Game Playing

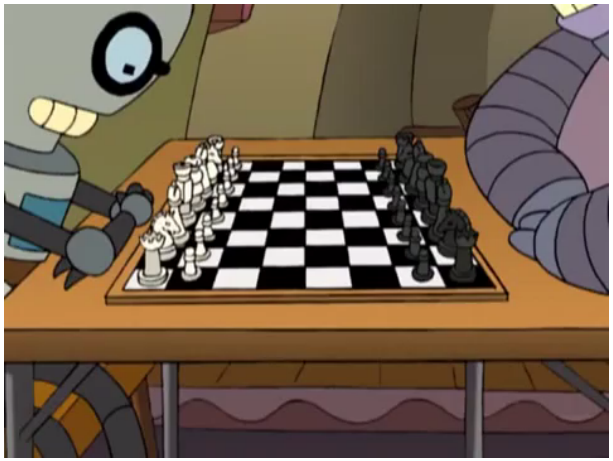
Chapter 5.1 – 5.3, 5.5

Game Playing and AI

Game playing as a 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

Computers Playing Chess



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)	Stratego, Battleship	Bridge, Poker, Scrabble

All are also multi-agent, adversarial, static tasks

Game Playing as Search

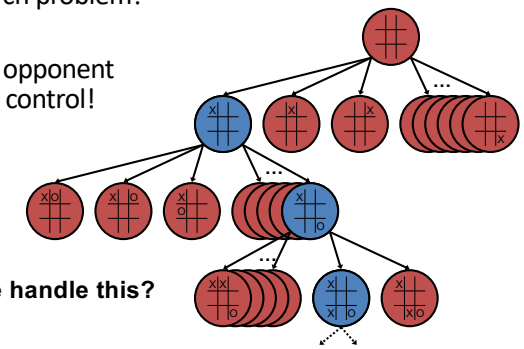
- Consider two-player, perfect information, deterministic, 0-sum board games:
  - e.g., chess, checkers, tic-tac-toe
  - Board configuration: a specific arrangement of "pieces"
- Representing board games as search problem:
  - **states:** board configurations
  - **actions:** legal moves
  - **initial state:** starting 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?

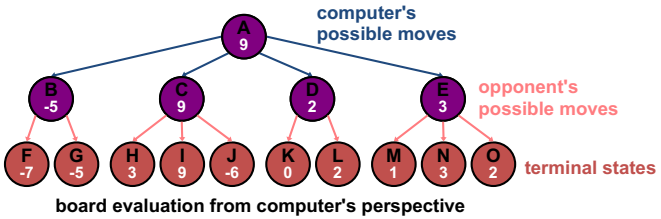


Greedy Search using an Evaluation Function

- A ***Utility function*** is used to map each **terminal state** of the board (i.e., states where the 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

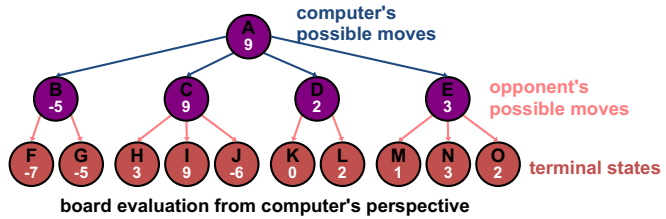
Greedy Search using an Evaluation Function

- Expand the search tree to the terminal states on each branch
- Evaluate the 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



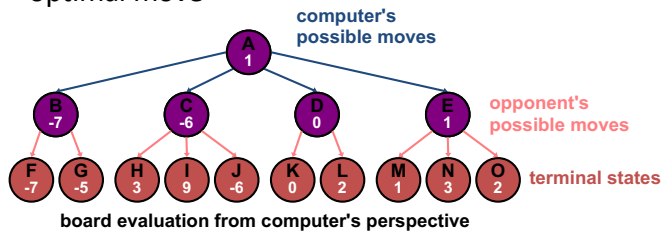
Minimax Principle

Assume *both* players play optimally

- high Utility values favor the computer
  - computer should choose *maximizing* moves
- low Utility values 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

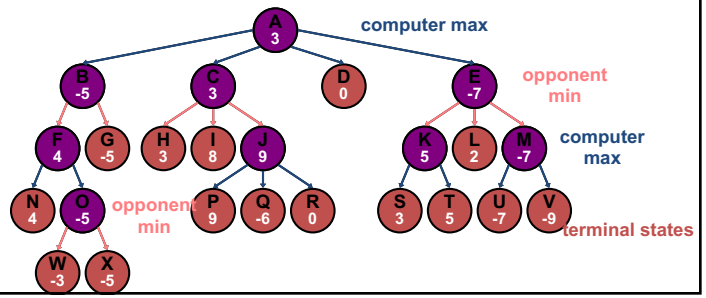


Propagating Minimax Values Up the Game Tree

- Explore the tree to reach terminal states
- Evaluate the 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(s):
  - start at the leaves
  - assign value to the parent node as follows
    - use ***minimum*** when node is the opponent's move
    - use ***maximum*** when node is the computer's move

Deeper Game Trees

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



Minimax Algorithm

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

Search algorithm independently invented by Claude Shannon (1950) and Alan Turing (1951)

Complexity of Minimax Algorithm

Assume all terminal states are at depth d and there are b possible moves at each step

- 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 limited 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 per game
    - 5^9 = 1,953,125 states
  - **Chess:** ~35 legal moves, ~100 moves per game
    - b^d ~ 35^{100} ~10^{154} states, only ~10^{40} legal states
  - **Go**: ~250 legal moves, ~150 moves per game
- Common games produce ***enormous*** search trees

- Start here

Complexity of Minimax Algorithm

- Minimax algorithm applied to *complete* game trees is impractical in practice
  - instead do depth-limited search to **ply** (depth) m
  - but Utility function is 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 Evaluation

- A **Static Board Evaluation (SBE) 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 a 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.**

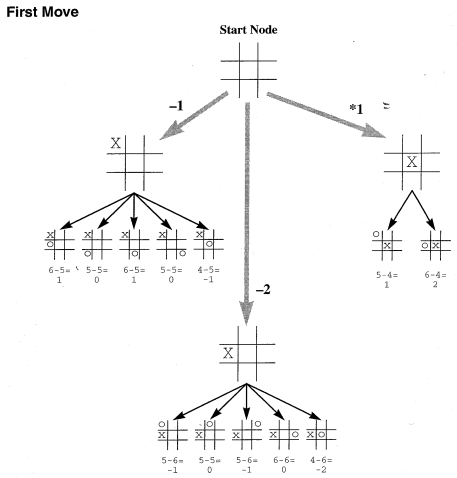
Static Board Evaluation

- 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

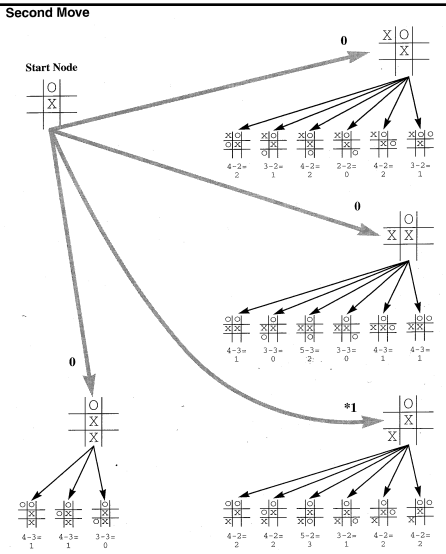
Minimax with Evaluation Functions

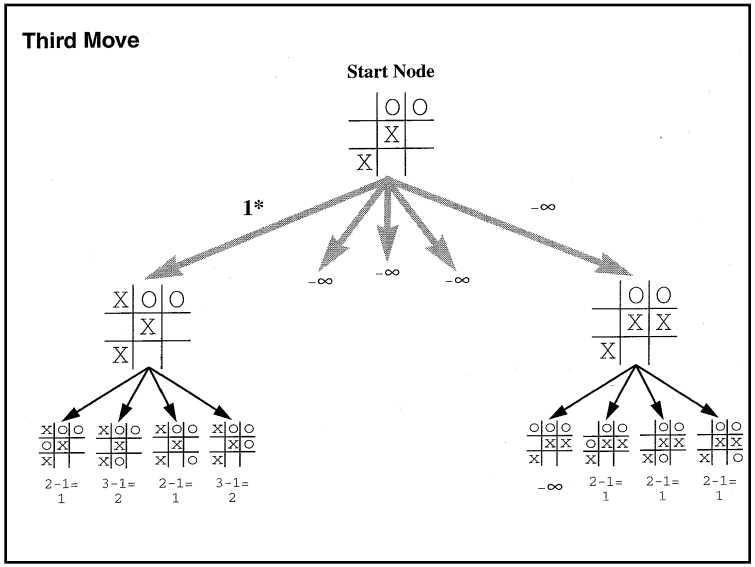
- The same as general Minimax, except
  - only go to depth m
  - estimates value at leaves using the SBE function
- How would this algorithm perform at Chess?
  - if could look ahead ~4 pairs of moves (i.e., 8 ply), would be consistently beaten by average players
  - if could look ahead ~8 pairs, is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

Tic-Tac-Toe Example



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



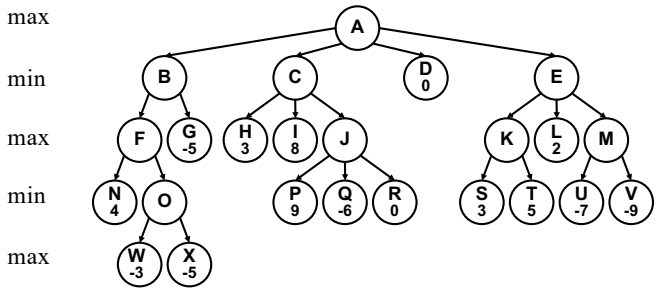


Minimax Algorithm

```
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
            v = – ∞
            foreach s' in Successors(s)
                v = max( v , Min-Value(s'))
    return v
```

```
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
            v = +∞
            foreach s' in Successors(s)
                v = min( v , Max-Value(s'))
    return v
```

Minimax Example



Summary So Far

- Can't use Minimax search to end of the game
  - if we could, then choosing optimal 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:
  - Use Minimax to cutoff search at depth m
  - use SBE to estimate/score board configuration

Alpha-Beta Idea

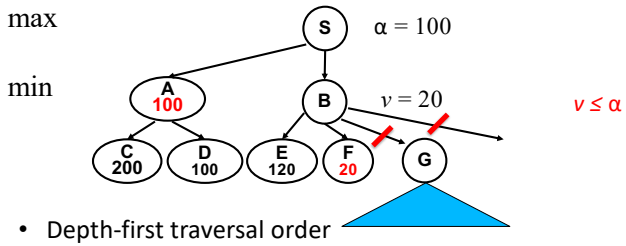
- Some of the branches of the game tree won't be taken if playing against an intelligent opponent
- "If you have an idea that is surely bad, don't take the time to see how truly awful it is."
  -- Pat Winston
- ***Pruning* can be used to ignore some branches**
- While doing DFS of game tree, keep track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

- Also keep track of

  α = best already explored option along the path to the root for MAX, including the current node

  β = best already explored option along the path to the root for MIN, including the current node

Alpha-Beta Idea: Alpha Cutoff

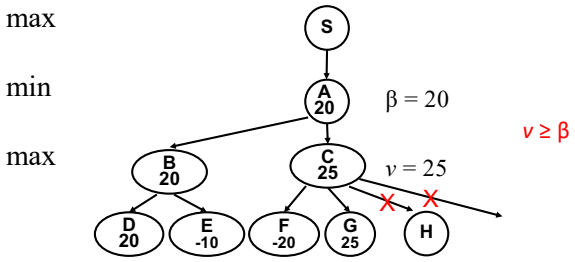


- Depth-first traversal order
- After returning from A, can get ***at least 100 at S***
- After returning from F, can get ***at most 20 at B***
- At this point no matter what minimax value is computed at G, S will prefer A over B. So, S loses interest in B
- There is no need to visit G. The subtree at 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 *v*
- Each time *v* is updated (at a MIN node), check its value against the α value of all its MAX node ancestors
- If ***v* ≤ α** for *some* MAX node ancestor, **don't visit** any more of the current MIN node's children; i.e., ***prune (cutoff) all subtrees rooted at remaining children of MIN***

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)

Beta Cutoff

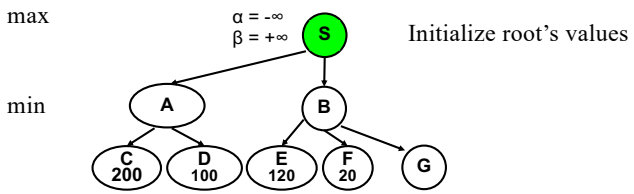
- At each MAX node, keep track of the *maximum* value returned so far from its visited children
- Store this value as *v*
- Each time *v* is updated (at a MAX node), check its value against the β value of all its MIN node ancestors
- If ***v* ≥ β** for *some* MIN node ancestor, **don't visit** any more of the current MAX node's children; i.e., ***prune (cutoff) all subtrees rooted at remaining children of MAX***

Implementation of Cutoffs

At each node, keep *both* α and β values, where α = largest (i.e., best) value of all MAX node *ancestors* in search tree, and β = smallest (i.e., best) value of all MIN node *ancestors* in search tree. Pass these *down the tree* during traversal

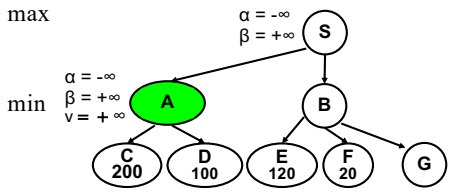
- At MAX node, *v* = largest value from its ***children*** visited so far; cutoff if $v \geq \beta$
  - ***v* value at MAX comes from its *descendants***
  - **β value at MAX comes from its MIN node *ancestors***
- At MIN node, *v* = smallest value from its ***children*** visited so far; cutoff if $v \leq \alpha$
  - **α value at MIN comes from its MAX node ancestors**
  - ***v* value at MIN comes from its descendants**

Implementation of Alpha Cutoff

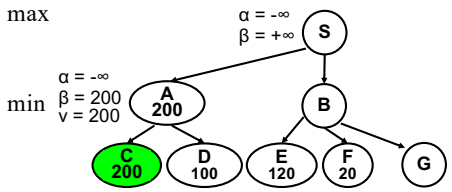


- At each node, keep two bounds (based on all *nodes* on path back to root):
  - α: the best (largest) MAX can do at any ancestor
  - β: the best (smallest) MIN can do at any ancestor
- *v*: the best value returned by current node's visited *children*
- If at anytime $\alpha \geq v$ at a MIN node, the remaining children are pruned (i.e., not visited)

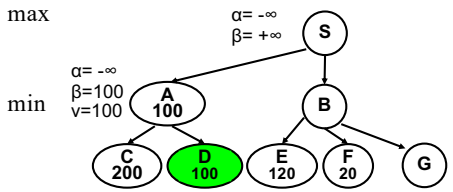
Alpha Cutoff Example



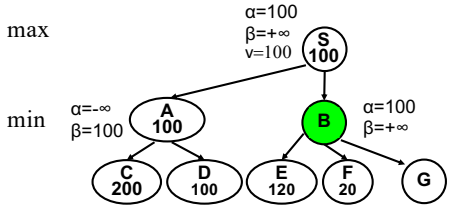
Alpha Cutoff Example



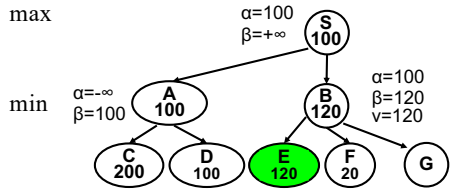
Alpha Cutoff Example



Alpha Cutoff Example

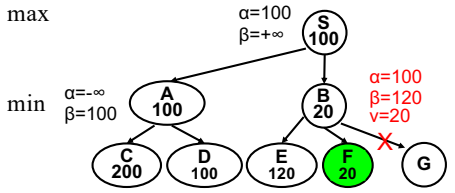


Alpha Cutoff Example



v > α so continue to next child of B

Alpha Cutoff Example



$v = 20 \leq \alpha = 100$ so prune (i.e., don't visit G)

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

Alpha-Beta Cutoffs

- At a MAX node, if $v \geq \beta$ then don't visit (i.e., cutoff) remaining children of this MAX node
  - *v* is the max value found so far from visiting current MAX node's children
  - β is the best value found so far at any MIN node ancestor of the current MAX node
- At a MIN node, if $v \leq \alpha$ then don't visit remaining children of this MIN node
  - *v* is the min value found so far from visiting current MIN node's children
  - α is the best value found so far at any MAX node ancestor of the current MIN node

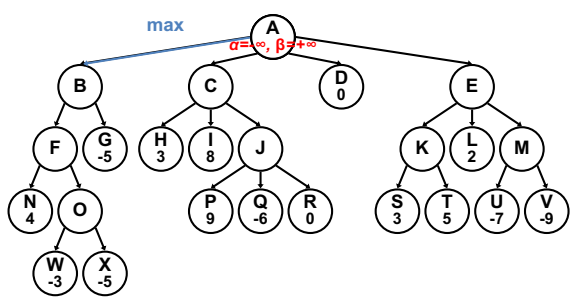
Alpha-Beta Algorithm

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

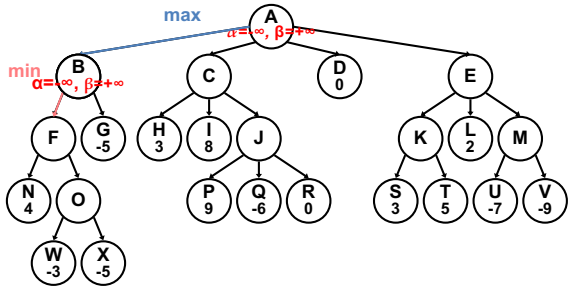
```
function Max-Value (s, α, β)
inputs:
   s: current state in game, Max about to play
   α: best score (highest) for Max along path from s to root
   β: best score (lowest) for Min along path from s to root
   if ( s is a terminal state or at depth limit )
   then return ( SBE value of s )
   v = -∞
   for each s' in Successors(s)
      v = max( v, Min-Value(s', α, β))
      if (v ≥ β ) then return v    // prune remaining children
      α = max(α, v)
   return v                        // return value of best child
```

```
function Min-Value(s, α, β)
  if ( s is a terminal state or at depth limit )
  then return ( SBE value of s)
  v = +∞
  for each s' in Successors(s)
    v = min( v, Max-Value(s', α, β))
    if (v ≤ α ) then return v    // prune remaining children
    β = min(β, v)
  return v                       // return value of best child
```

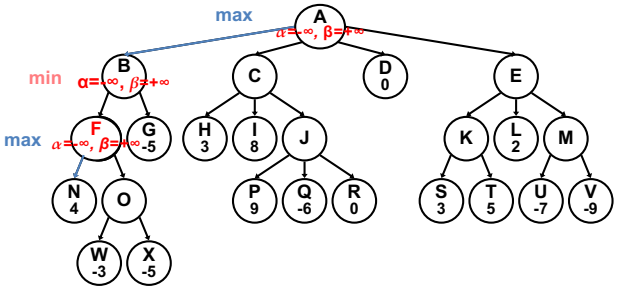
Alpha-Beta Example



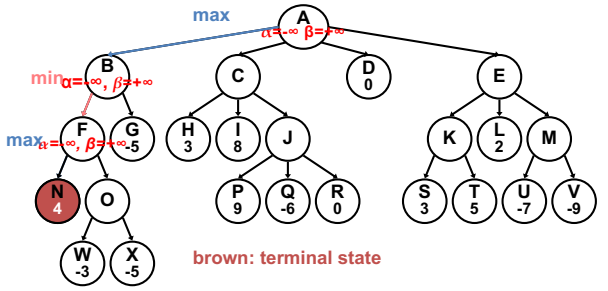
Alpha-Beta Example



Alpha-Beta Example

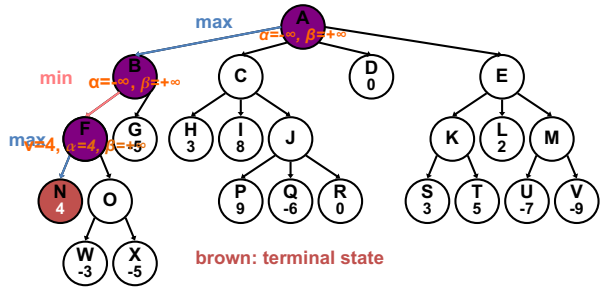


Alpha-Beta Example

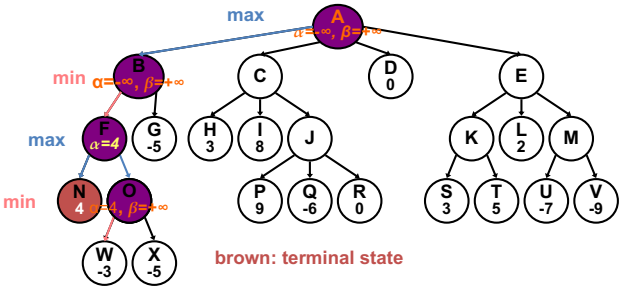


Alpha-Beta Example

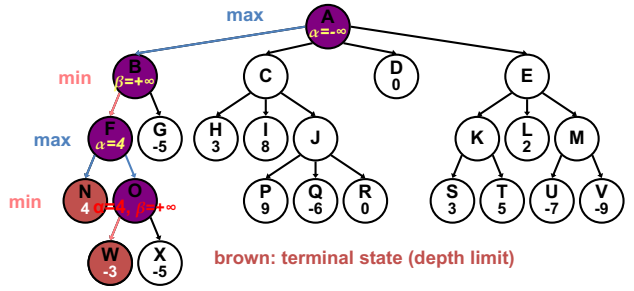
`v(F) = alpha(F) = 4`, maximum seen so far



Alpha-Beta Example

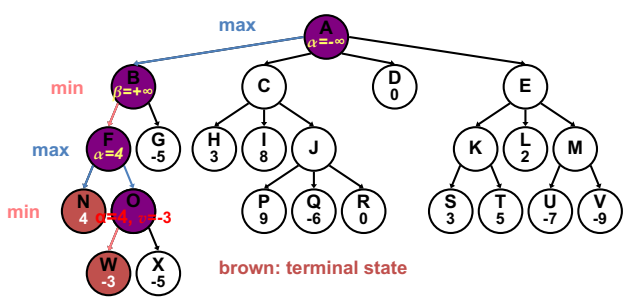


Alpha-Beta Example



Alpha-Beta Example

v(O) = −3, minimum seen so far below O

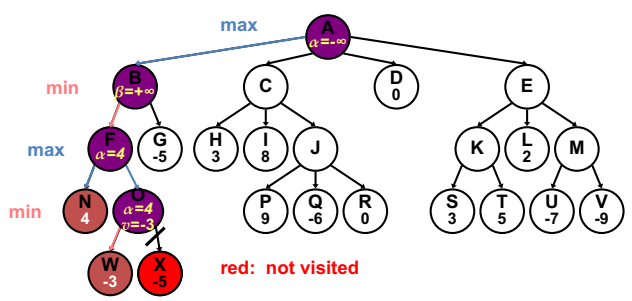


Call Stack

O
F
B
A

Alpha-Beta Example

v(O) = -3 ≤ alpha(O) = 4: stop expanding O (cutoff)

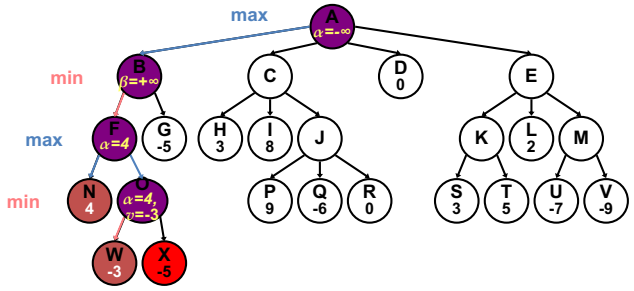


Call Stack

O
F
B
A

Alpha-Beta Example

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

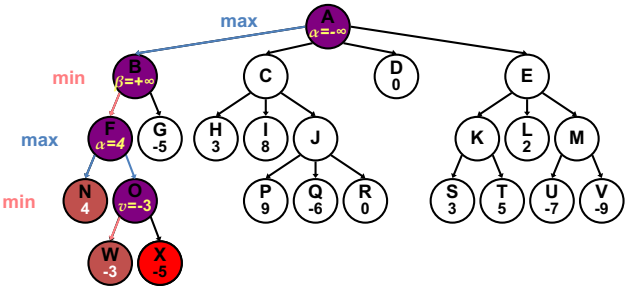


Call Stack

O
F
B
A

Alpha-Beta Example

v(F) = alpha(F) = 4 not changed (maximizing)

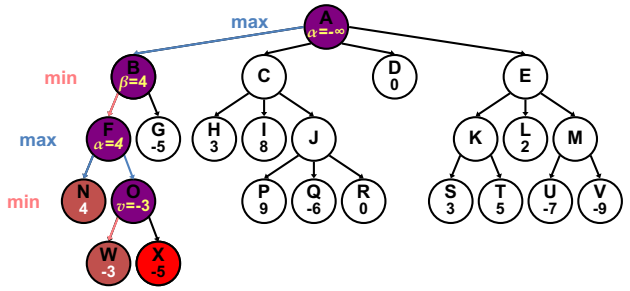


Call Stack

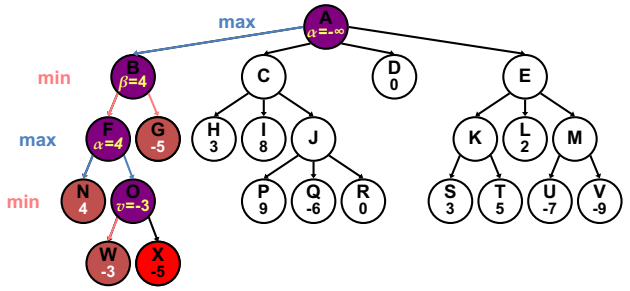
F
B
A

Alpha-Beta Example

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

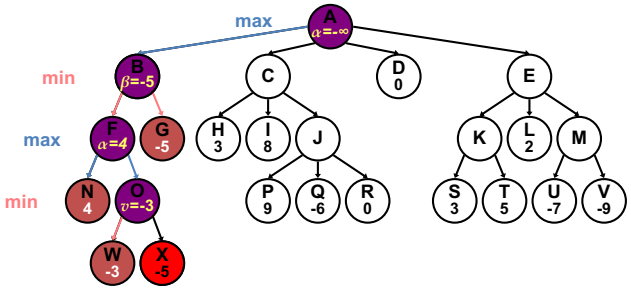


Alpha-Beta Example



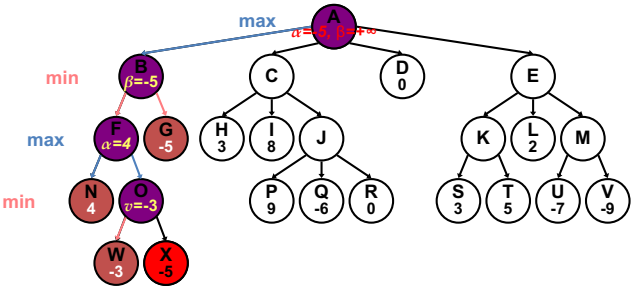
Alpha-Beta Example

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



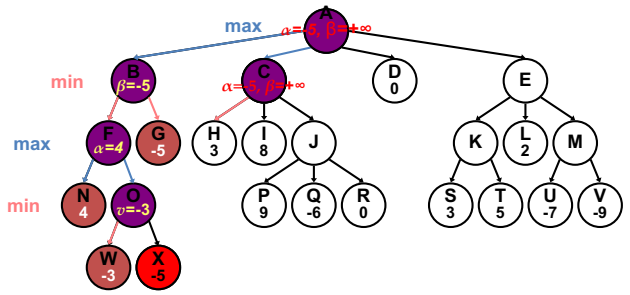
Alpha-Beta Example

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

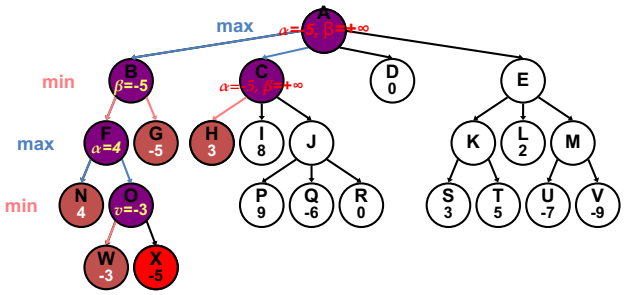


Alpha-Beta Example

Copy alpha and beta values from A to C

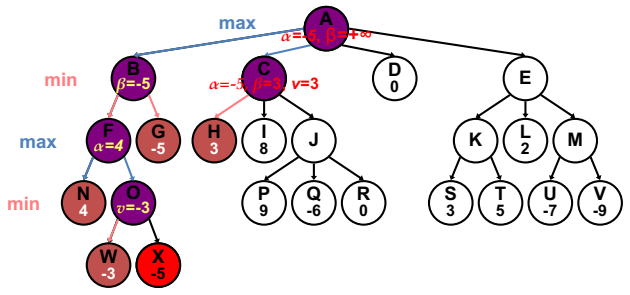


Alpha-Beta Example

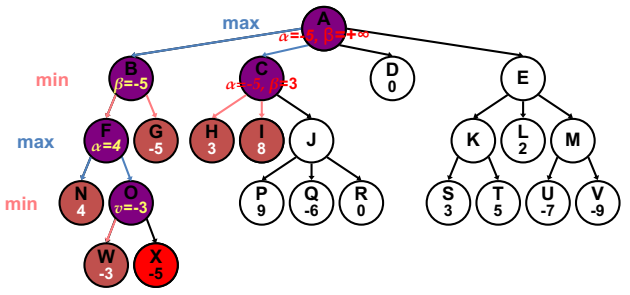


Alpha-Beta Example

`v(C) = beta(C) = 3`, minimum seen so far
v(C) (= 3) > α(C) (= -5), so ***no*** cutoff

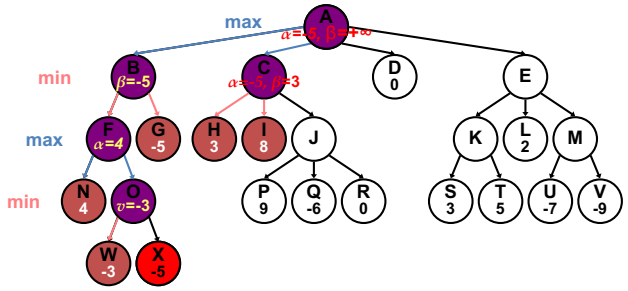


Alpha-Beta Example

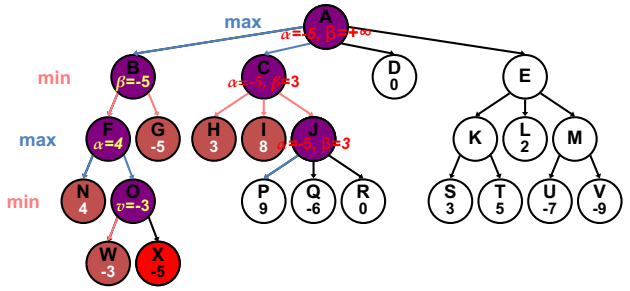


Alpha-Beta Example

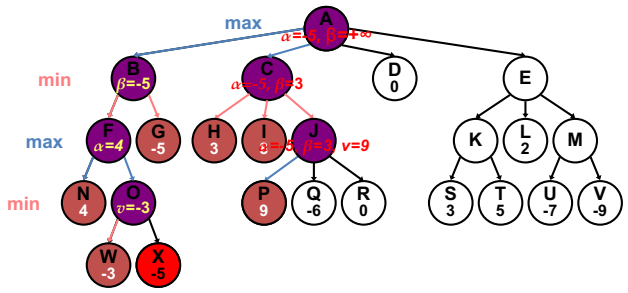
beta(C) not changed (minimizing)



Alpha-Beta Example

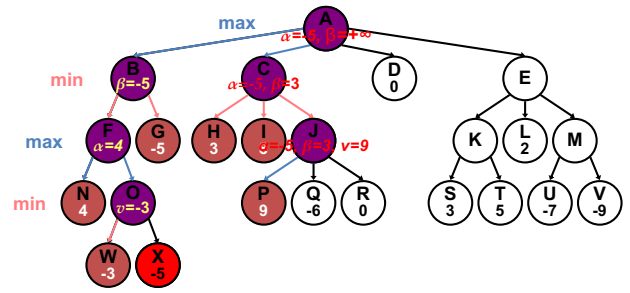


Alpha-Beta Example



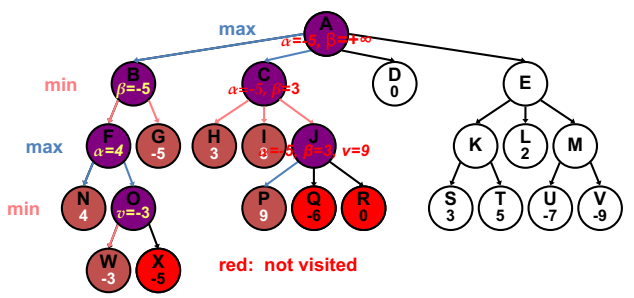
Alpha-Beta Example

v(J) = 9



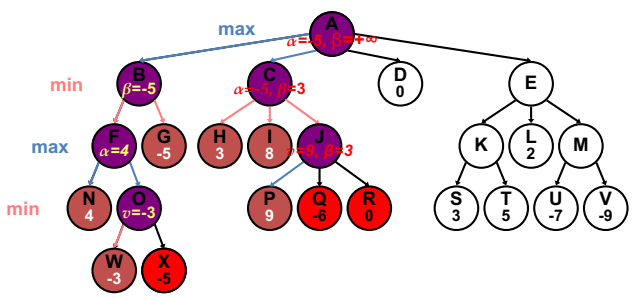
Alpha-Beta Example

v(J) (= 9) ≥ beta(J) (= 3) so stop expanding J (beta cutoff)



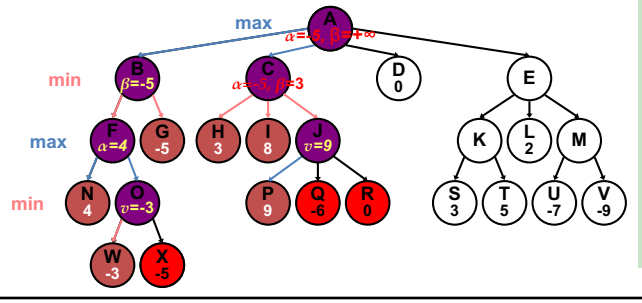
Alpha-Beta Example

Why? Computer should choose P or better at J, so J's lower bound is 9. But, smart opponent at C won't take J:9 since H:3 is better for opponent.



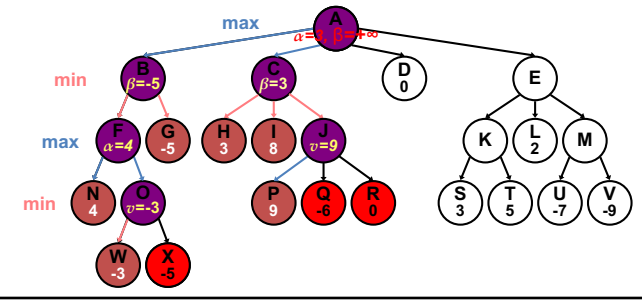
Alpha-Beta Example

v(C) and beta(C) not changed (minimizing)

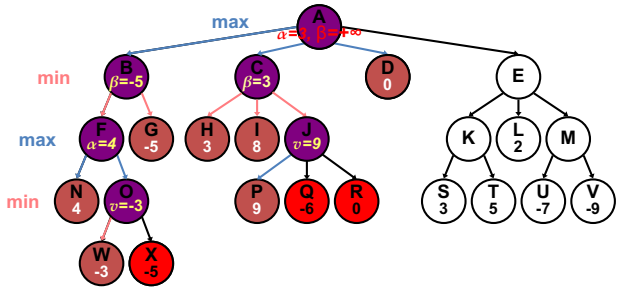


Alpha-Beta Example

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



Alpha-Beta Example



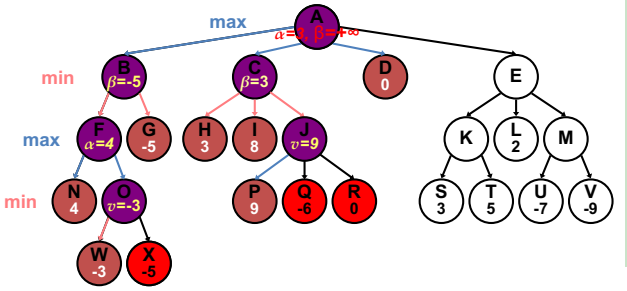
Call Stack

D

A

Alpha-Beta Example

alpha(A) and v(A) not updated after returning from D
(because A is a maximizing node and 0 < 3)

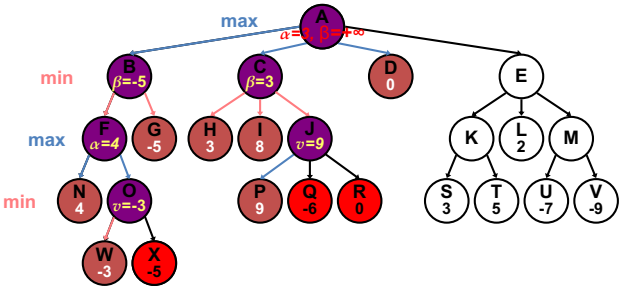


Call Stack

A

Alpha-Beta Example

How does the algorithm finish the search tree?



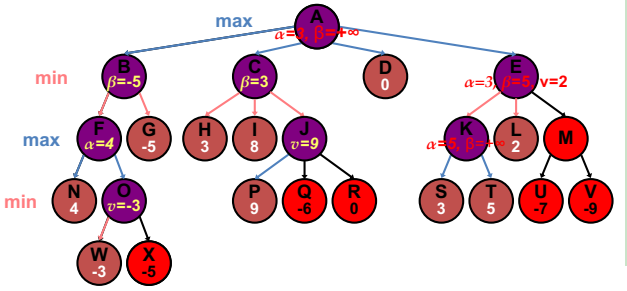
Call Stack

A

Alpha-Beta Example

After visiting K, S, T, L and returning to E,

v(E) (= 2) ≤ alpha(E) (= 3) so stop expanding E
and don't visit M (alpha cutoff)

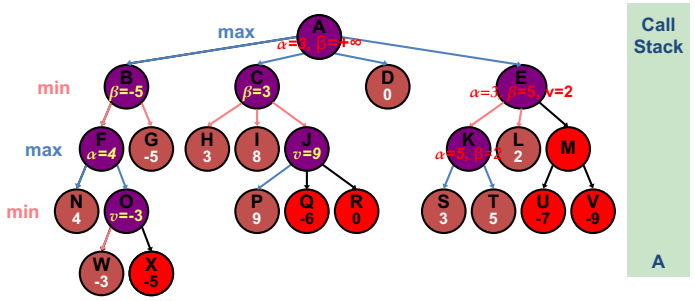


Call Stack

A

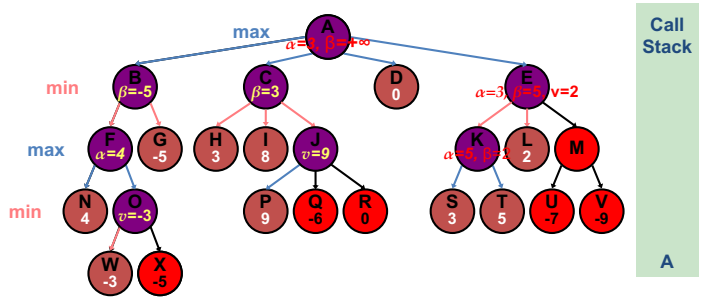
Alpha-Beta Example

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



Alpha-Beta Example

Final result: Computer chooses move C



Another step-by-step example (from AI course at UC Berkeley) given at

https://www.youtube.com/watch?v=xBXHtz4Gbdo

Effectiveness of Alpha-Beta Search

- Effectiveness (i.e., amount of pruning) depends on the *order* in which successors are examined
- Worst Case:
  - ordered so that ***no*** pruning takes place
  - no improvement over exhaustive search
- Best Case:
  - each player's *best* move is visited *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
  - Deep Blue went from ***b* ~ *35*** to **b ~ 6**, visiting 1 billionth the number of nodes visited by the Minimax algorithm
  - 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 deal with this?
  - cannot stop alpha-beta algorithm midway and expect to use results with any confidence
  - Solution #1: set a (conservative) ***depth-limit*** that guarantees we will finish in time < *T*
  - but the search may finish very early and the opportunity is lost to do more searching

Dealing with Limited Time

Solution #2: use **iterative deepening search** (IDS)

- run alpha-beta search with **depth-first search and an *increasing depth-limit***
- when time runs out, use the solution found for the last completed alpha-beta search (i.e., the deepest search that was completed)
- **"anytime algorithm"**

The Horizon Effect

- Sometimes disaster lurks just beyond the 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 the depth-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 or machine learning
  2. pick or learn good weights

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

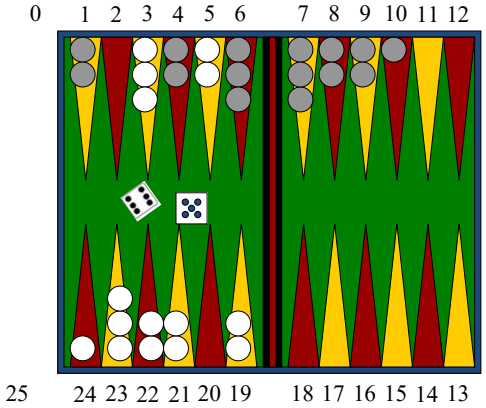
Examples of Algorithms that Learn to Play Well

Backgammon

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

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

Non-Deterministic Games

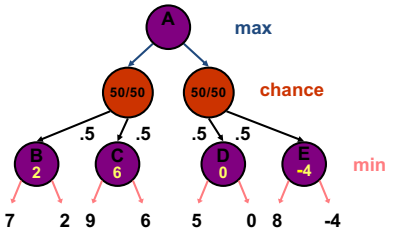


Non-Deterministic Games

- Some games involve chance, for example:
  - roll of dice
  - spin of game wheel
  - deal cards from a shuffled deck
- How can we handle games with random elements?
  - Modify the game search tree to include "**chance nodes**:"
  1. computer moves
  2. chance nodes (representing random events)
  3. opponent moves

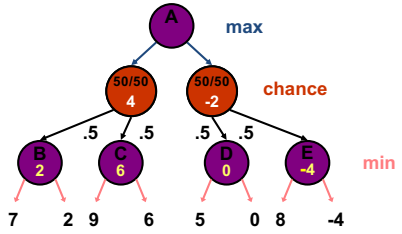
Non-Deterministic Games

Extended game tree representation:



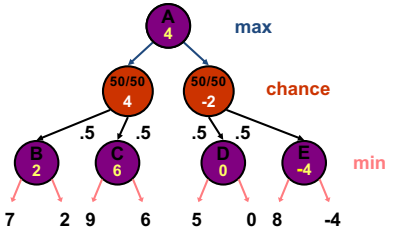
Non-Deterministic Games

- 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 the ***highest expected value***



Expectiminimax

expectiminimax(n) =

SBE(n)	for n, a Terminal state or state at cutoff depth
$max_{s \in Succ(n)}$ expectiminimax(s)	for n, a Max node
$min_{s \in Succ(n)}$ expectiminimax(s)	for n, a Min node
$\Sigma_{s \in Succ(n)} P(s) *$ expectiminimax(s)	for n, a Chance node

Non-Deterministic Games

- Non-determinism increases branching factor
  - 21 possible distinct rolls with 2 dice (since 6-5 is same as 5-6)
- *Value of look-ahead diminishes*: as depth increases, probability of reaching a given node decreases
- Alpha-Beta pruning is less effective

Computers Play GrandMaster Chess

"Deep Blue" (1997, IBM)

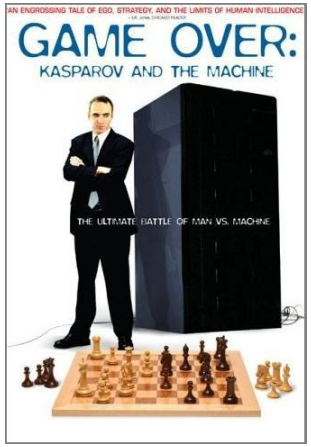
- Parallel processor, 32 "nodes"
- Each node had 8 dedicated VLSI "chess chips"
- Searched 200 million configurations/second
- Used minimax, alpha-beta, sophisticated heuristics
- Average branching factor ~6 instead of ~40
- In 2001 searched to 14 ply (i.e., 7 pairs of moves)
- Avoided horizon effect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

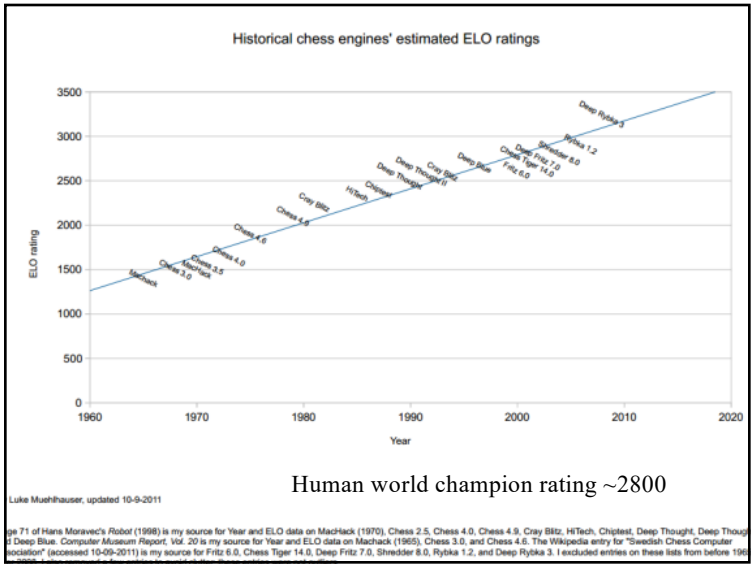
Computers can Play GrandMaster Chess

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)





Human world champion rating ~2800

Status of Computers Playing Other Games

- Checkers
  - First computer world champion: **Chinook**
  - Beat all humans (beat Marion Tinsley in **1994**)
  - Used Alpha-Beta search and book moves
- Othello
  - Computers easily beat world experts
- Go
  - Branching factor b ~ 360, very large!

Game Playing: Go

Google's AlphaGo beat Korean grandmaster Lee Sedol 4 games to 1 in 2016

AlphaGo Documentary Movie (2017)

Game Playing Summary

- Game playing is modeled as a search problem, doing a limited look-ahead (depth bound)
- Search trees for games represent both computer and opponent moves
- A single evaluation functions estimates the quality of a given board configuration for both players (zero-sum assumption)
  - − good for opponent
  - 0 neutral
  - + good for computer

Summary

- **Minimax** algorithm determines the "optimal" moves by assuming that both players always chooses their best move
- **Alpha-beta** algorithm 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 experts

How to Improve Performance?

- Reduce depth of search
  - Better SBEs
    - Use machine learning to learn good features rather than use "manually-defined" features
- Reduce breadth of search
  - Explore a *subset* of the possible moves instead of exploring all
    - Use ***randomized exploration*** of the search space

Monte Carlo Tree Search (MCTS)

- Concentrate search on *most promising moves*
- Best-first search based on **random sampling of search space**

- **Monte Carlo methods** are a broad class of algorithms that rely on repeated random sampling to obtain numerical results. They can be used to solve problems having a probabilistic interpretation.

Pure Monte Carlo Tree Search

- For each possible legal move of current player, **simulate *k* random games** by selecting moves *at random* for both players until game over (called ***playouts***); count how many were wins out of each *k* playouts; move with most wins is selected
- **Stochastic simulation** of game
- Game must have finite number of possible moves, and game length is finite

Exploitation vs. Exploration

- Rather than selecting a child at random, how to select *best* child node during tree descent?
  - **Exploitation**: Keep track of average win rate for each child from previous searches; prefer child that has previously lead to more wins
  - **Exploration**: Allow for exploration of relatively unvisited children (moves) too
- Combine these factors to compute a "score" for each child; pick child with highest score at each successive node in search

Scoring Child Nodes

- Choose child node, *i*, with highest score
- One way of defining the score at a node:
  - Upper Confidence Bound for Trees (UCT):

Exploitation term

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

Exploration term

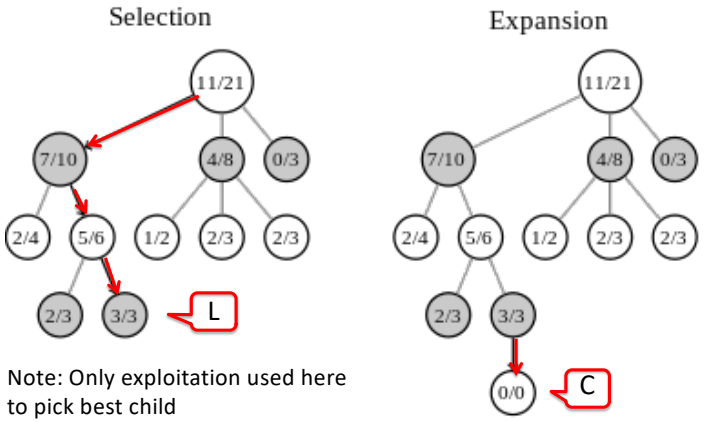
where w_i = number of wins after i^{th} move,
n_i = number of playout simulations after i^{th} move,
t = total number of playout simulations

MCTS Algorithm

Recursively build search tree, where each round consists of:

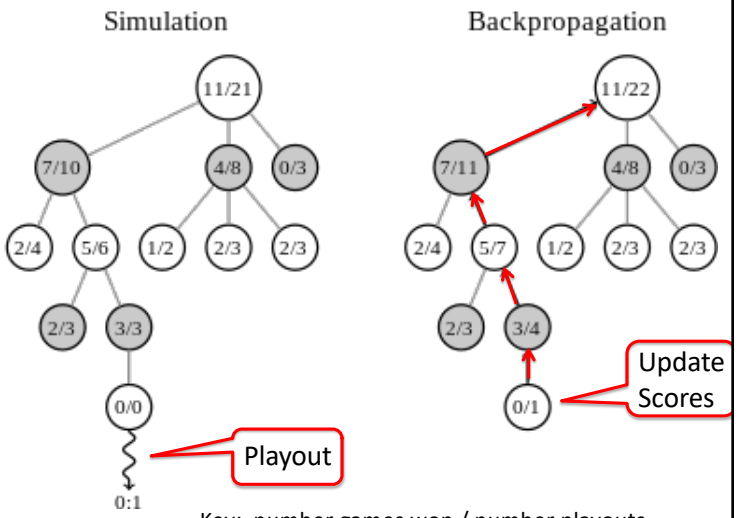
1. Selection: Starting at root, successively select best child nodes using scoring method until leaf node *L* reached
2. Expansion: Create and add best (or random) new child node, *C*, of *L*
3. Simulation: Perform a (random) playout from *C*
4. Backpropagation: Update score at *C* and all of *C*'s ancestors in search tree based on playout results

Monte Carlo Tree Search (MCTS)



Note: Only exploitation used here to pick best child

Key: number games won / number playouts



Key: number games won / number playouts

State-of-the Art Go Programs

- Google's AlphaGo
- Facebook's Darkforest
- MCTS implemented using multiple threads and GPUs, and up to 110K playouts
- Also used a deep neural network to compute SBE