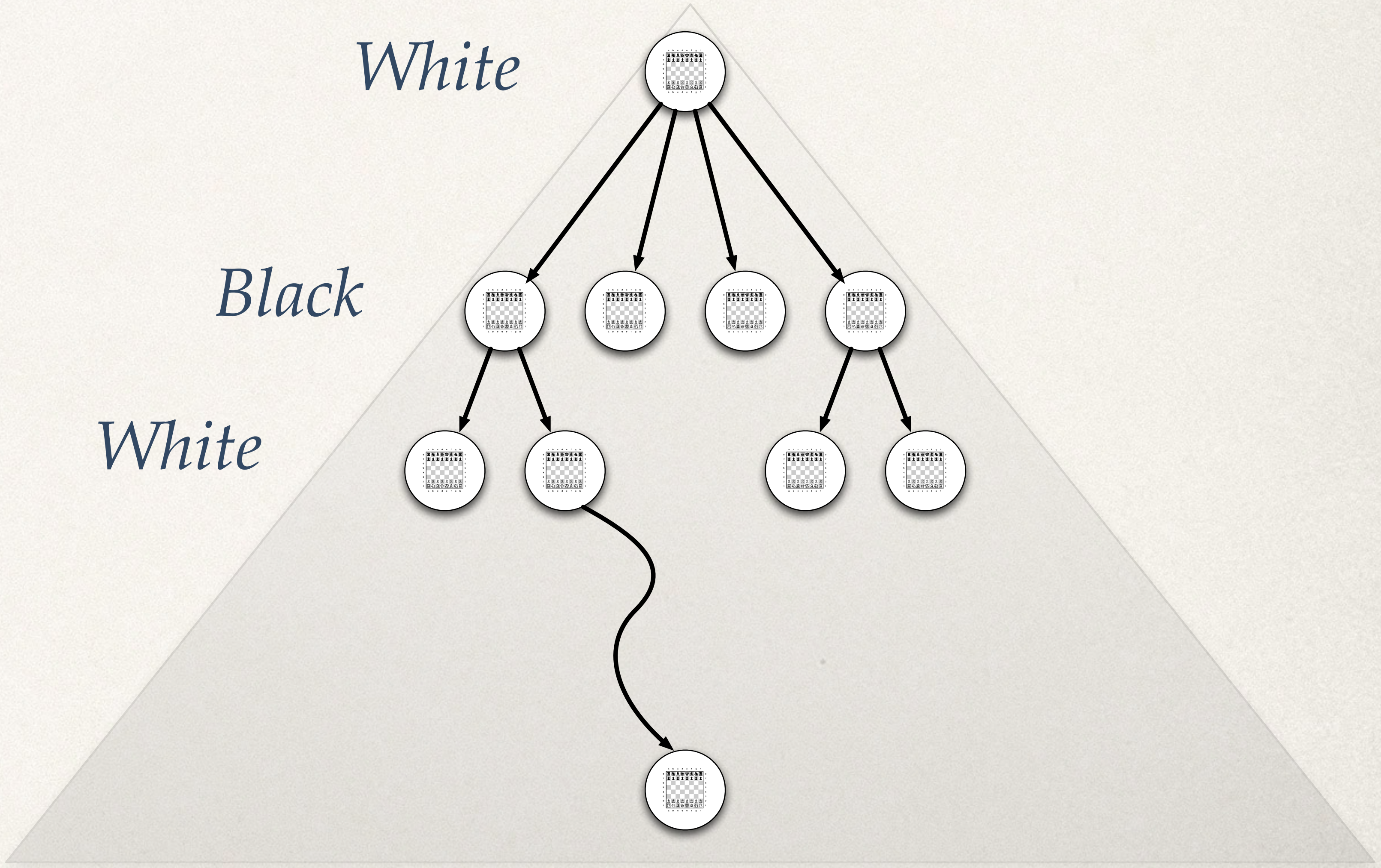
Chess:


Initial position
Possible positions after 1 play
Possible positions after 2 plays

Apologies: Trees are always upside down in computer science!

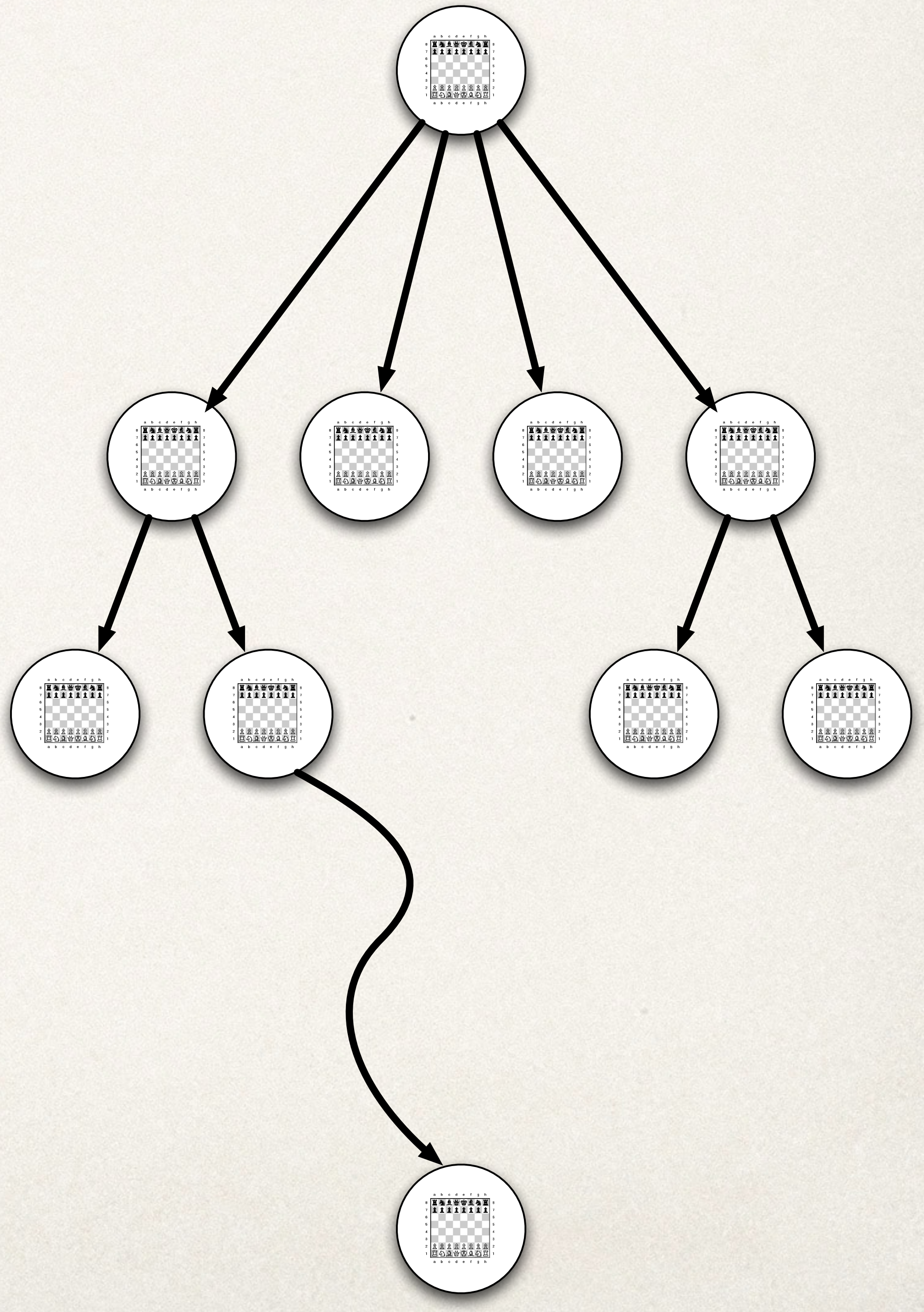
# The shape of the Game Tree

- Players alternate: Each level corresponds to a white or black move.
- Each node has children corresponding to possible next moves.
- At end, no possible moves.



# Tree terminology

- Composed of *nodes* (circles) and *edges* (lines).
- The *root* is the top node.
- Nodes directly beneath a given node are its *children*.
- A node with no children is a *leaf*.

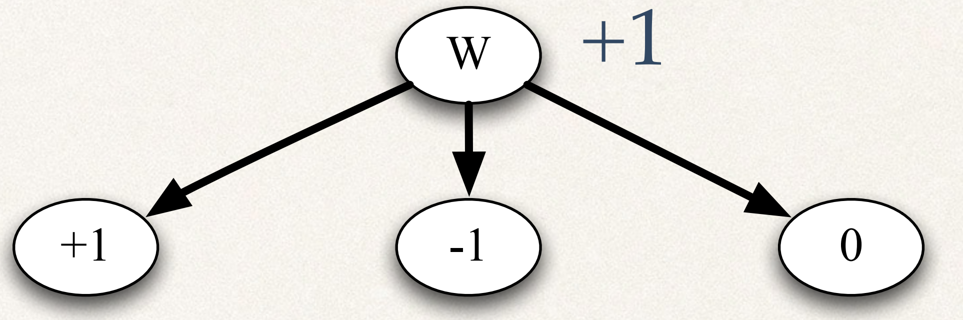


## Game tree semantics

- The history of a particular game then corresponds to a *path* in this tree
  - starts at the root,
  - travels downwards one step (move) at a time, from a node to one of its children,
  - ends at a leaf, which determines who wins.

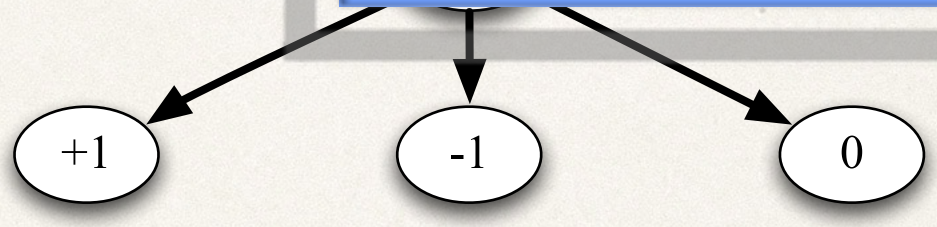
# Evaluation of Game Trees

- We begin by labeling leaves:
  - +1 if win for White;
  - -1 if win for Black;
  - 0 if tie.
- We would like to evaluate interior nodes of the tree.
- Note: Chess tree is finite.



White labels his nodes with the MA

By these rules, we may recursively evaluate every node of the tree.

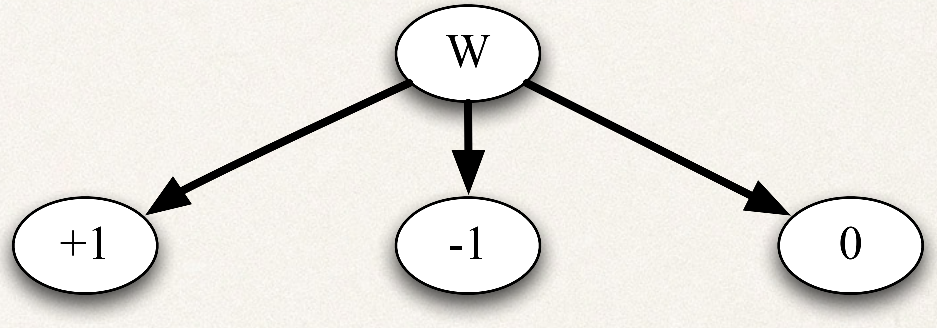


Black labels his nodes with the MIN child value.

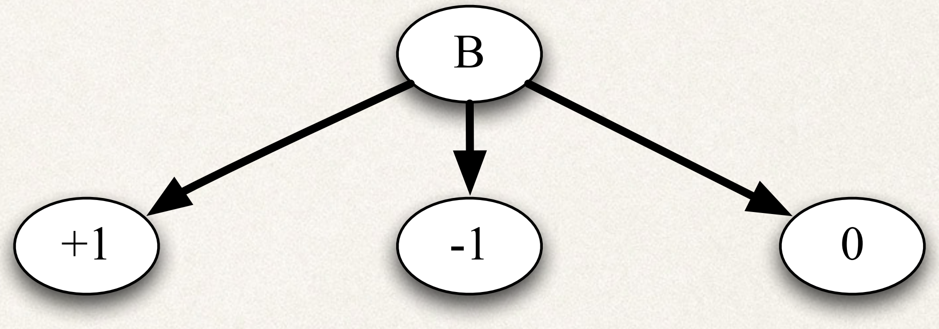
# When gods play chess: Game trees determine *perfect play*

- These evaluations determine *perfect play*: Given a position in which White must play, he simply selects the child with the largest value. If a node is labelled +1, White can force a win *regardless of how Black plays*.

- Similarly, for Black, who chooses the minimum.



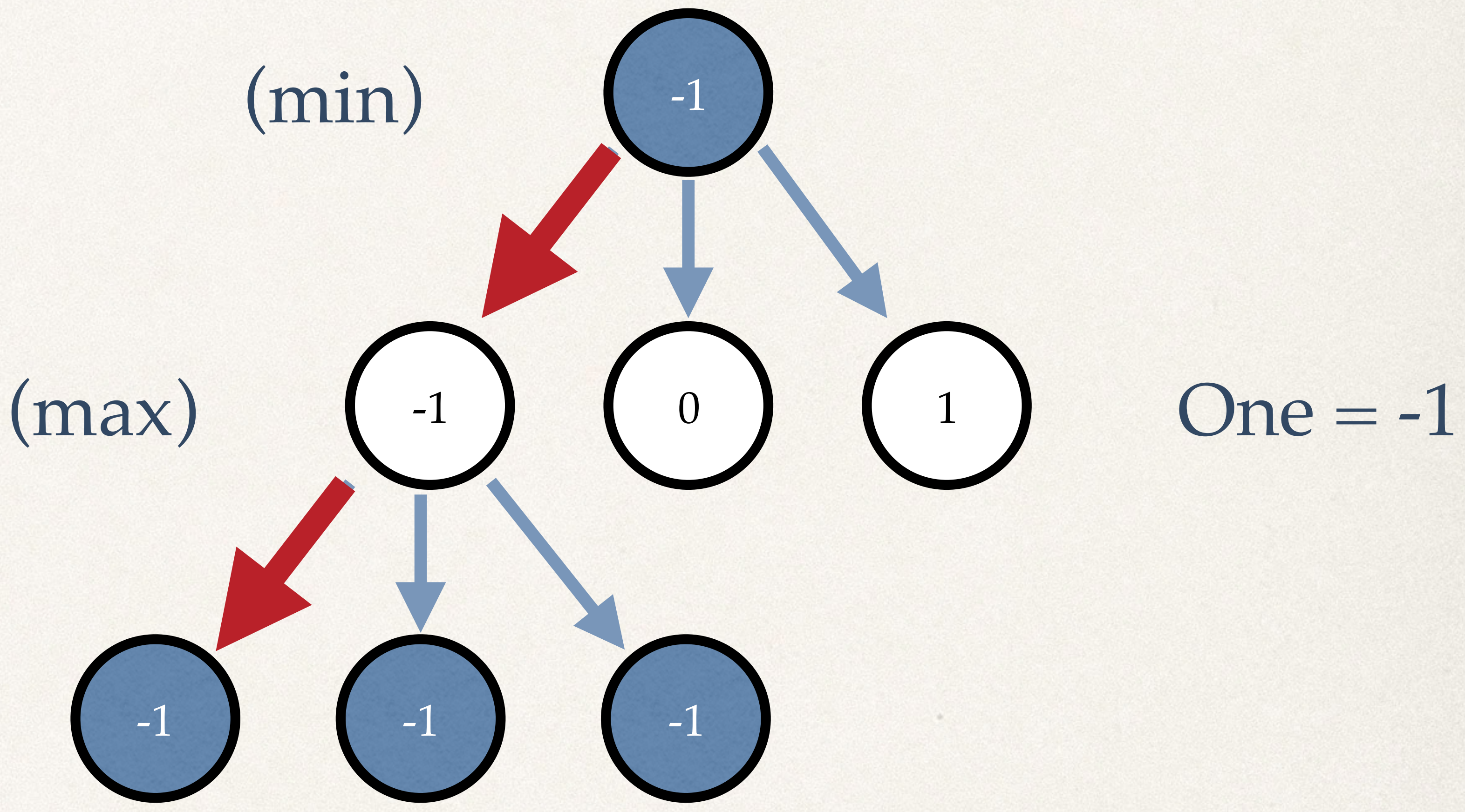
White labels his nodes with the MAX child value.



Black labels his nodes with the MIN child value.

# Using a labeled game tree for perfect play

- Suppose the root is labeled -1, a win for black.
- How, exactly, does black play to win?

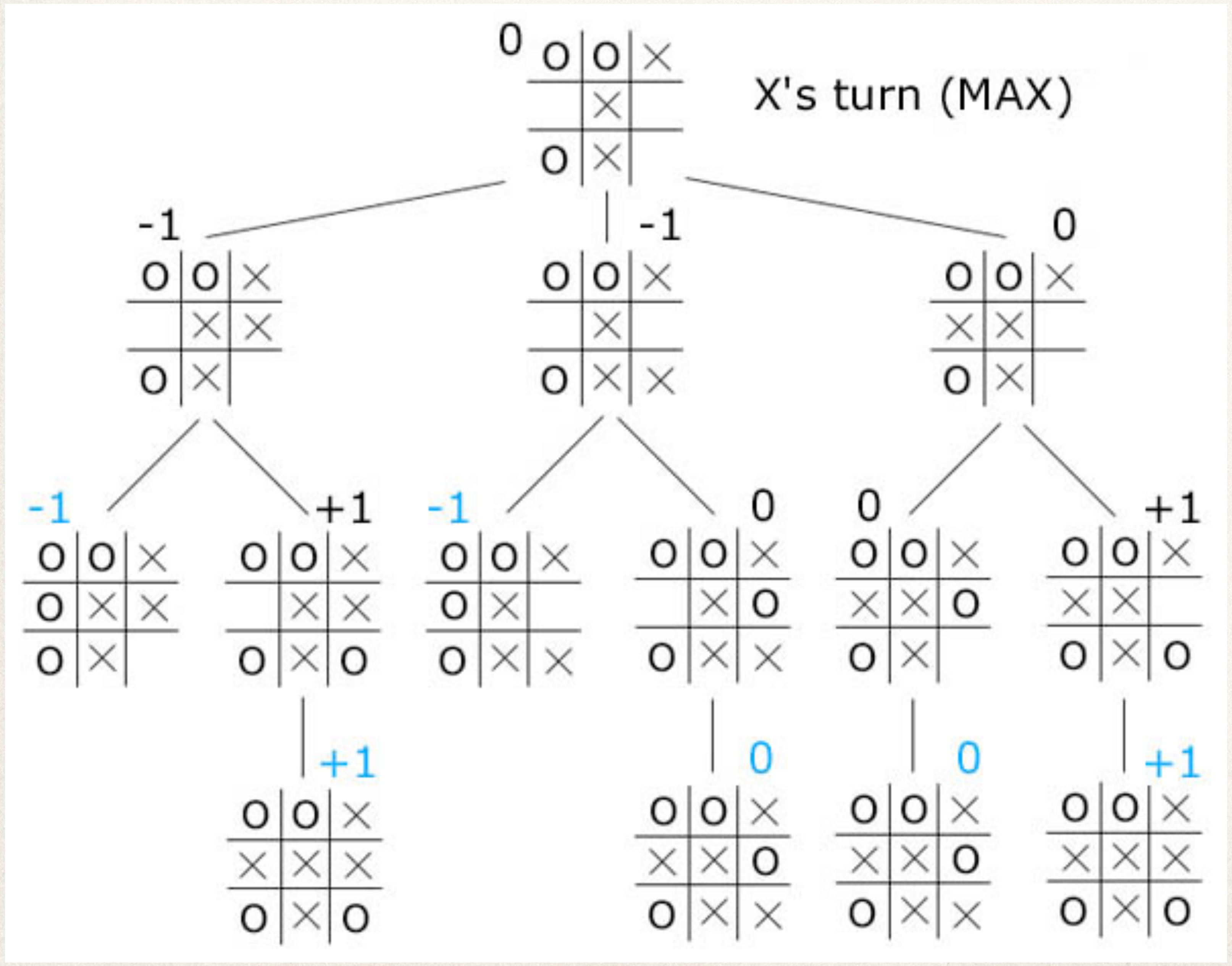


Great! Back to a black -1 node.

Eventually this leads to a -1 child: a win for black.

# An example: Tic-tac-toe

- Consider evaluation of a portion of the Tic-tac-toe game tree:



0 plays (min)

X plays (max)

0 plays (min)

# A natural question...

- Who wins, under perfect play, in Chess? No one knows! (In fact, it could be a win for Black.)

- Reversi? No one knows.

- Checkers? Major result in 2007: Checkers is a draw under perfect play. The Chinook team announced this result after ~$10^{14}$ calculations. It took about 200 desktop computers a year of computation.

- Tic-tac-toe? A draw under perfect play.

# Why can't we figure out who wins Chess under perfect play?

- What's the p
  - We have a fi space to rch, an
  - A clear evalu

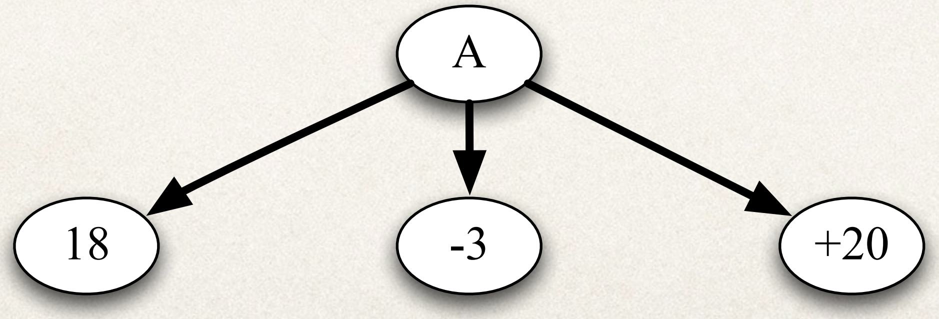
The trees are BIG

# Effective branching factor

- How large are these trees at depth d? Depends on *average branching factor*. (Average number of children beneath each node.)
    - Checkers: ~8
    - Reversi: ~10
    - Chess: ~35
    - Go: ~200
- If trees have branching factor **b**, there are $\mathbf{b^d}$ nodes at depth **d**...grows exponentially.
    - There are about $10^{80}$ particles in the observable universe. The chess game tree has about $10^{123}$ leaves!

# So that's nice, but...

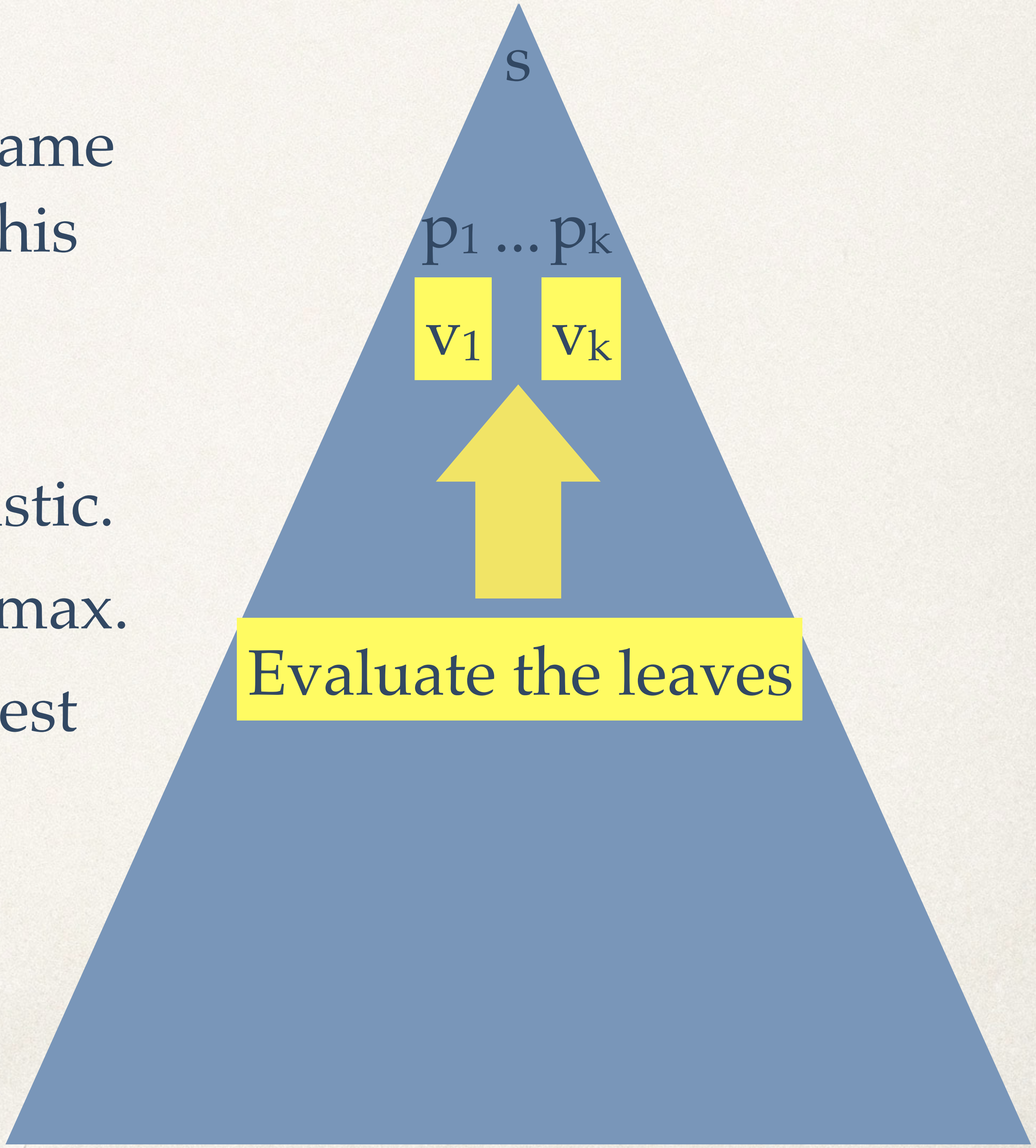
- How is this any good if you don't evaluate the tree all the way to the leaves?
- Idea: **Evaluation function**. A function f(*BOARD*) that heuristically captures the value of a board from, say, White's perspective.
- Then, evaluate to a particular depth, use f at the leaves, and percolate the values up using MIN/MAX.



# In a picture

It's White's turn: game position s

- White would like to know the game tree values of the children; but this takes too long. Instead...
- Cut off the game tree.
- Evaluate the leaves with a heuristic.
- Percolate the values up by min-max.
- Use these values to decide the best move



# Evaluation functions..

- For Chess: perhaps
  - +10 for a white queen
    -10 for a black queen,
    +5 for a white rook,
    -5 for a black rook, ...

  what about positional information?

- For Reversi: perhaps
  - +10 for a white corner piece,
    -10 for a black corner piece,
    +5 for a white edge piece,
    -5 for a black edge piece, ...

Learning good evaluations functions is a well-studied problem.

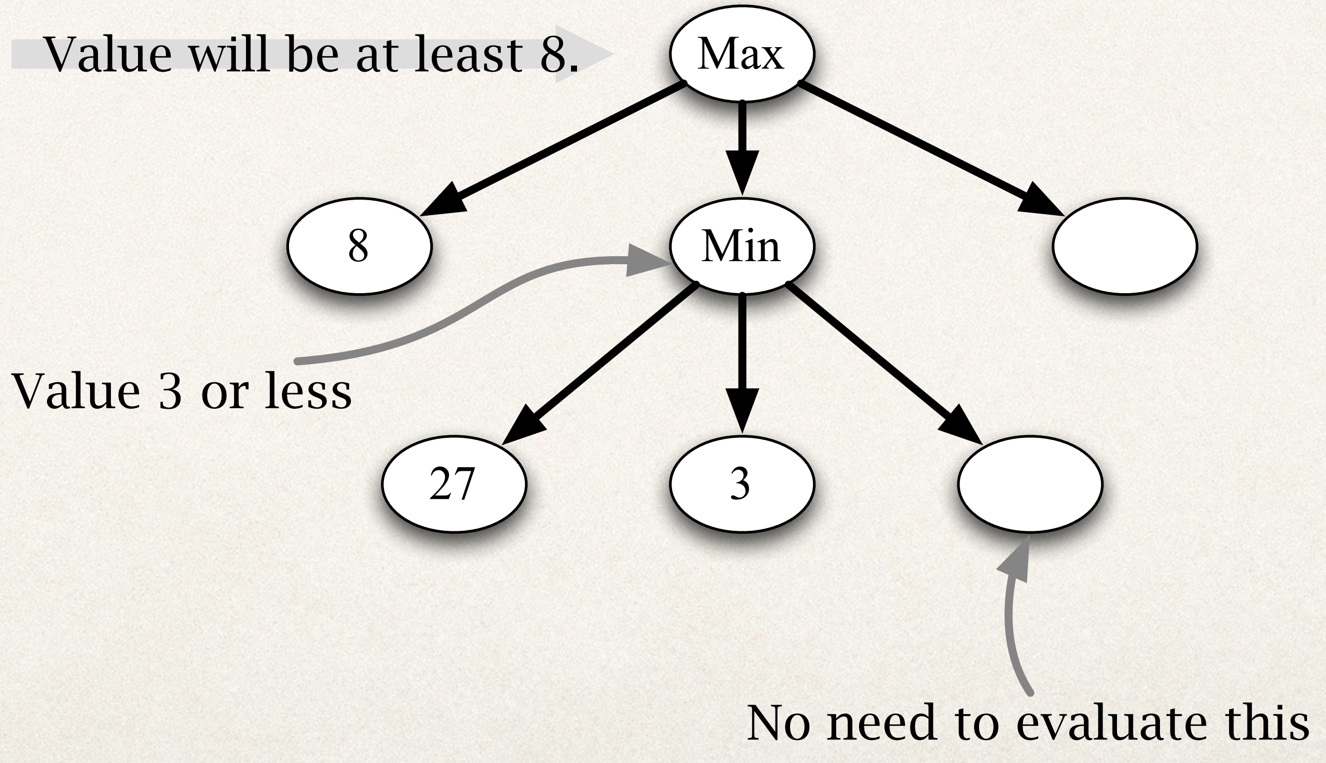
## In practice…

- Min-max evaluation of a tree of depth-d improves an evaluation function. (If you carry it to the logical extreme of arbitrary depth, it turns any reasonable evaluation function into a perfect one!)
- Logistello expands to depth ~17 in reversi.
- Deep blue expands to depth 5 or 6, but adaptively expands to greater depth (~20).

# Speeding things up...

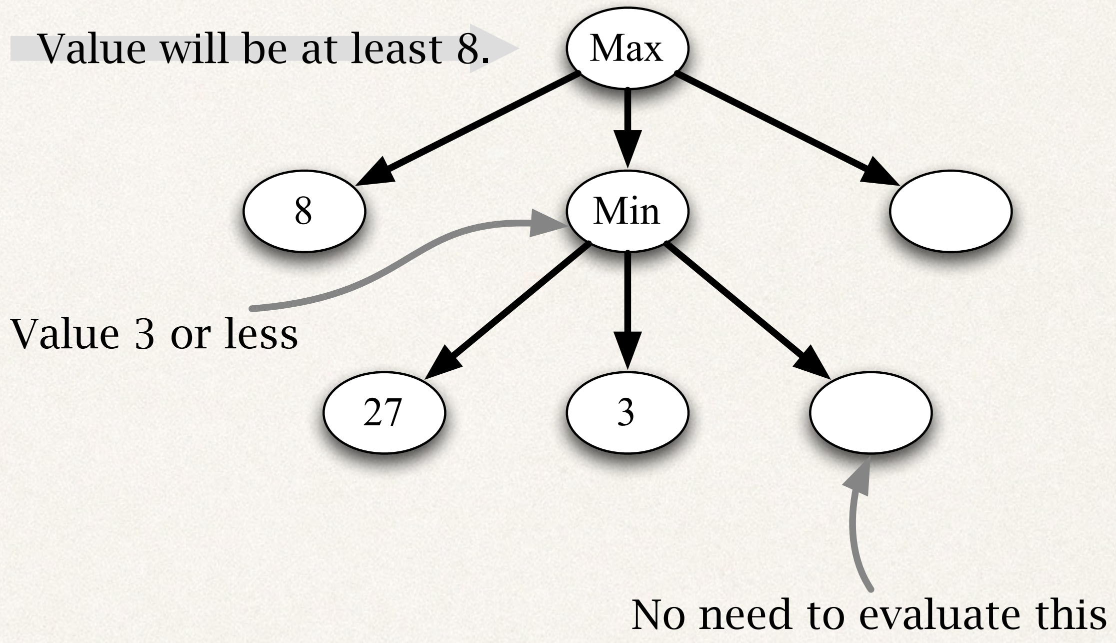
- Turns out there are many nice situations where you can prune the tree, *without affecting the result*. Often called α-β-pruning.

- Here’s the idea. White is exploring a node, say in “depth-first” fashion:



# Interesting side effect: order matters!

- Re-examine our example:



- If the 3 node and the 27 node had been exchanged, we could have expanded even less.

- How to order? One proposal: Based on evaluation function! (After all, it is an approximation to the real value.)

# Heuristic effect of pruning…

- …is remarkable!
- Under some reasonable assumptions, it can reduce the branching factor b to sqrt(b). This can *double* the depth to which you can search in the same amount of time. Why?
- Naive search: b * b * b * b … to depth d; $b^d$.
- Pruning search: b * 1 * b * 1 * … to depth d; $b^{d/2}$. For argument, imagine only using best moves for refutations.

# Iterative deepening and node ordering

- Often, unclear how deep you can search in given time limit.
- Solution: search to depth 2, then 3, then 4, ...
- Wasteful? Not much: each search takes roughly sqrt(b) more time. Most time spent in last search.
- Nice side effect: remembering previous search values for nodes near top of tree can provide *excellent ordering information*.

# Parallel game tree search

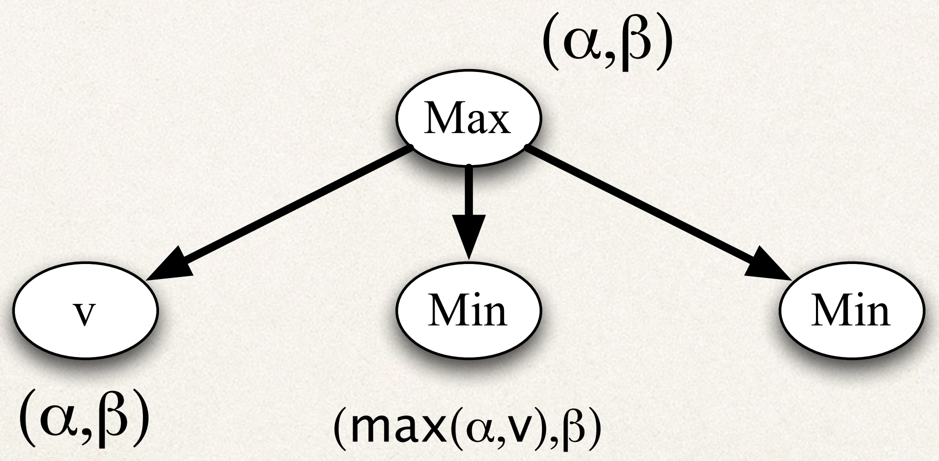
- Different subtrees can obviously be evaluated in parallel...BUT...
- $\alpha$-β-pruning seems to require a kind of depth-first evaluation.
- One proposal: **aspiration** search:
    - Break up value space into a family of windows of possible values:

      [*,-20] [-20,-10] [-10,0] [0,10] [10,20] [20,*]

    - One pruned search per window that may assume that all output values are in this window.

# Aspiration search, refines pruning

- A refinement of pruning: every tree is called with a *promise*. The promise $(\alpha,\beta)$ means: the output of this tree is only relevant to the overall search if its value is at least $\alpha$ and no more than $\beta$.
- These promises can be propagated through the search:



If $v > \beta$ then quit; this tree is no use!

Otherwise, call next tree with $(\max(\alpha,v),\beta)$

## Want to practice?

- You have all the tools to write a chess player in Scheme. Build it up in blocks: You'll need code for:
  - Maintaining a board (try a vector of vectors—you need to maintain a "two-dimensional" 8x8 board).
  - Generating all possible next moves.
  - Recursively exploring the game tree.
  - Evaluating a simple evaluation function.
  - Interacting with the user.

# My Recommendations for Your Future: This Summer

- **Write some code**. Either work in Scheme or dive in to Python (which will be used in the next course, CSE2050).

- If you want to prepare for CSE2050, look over the online book

  "Solving problems with data structures and algorithms in Python"

- If you want to prepare for CSE2500, my recommendation would be to look over *The Book of Proof*:

  http://www.people.vcu.edu/~rhammack/BookOfProof/

# My Recommendations for Your Future: At UConn

- Master one of the **big industrial-strength languages**: C++, Swift, or Java. Expect this to take a few years of low-level effort. I recommend C++, but there are benefits to the others as well. There is a new C++ course in the department (3150); take it when you take 3100.
- Take CSE3100—learn how to write and debug **multi-threaded code**. (If you are on the new plan of study, it's required.)
- **Algorithms** (CSE3500): Take the class and get serious about further study: these are the real secrets of computer science.
- **Linear Algebra**: You need it for sophisticated modeling of real-world phenomena—in particular, you need it to spin up on **machine learning**. (Of course, it is also critical for graphics.)

## Final exam

- Basic recursive programming dynamics. Scheme environments.
- List processing: sorting; maintaining a set in a list. Other ADTs (stacks/queues).
- (Functional) tree processing: bst and heap conventions. Time complexity of basic operations. Relationship to sorting.
- Mutable data and objects. Basic object design and semantics. Mutable trees. Vectors; maintaining sets (other ADTs) in vectors.