Game Playing

Chapter 5.1 – 5.3

Game Playing and AI

- Game playing was thought to be a good problem for AI research:
  - game playing is non-trivial
    - players need "human-like" intelligence
    - games can be very complex (e.g., Chess, Go)
    - requires decision making within limited time
  - games usually are:
    - well-defined and repeatable
    - fully observable and limited environments
  - can directly compare humans and computers

Types of Games

Definitions:

- Zero-sum: one player's gain is the other player's loss. Does not mean *fair*.
- Discrete: states and decisions have discrete values
- Finite: finite number of states and decisions
- Deterministic: no coin flips, die rolls – no chance
- Perfect information: each player can see the complete game state. No simultaneous decisions.

Game Playing and AI

	Deterministic	**Stochastic (chance)**
Fully Observable (perfect info)	Checkers, Chess, Go, Othello	Backgammon, Monopoly
Partially Observable (imperfect info)	?	Bridge, Poker, Scrabble

All are also multi-agent, adversarial, static tasks

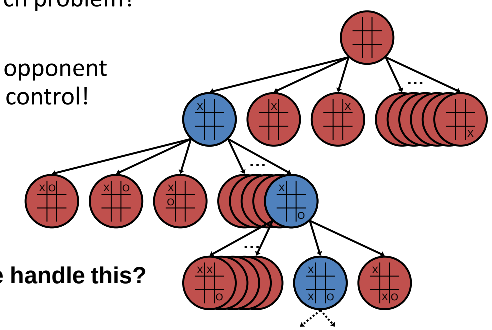
Game Playing as Search

- Consider two-player, perfect information, 0-sum board games:
  - e.g., chess, checkers, tic-tac-toe
  - board configuration: a unique arrangement of "pieces"
- Representing board games as search problem:
  - **states:** board configurations
  - **actions:** legal moves
  - **initial state:** current board configuration
  - **goal state:** game over/terminal board configuration

Game Tree Representation

What's the new aspect to the search problem?

There's an opponent we cannot control!



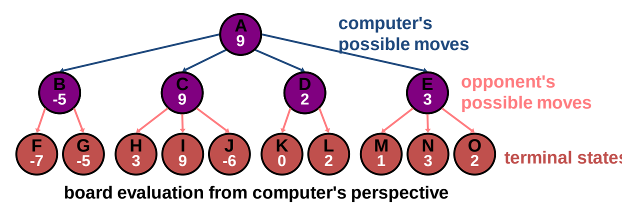
How can we handle this?

Greedy Search using an Evaluation Function

- A ***Utility function*** is used to map each **terminal state** of the board (i.e., states where game is over) to a score indicating the value of that outcome to the computer
- We'll use:
  - positive for winning; large + means better for computer
  - negative for losing; large - means better for opponent
  - 0 for a draw
  - typical values (loss to win):
    - -∞ to +∞
    - -1.0 to +1.0

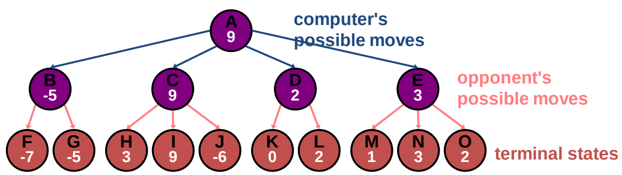
Greedy Search using an Evaluation Function

- Expand the search tree to the terminal states on each branch
- Evaluate utility of each terminal board configuration
- Make the initial move that results in the board configuration with the maximum value



Greedy Search using an Evaluation Function

- Assuming a reasonable search space, what's the problem?
  This ignores what the opponent might do!
  Computer chooses C
  Opponent chooses J and defeats computer



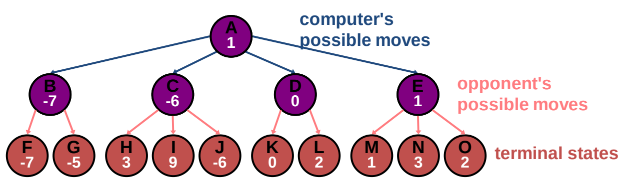
board evaluation from computer's perspective

Minimax Principle

- **Assume *both* players play optimally**
  - given there are two moves until the terminal states
  - high utility numbers favor the computer
    - computer should choose maximizing moves
  - low utility numbers favor the opponent
    - smart opponent chooses minimizing moves

Minimax Principle

- The computer assumes after it moves the opponent *will* choose the minimizing move
- The computer chooses the best move considering both its move ***and*** the opponent's optimal move



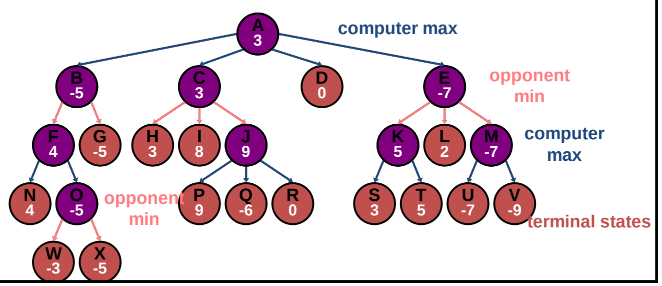
board evaluation from computer's perspective

Propagating Minimax Values up the Game Tree

- Explore the tree to the terminal states
- Evaluate utility of the resulting board configurations
- The computer makes a move to put the board in the best configuration for it assuming the opponent makes her best moves on her turn:
  - start at the leaves
  - assign value to the parent node as follows
    - use minimum when children are opponent's moves
    - use maximum when children are computer's moves

Deeper Game Trees

- Minimax can be generalized to more than 2 moves
- **Propagate** values up through the tree



General Minimax Algorithm

```
For each move by the computer:
1. Perform depth-first search to a
   terminal state
2. Evaluate each terminal state
3. Propagate upwards the minimax values
   if opponent's move, propagate up
     minimum value of children
   if computer's move, propagate up
     maximum value of children
4. choose move at root with the maximum of
   minimax values of children
```

Complexity of Minimax Algorithm

Assume all terminal states are at depth d

- Space complexity
  - Depth-first search, so $O(bd)$
- Time complexity
  - Branching factor b, so $O(b^d)$

- Time complexity is a major problem since computer typically only has a finite amount of time to make a move

Complexity of Game Playing

- Assume the opponent's moves can be predicted given the computer's moves
- How complex would search be in this case?
  - worst case: $O(b^d)$ ***b***ranching factor, ***d***epth
  - **Tic-Tac-Toe:** ~5 legal moves, 9 moves max game
    - 5^9 = 1,953,125 states
  - **Chess:** ~35 legal moves, ~100 moves per game
    - $b^d \sim 35^{100} \sim 10^{154}$ states, only $\sim 10^{40}$ legal states

- Common games produce ***enormous*** search trees

Complexity of Minimax Algorithm

- Minimax algorithm applied to complete game trees is impractical in practice
  - instead do depth-limited search to **ply** (depth) ***m***, i.e., **local search**
  - but Utility function defined only for terminal states
  - we need to know a value for non-terminal states
- **Static Evaluation functions** use heuristics to estimate the value of non-terminal states

Static Board Evaluator (SBE)

- A **Static Board Evaluation function** is used to estimate how good the current board configuration is for the computer
  - it reflects the computer's chances of winning from that node
  - it must be easy to calculate from board configuration
- For example, for Chess:

  SBE = α * materialBalance + β * centerControl + γ * ...

  where **material balance = Value of white pieces - Value of black pieces, pawn = 1, rook = 5, queen = 9, etc.**

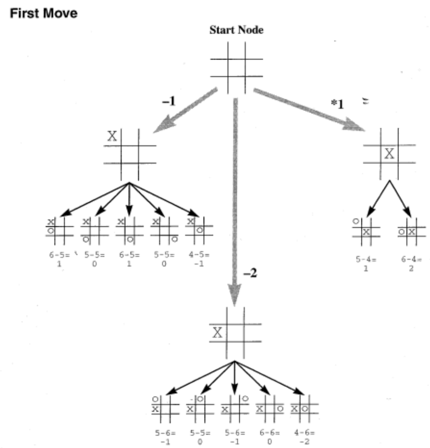
Static Board Evaluator (SBE)

- Typically, one subtracts how good it is for the opponent from how good it is for the computer
- If the SBE gives X for a player, then it gives -X for the opponent
- SBE should agree with the Utility function when calculated at terminal nodes

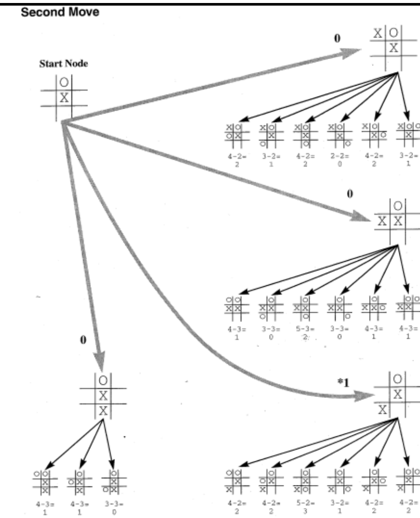
Minimax with Evaluation Functions

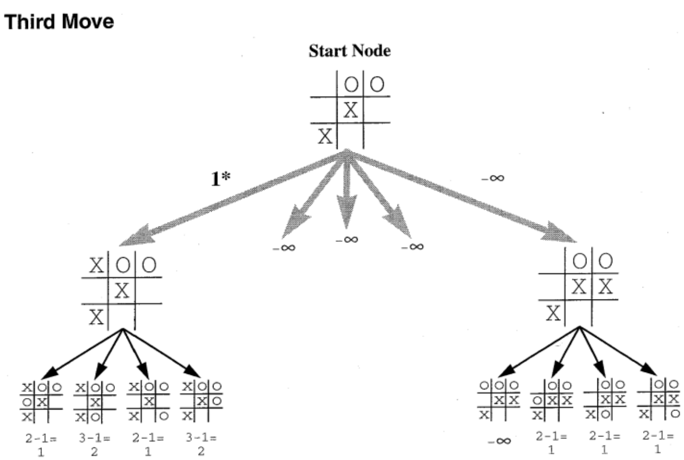
- The same as general Minimax, except
  - only goes to depth ***m***
  - estimates value using SBE function
- How would this algorithm perform at Chess?
  - if could look ahead ~4 pairs of moves (i.e., 8 ply), would be consistently beaten by average players
  - if could look ahead ~8 pairs, is as good as human master

Tic-Tac-Toe Example



Evaluation function = (# 3-lengths open for me) – (# 3-lengths open for opponent)





Minimax Algorithm

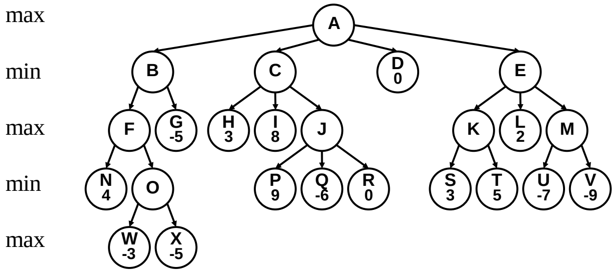
function Max-Value(s)
inputs:
    s: current state in game, Max about to play
output: *best-score* (*for Max*) *available from s*

if (s is a terminal state or at depth limit)
then return (SBE value of s)
else
    α := – ∞
    foreach s′ in Successors(s)
        α := max(α , Min-Value(s′))
return α

function Min-Value(s)
output: *best-score (for Min) available from s*

if (s is a terminal state or at depth limit)
then return (SBE value of s)
else
    β := ∞
    foreach s′ in Successors(s)
        β := min(β , Max-Value(s′))
return β

Minimax Example



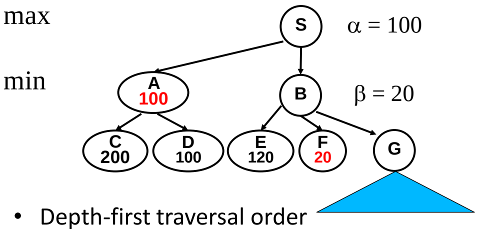
Summary So Far

- Can't use Minimax search to end of the game
  – if we could, then choosing move is easy
- SBE isn't perfect at estimating/scoring
  – if it was, just choose best move without searching
- Since neither is feasible for interesting games, combine Minimax and SBE concepts:
  – Minimax to depth ***m***
  – use SBE to estimate/score board configuration

Alpha-Beta Idea

- Some of the branches of the game tree won't be taken if playing against an intelligent opponent
- “If you have an idea that is surely bad, don’t take the time to see how truly awful it is.”
  -- Pat Winston
- ***Pruning* can be used to ignore some branches**
- While doing DFS of game tree, keep track of:
  – **Alpha (α)** at maximizing levels:
    - highest SBE value seen so far in subtree below node
    - **lower bound** on node's final minimax value
  – **Beta (β)** at minimizing levels:
    - lowest SBE value seen so far in subtree below node
    - **upper bound** on node's final minimax value

Alpha-Beta Idea: Alpha Cutoff



- Depth-first traversal order
- After returning from A, can get ***at least 100 at S***
- After returning from F, can get ***at most 20 at B***
- At this point no matter what minimax value is computed at G, S will prefer A over B. So, S loses interest in B
- There is no need to visit G. The subtree at G is pruned. Saves time. Called “***Alpha cutoff***” (at MIN node B)

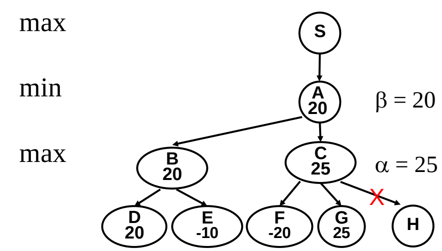
Alpha Cutoff

- At each MIN node, keep track of the minimum value returned so far from its visited children
- Store this value as β
- Anytime β is updated (at a MIN node), check its value against the α value of (all) its MAX node ancestor(s)
- If $\alpha \geq \beta$ for some MAX node ancestor, don't visit any more of the current MIN node's children

Beta Cutoff

- At each MAX node, keep track of the maximum value returned so far from its visited children
- Store this value as α
- Anytime α is updated (at a MAX node), check its value against the β value of (all) its MIN node ancestor(s)
- If $\alpha \geq \beta$ for some MIN node ancestor, don't visit any more of the current MAX node's children

Beta Cutoff Example



- After returning from B, can get at most 20 at MIN node A
- After returning from G, can get at least 25 at MAX node C
- No matter what minimax value is found at H, A will NEVER choose C over B, so don't visit node H
- Called "Beta Cutoff" (at MAX node C)

Alpha-Beta Idea

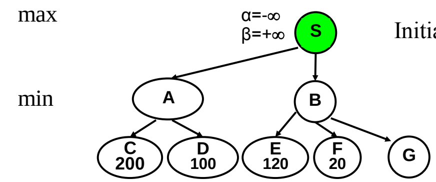
- Store α value at MAX nodes and β value at MIN nodes
- Cutoff/pruning occurs
  - At MAX node (when **maximizing**)
    if $\alpha \geq \beta$ for some MIN ancestor, stop expanding
  - Don't visit more children of MAX node
  - Opponent won't allow computer to make this move

  - At MIN node (when **minimizing**)
    if, for some MAX node ancestor, $\alpha \geq \beta$, stop expanding
  - Don't visit more children of MIN node
  - Computer won't want to take this move

Implementation of Cutoffs

- At each node, keep ***both*** α and β values
  - At MAX node, α = largest value from its children visited so far, and β = smallest value from its MIN node ancestors in search tree
    - **α value at MAX comes from *descendants***
    - **β value at MAX comes from MIN node *ancestors***
  - At MIN node, β = smallest value from its children visited so far, and α = largest value from its MAX node ancestors in search tree
    - **α value at MIN comes from MAX node ancestors**
    - **β value at MIN comes from descendants**

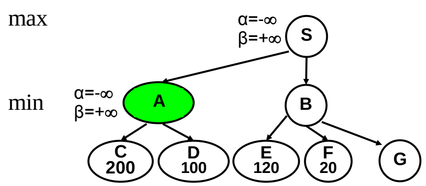
Implementation of Alpha Cutoff



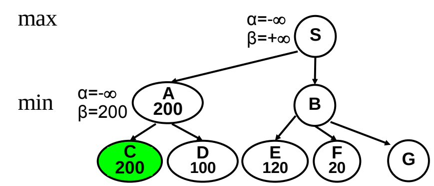
Initialize root's values

- At each node, keep two bounds (based on all ancestors and descendants visited so far):
  - α: the best (largest) MAX can do
  - β: the best (smallest) MIN can do
- If at anytime $\alpha \geq \beta$ at a node, the remaining children are pruned

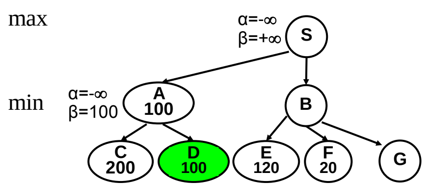
Alpha Cutoff Example



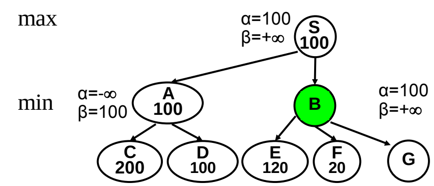
Alpha Cutoff Example



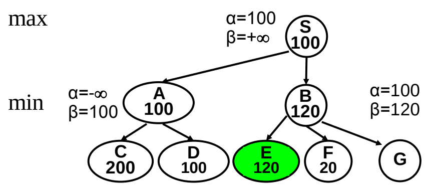
Alpha Cutoff Example



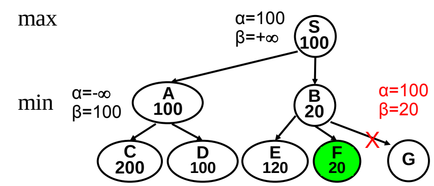
Alpha Cutoff Example



Alpha Cutoff Example



Alpha Cutoff Example



Notes:
- Alpha cutoff means not visiting some of a MIN node's children
- **Beta values at MIN come from *descendants***
- **Alpha value at MIN come from MAX node *ancestors***

Alpha-Beta Algorithm

function Max-Value (s, α, β)
inputs:
  s: current state in game, Max about to play
  α: best score (highest) for Max along path to s
  β: best score (lowest) for Min along path to s

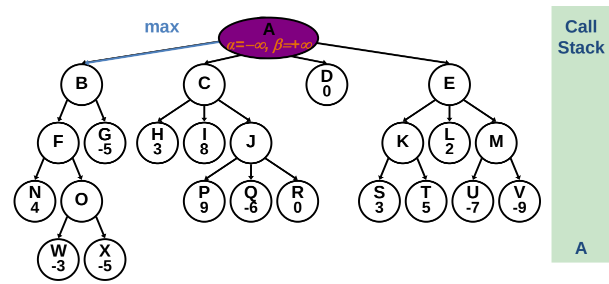
  if (s is a terminal state)
  then return (SBE value of s)
  else for each s' in Successors(s)
    α := max(α, Min-Value(s', α, β))
    if (α ≥ β) **then return** α   /* prune remaining children of Max */
  return α

function Min-Value(s, α, β)

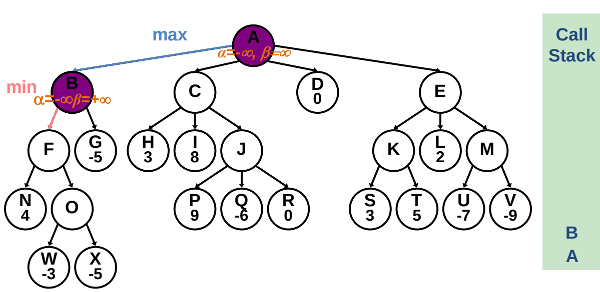
  if (s is a terminal state)
  then return (SBE value of s)
  else for each s' in Successors(s)
    β := min(β, Max-Value(s', α, β))
    if (α ≥ β) **then return** β   /* prune remaining children of Min */
  return β

Starting from the root:
Max-Value(root, -∞, +∞)

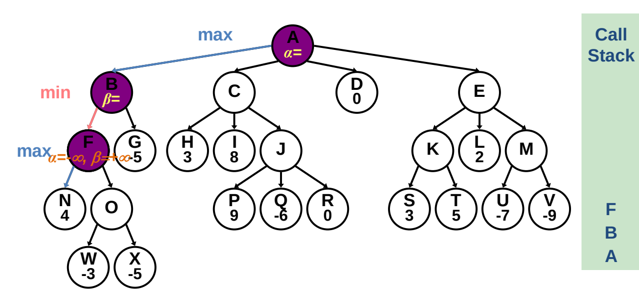
Alpha-Beta Example



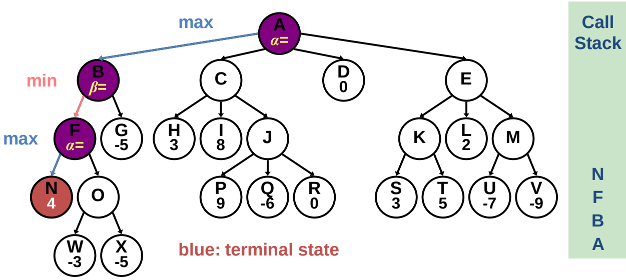
Alpha-Beta Example



Alpha-Beta Example

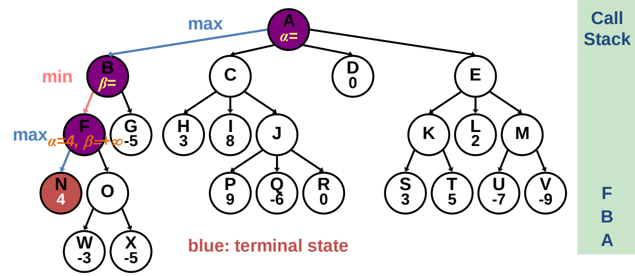


Alpha-Beta Example

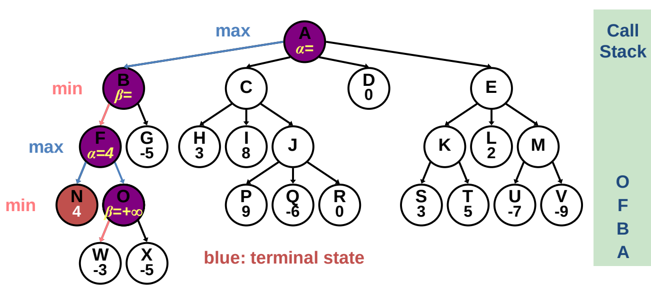


Alpha-Beta Example

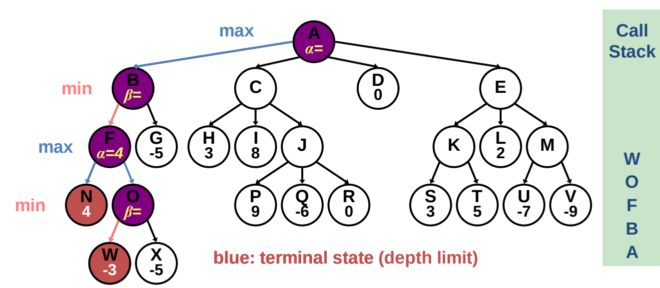
`alpha(F) = 4`, maximum seen so far



Alpha-Beta Example

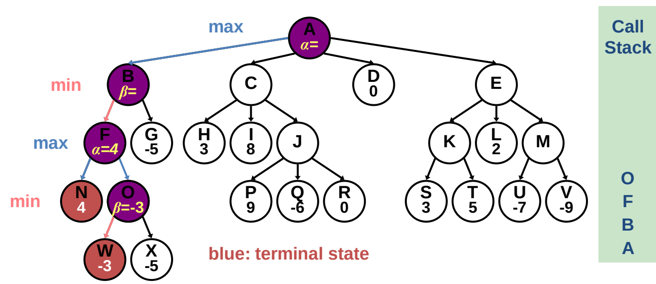


Alpha-Beta Example



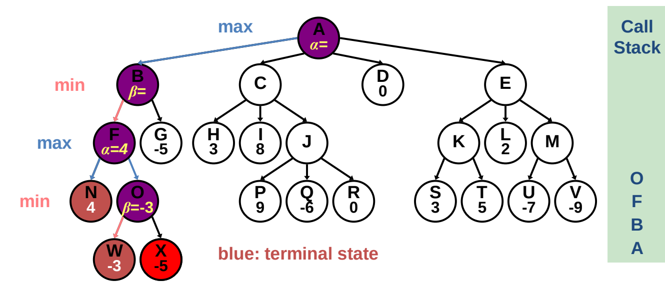
Alpha-Beta Example

`beta(O) = -3`, minimum seen so far



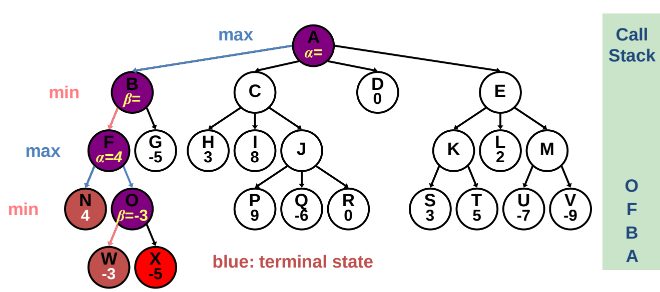
Alpha-Beta Example

O's beta ≤ F's alpha: stop expanding O (alpha cutoff)



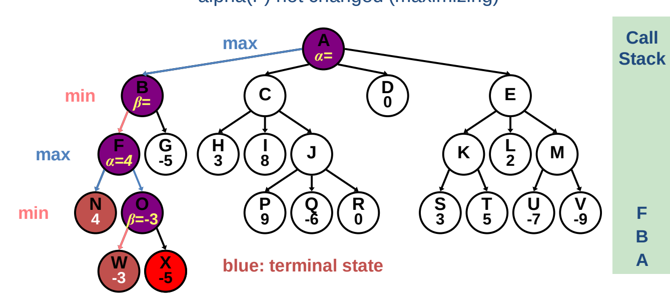
Alpha-Beta Example

Why? Smart opponent will choose W or worse, thus O's upper bound is –3. So computer shouldn't choose O:-3 since N:4 is better.



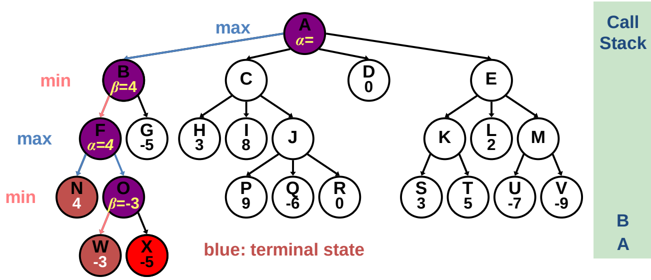
Alpha-Beta Example

alpha(F) not changed (maximizing)

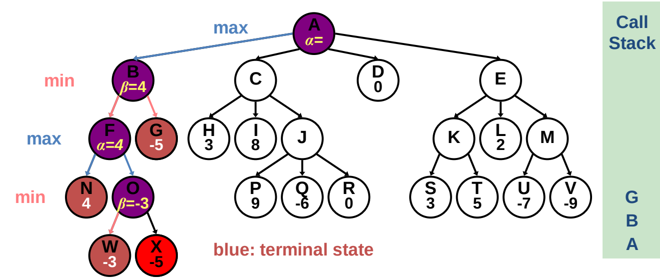


Alpha-Beta Example

`beta(B) = 4`, minimum seen so far

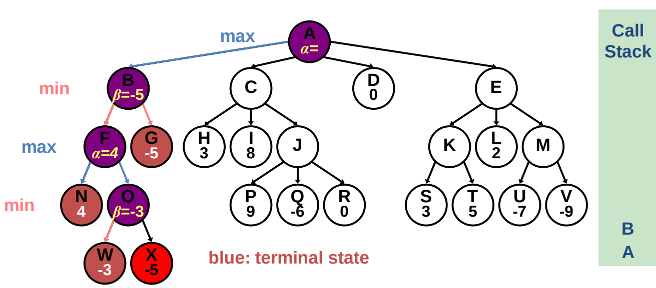


Alpha-Beta Example



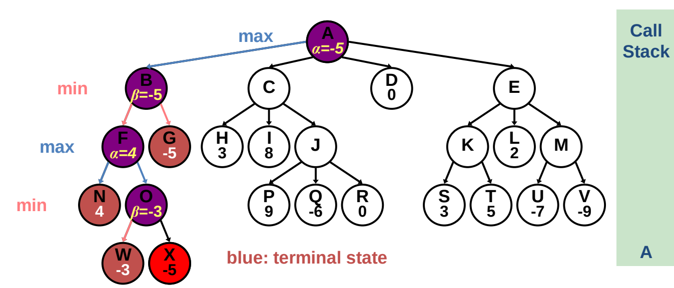
Alpha-Beta Example

`beta(B) = -5`, updated to minimum seen so far

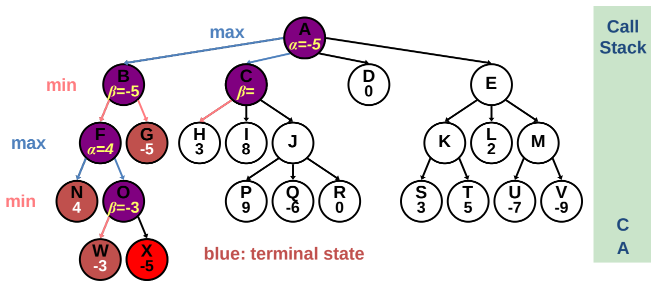


Alpha-Beta Example

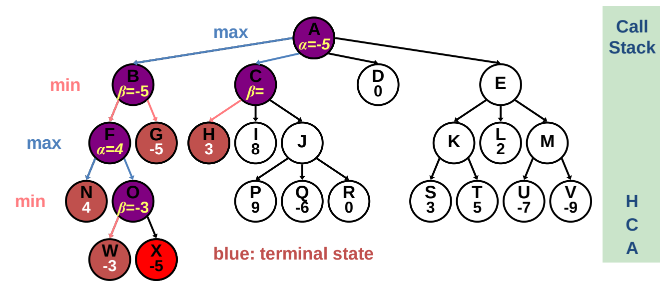
`alpha(A) = -5`, maximum seen so far



Alpha-Beta Example

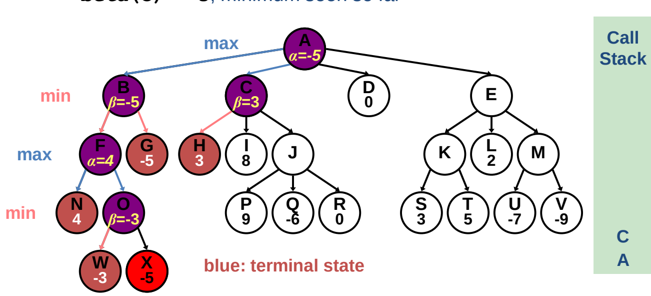


Alpha-Beta Example

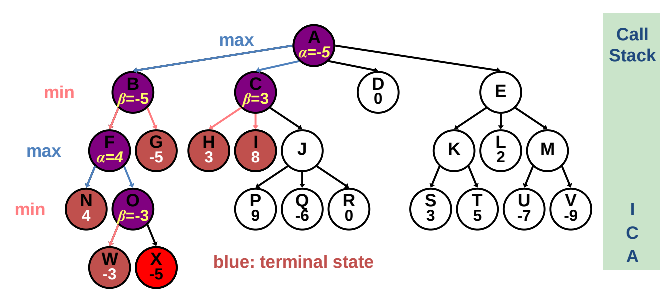


Alpha-Beta Example

`beta(C) = 3`, minimum seen so far

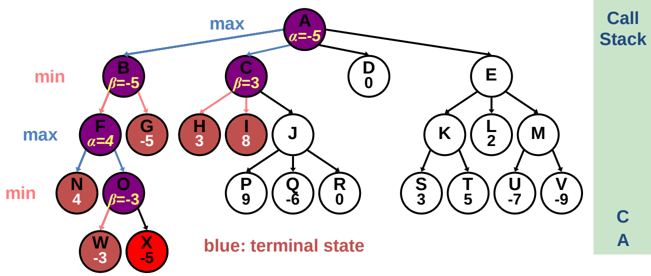


Alpha-Beta Example

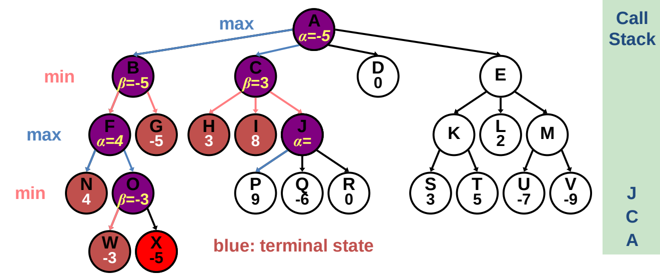


Alpha-Beta Example

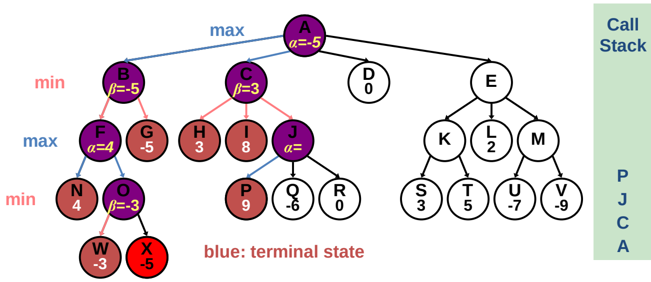
beta(C) not changed (minimizing)



Alpha-Beta Example

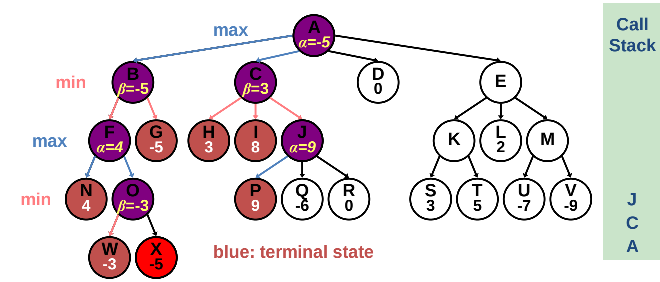


Alpha-Beta Example



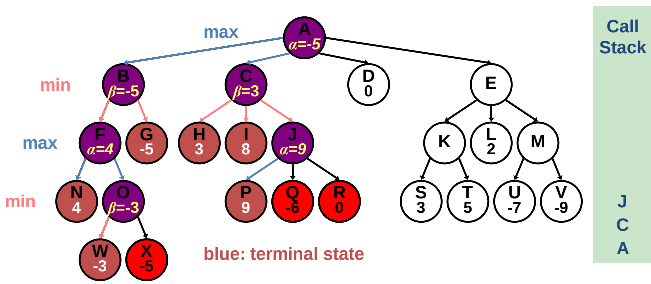
Alpha-Beta Example

`alpha(J) = 9`



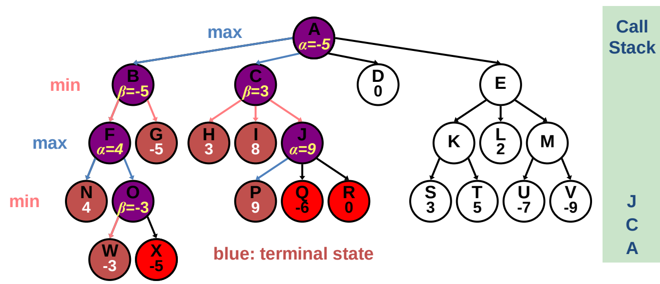
Alpha-Beta Example

J's alpha ≥ C's beta: stop expanding J (beta cutoff)



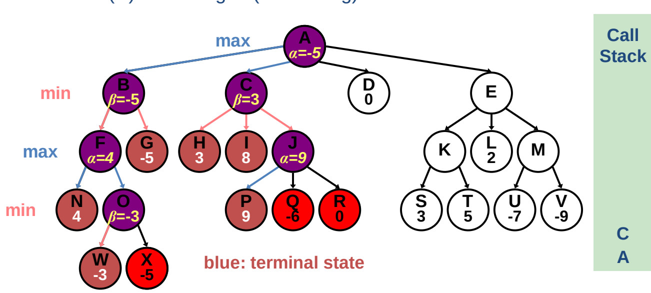
Alpha-Beta Example

Why? Computer should choose P or better, thus J's lower bound is 9. So smart opponent won't take J:9 since H:3 is worse.



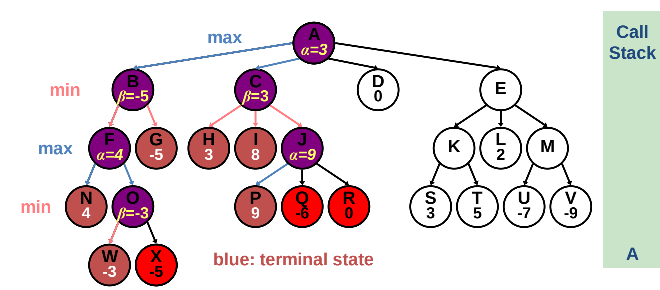
Alpha-Beta Example

beta(C) not changed (minimizing)

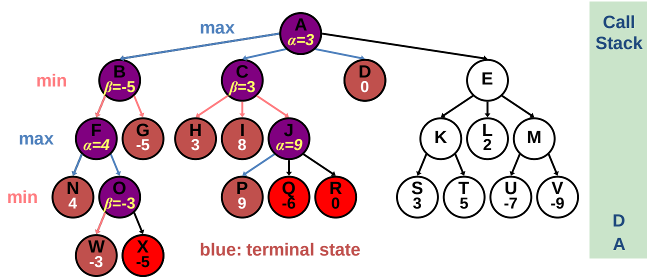


Alpha-Beta Example

`alpha(A) = 3`, updated to maximum seen so far

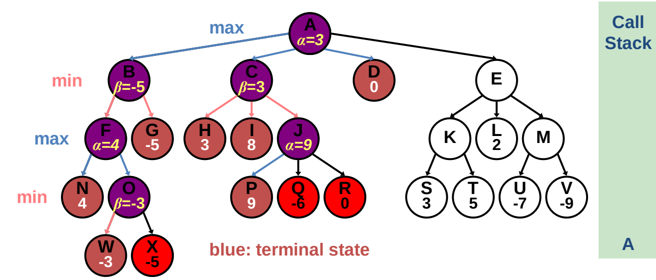


Alpha-Beta Example



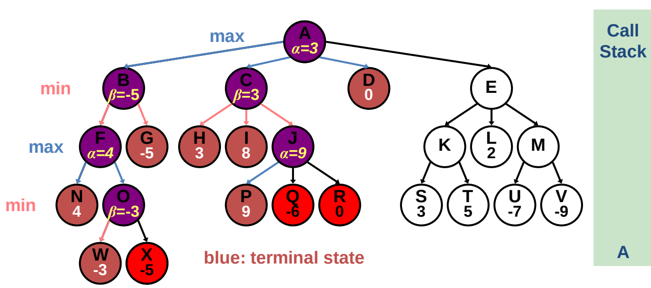
Alpha-Beta Example

alpha(A) not updated (maximizing)



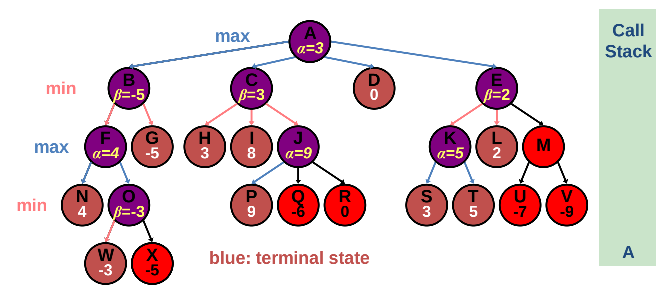
Alpha-Beta Example

How does the algorithm finish the search tree?



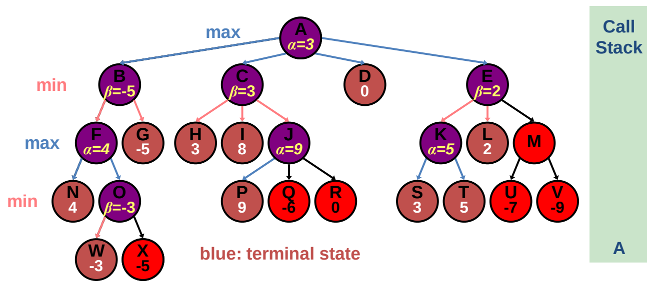
Alpha-Beta Example

E's beta ≤ A's alpha: stop expanding E (alpha cutoff)



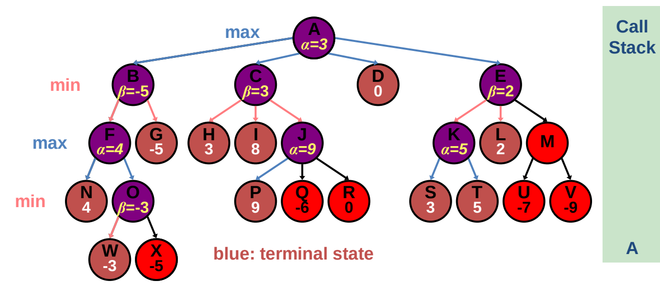
Alpha-Beta Example

Why? Smart opponent will choose L or worse, thus E's upper bound is 2. So computer shouldn't choose E:2 since C:3 is better path.



Alpha-Beta Example

Final result: Computer chooses move C



Effectiveness of Alpha-Beta Search

- Effectiveness depends on the *order* in which successors are examined; more effective if *best* successors are examined *first*

- Worst Case:
  - ordered so that ***no*** pruning takes place
  - no improvement over exhaustive search
- Best Case:
  - each player's *best* move is evaluated *first*
- In practice, performance is closer to best, rather than worst, case

Effectiveness of Alpha-Beta Search

- In practice often get $O(b^{(d/2)})$ rather than $O(b^d)$
  - same as having a branching factor of $\sqrt{b}$ since $(\sqrt{b})^d = b^{(d/2)}$
- Example: Chess
  - goes from $b \sim 35$ to $b \sim 6$
  - permits much deeper search for the same time
  - makes computer chess competitive with humans

Dealing with Limited Time

- In real games, there is usually a time limit *T* on making a move
- How do we take this into account?
  - cannot stop alpha-beta midway and expect to use results with any confidence
  - so, we could set a conservative depth-limit that guarantees we will find a move in time < *T*
  - but then, the search may finish early and the opportunity is wasted to do more search

Dealing with Limited Time

- In practice, **iterative deepening search** (IDS) is used
  - run alpha-beta search with an *increasing depth limit*
  - when the clock runs out, use the solution found for the last completed alpha-beta search (i.e., the deepest search that was completed)

The Horizon Effect

- Sometimes disaster lurks just beyond search depth
  - computer captures queen, but a few moves later the opponent checkmates (i.e., wins)
- The computer has a **limited horizon**, it cannot see that this significant event could happen
- How do you avoid catastrophic losses due to "short-sightedness"?
  - quiescence search
  - secondary search

The Horizon Effect

- **Quiescence Search**
  - when SBE value is frequently changing, look deeper than limit
  - look for point when game "quiets down"
  - E.g., always expand any forced sequences
- **Secondary Search**
  1. find best move looking to depth d
  2. look k steps beyond to verify that it still looks good
  3. if it doesn't, repeat step 2 for next best move

Book Moves

- Build a database of opening moves, end games, and studied configurations
- If the current state is in the database, use database:
  - to determine the next move
  - to evaluate the board
- Otherwise, do alpha-beta search

More on Evaluation Functions

- The board evaluation function estimates how good the current board configuration is for the computer
  - it is a heuristic function of the board's features
    - i.e., ***function***$(f_1, f_2, f_3, \ldots, f_n)$
  - the features are numeric characteristics
    - feature 1, f_1, is number of white pieces
    - feature 2, f_2, is number of black pieces
    - feature 3, f_3, is f_1/f_2
    - feature 4, f_4, is estimate of "threat" to white king
    - etc.

Linear Evaluation Functions

- A **linear evaluation function** of the features is a weighted sum of $f_1, f_2, f_3, \ldots$
  $w_1 * f_1 + w_2 * f_2 + w_3 * f_3 + \ldots + w_n * f_n$
  - where $f_1, f_2, \ldots, f_n$ are the features
  - and $w_1, w_2, \ldots, w_n$ are the weights
- More important features get more weight

Linear Evaluation Functions

- The quality of play depends directly on the quality of the evaluation function
- To build an evaluation function we have to:
  1. construct good features using expert domain knowledge
  2. pick or learn good weights

Learning the Weights in a Linear Evaluation Function

- How could we learn these weights?
- Basic idea:
  play lots of games against an opponent
  - for every move (or game), look at the error = true outcome – evaluation function
  - if error is positive (under-estimating), adjust weights to increase the evaluation function
  - if error is 0, do nothing
  - if error is negative (over-estimating), adjust weights to decrease the evaluation function

Examples of Algorithms that Learn to Play Well

Checkers

A. L. Samuel, “Some Studies in Machine Learning using the Game of Checkers,” *IBM Journal of Research and Development*, 11(6):601-617, 1959

- Learned by playing thousands of times against a copy of itself
- Used an IBM 704 with 10,000 words of RAM, magnetic tape, and a clock speed of 1 kHz
- Successful enough to compete well at human tournaments

Examples of Algorithms that Learn to Play Well

Backgammon

G. Tesauro and T. J. Sejnowski, “A Parallel Network that Learns to Play Backgammon,” *Artificial Intelligence*, 39(3), 357-390, 1989

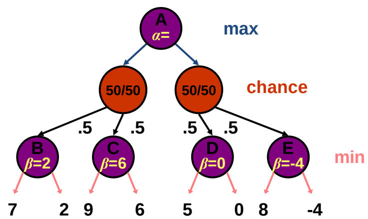
- Also learns by playing against copies of itself
- Uses a non-linear evaluation function - a neural network
- Rated one of the top three players in the world

Non-Deterministic Games

- Some games involve chance, for example:
  - roll of dice
  - spin of game wheel
  - deal of cards from shuffled deck
- How can we handle games with random elements?
- The game tree representation is extended to include “**chance nodes**:”
  1. computer moves
  2. chance nodes
  3. opponent moves

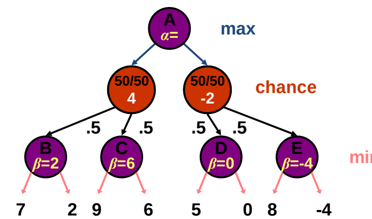
Non-Deterministic Games

Extended game tree representation:



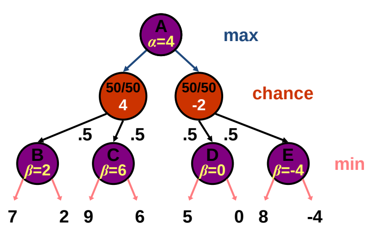
Non-Deterministic Games

- Weight score by the *probability* that move occurs
- Use expected value for move: instead of using max or min, compute the average, weighted by the probabilities of each child



Non-Deterministic Games

- Choose move with *highest expected value*



Non-Deterministic Games

- Non-determinism increases branching factor
  - 21 possible rolls with 2 dice
- *Value of look ahead diminishes*: as depth increases, probability of reaching a given node decreases
- alpha-beta pruning less effective
- TDGammon:
  - depth-2 search
  - very good heuristic
  - played at world champion level

Computers can Play GrandMaster Chess

"Deep Blue" (IBM)

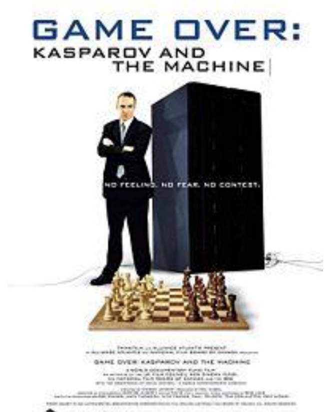
- Parallel processor, 32 "nodes"
- Each node had 8 dedicated VLSI "chess chips"
- Searched 200 million configurations/second
- Used minimax, alpha-beta, sophisticated heuristics
- Average branching factor ~6 instead of ~40
- In 2001 searched to 14 ply (i.e., 7 pairs of moves)
- Avoided horizon effect by searching as deep as 40 ply
- Used book moves

Computers can Play GrandMaster Chess

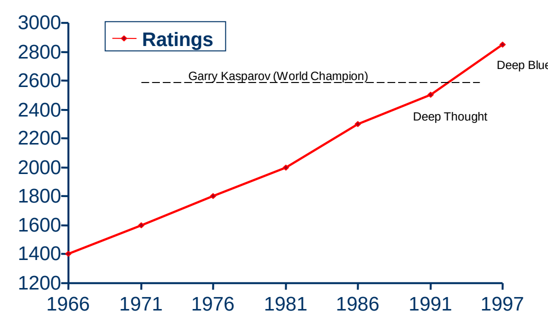
Kasparov vs. Deep Blue, May 1997

- 6 game full-regulation chess match sponsored by ACM
- Kasparov lost the match 2 wins to 3 wins and 1 tie
- Historic achievement for computer chess; the first time a computer became the best chess player on the planet
- Deep Blue played by "brute force" (i.e., raw power from computer speed and memory); it used relatively little that is similar to human intuition and cleverness

"Game Over: Kasparov and the Machine" (2003)



Chess Rating Scale



Status of Computers in Other Deterministic Games

- Checkers
  - First computer world champion: **Chinook**
  - beat all humans (beat Marion Tinsley in 1994)
  - used alpha-beta search, book moves (> 443 billion)
- Othello
  - computers easily beat world experts
- Go
  - branching factor b ~ 360, very large!
  - $2 million prize for any system that can beat a world expert

Summary

- Game playing is best modeled as a search problem
- Search trees for games represent alternate computer/opponent moves
- Evaluation functions estimate the quality of a given board configuration for each player
  – good for opponent
  0 neutral
  + good for computer

Summary

- **Minimax** is an algorithm that chooses "optimal" moves by assuming that the opponent always chooses their best move
- **Alpha-beta** is an algorithm that can avoid large parts of the search tree, thus enabling the search to go deeper
- For many well-known games, computer algorithms using heuristic search can match or out-perform human world experts