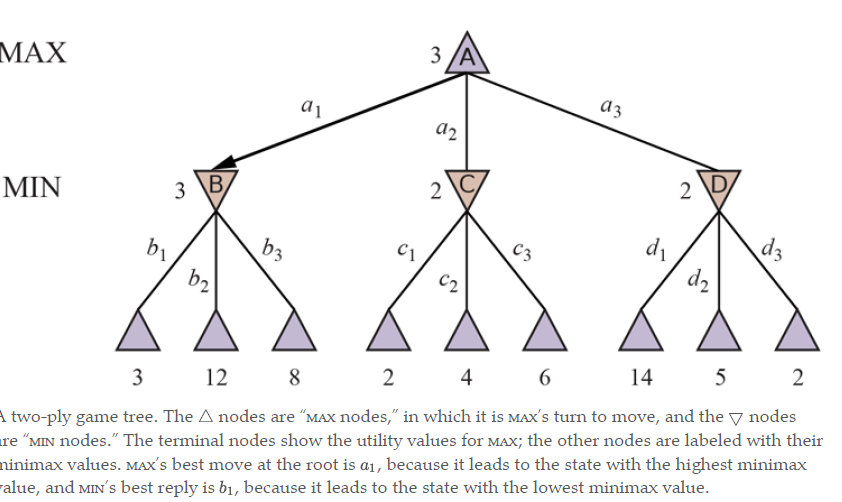
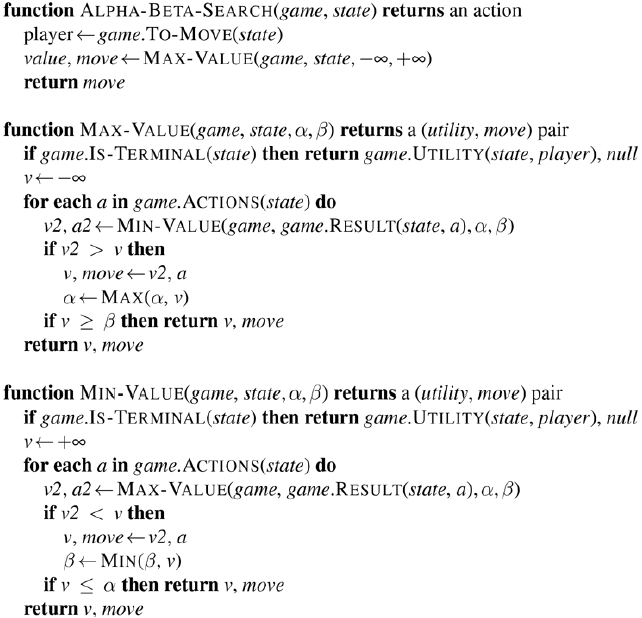
* **Multiagent environments**
  + Consider them in aggregate as an economy (appropriate for large number of agents)
    - Allows predictions like ‘increasing demand will cause prices to rise’ without having to predict action of any individual agent
  + Consider adversarial agents as part of the environment that makes it nondeterministic
    - But if we model adversaries nondeterministically(e.g.rain may fall, may not), we miss the idea that our adversaries are actively trying to defeat us
  + Explicitly model adversarial agent with techniques of **adversarial game-tree search**
* **2-player zero-sum games**
  + 2 player, turn-taking, perfect information, zero-sum games
    - Zero-sum: what’s good for one player is bad for the other
    - Perfect information: fully observable
  + Formal model
    - *S0:* Initial state
    - TO-MOVE(*s*): player whose turn it is to move
    - ACTIONS(*s*); set of legal moves in state *s*
    - RESULT(*s*, *a*): transition model
    - IS-TERMINAL(*s*): terminal test, true if the game is over or false otherwise
    - UTILITY(*s, p*): utility function that defines final numeric value to player *p* when game ends in terminal state *s*
  + ACTIONS and RESULT define state space graph
    - Can superimpose search tree over the graph
    - **Game tree**: search tree that follows every sequence of moves until terminal state
      * Might be infinite if state space is unbounded or rules allow infinite repeating positions
* **Optimal Decision Making -** strategy must be conditional and have contingencies specifying response to adversary’s moves
  + Can use AND-OR search for games with binary (win/lose) outcome for plan
    - Definition of winning strategy for game is identical to definition of solution for nondeterministic planning problem
  + **Minimax search:** For games with multiple outcome scores, need a more general algorithm
    - **Ply**: one move by one player
    - **Minimax value** (MINIMAX(*s*))**:** Utility for player being in that state assuming both players play optimally from there to the end of the game
      * Minimax value of terminal state is just its utility
      * Assumes both agents only play optimally

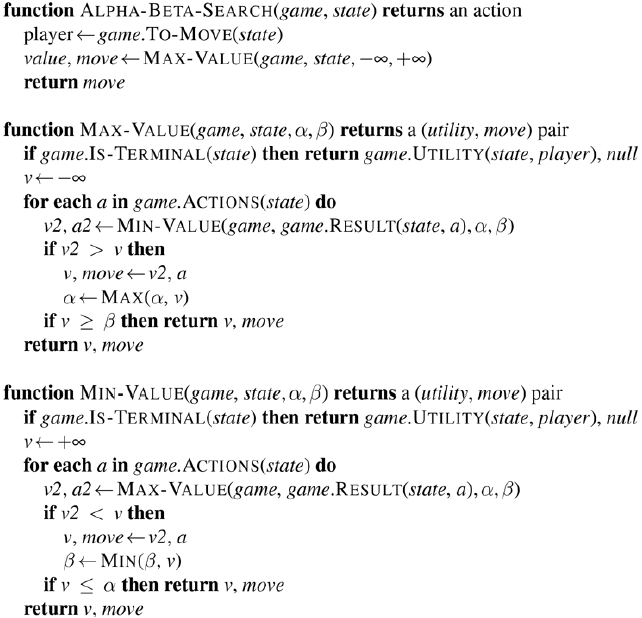
**Minimax Value Example** Max, Min. Max wants higher number Min wants lower number

* MINIMAX(B) = 3 
  + B is Min’s move, Min wants lowest number, so it would select 3
* MINIMAX(C) = 2
  + C is Min’s move, lower number
* MINIMAX(D) = 2
  + C is Min’s move
* MINIMAX(A) = 3
  + A is Max’s move, wants highest number
  + **Minimax decision:** a1 is the best move because it leads to the state with highest minimax value

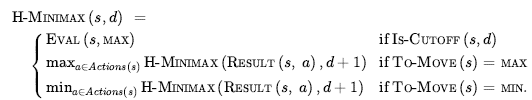
**Doesn’t always mean it’s best to play minimax optimal move when facing suboptimal opponent**

* **Minimax search algorithm** - finds best move by trying all actions and choosing the one whose resulting state has the highest MINIMAX value
  + Recursive that proceeds all the way down to leaves of tree and backs up minimax values through tree as recursion unwinds
  + Complete depth-first exploration
  + Space time complexity
    - *m*: maximum depth of tree
    - *b*: legal moves at each point
    - Time: O(*bm*)
    - Space: O(*bm*) if algorithm generates actions all at once
      * O(*m*) if generates actions one at a time
  + Impractical for complex games
* **Minimax for Multiplayer games**
  + Utility vector (v1, v2, v3,...vn)
    - In two player zero sum games two element vector can be a single value because values are always opposite
  + For terminal states, vector is utility of the state from each player’s viewpoint
  + For leaves, it’s the **backed up value**
    - Backed value of a node *n* is the utility vector of the successor state with the highest value of the player choosing *n*
  + **Alliances can be a natural consequence of optimal strategies for each player in multiplayer game**
    - Collaboration can emerge from purely selfish behavior
    - Players must balance immediate advantage of breaking an alliance against long term disadvantage of being perceived as untrustworthy
    - If game isn’t zero-sum collaboration can occur with just 2 players
* **Alpha-beta pruning** - pruning large parts of the game tree that make no difference to the outcome
  + Value of the root and minimax decision are independent of the values of the leaves, so some leaves can be pruned
  + Possible to prune entire subtrees rather than just leaves
  + General principle
    - Consider node *n* such that player has a choice of moving to *n*
    - If player has better choice either at the same level or at any point higher up in the tree, player will never move to *n*
    - Once we’ve found enough about *n* by examining its descendants to reach this conclusion, we can prune *n*
  + Gets name from two parameters that describe bounds on backed-up values that appear anywhere on the path
    - - value of the best (highest) choice we have found so far at any choice point along the path (at least)
    - - value of the best choice (lowest) choice we’ve found so far at any choice point along the path (at most)
  + Alpha-beta search updates values of and as it goes along and prunes remaining branches as soon as the value of the current node is known to be worse than current or for the respective player
* **Effectiveness of alpha-beta pruning is dependent on the order in which states are examined**
  + Might be worthwhile to try and examine the successors that are likely to be best
    - If done perfectly alpha-beta would only need to examine O(*bm/2*) nodes instead of O(*bm*)
    - Effective branching factor becomes instead of *b*
    - Alpha-beta with perfect move ordering can solve a tree roughly twice as deep as minimax in the same amount of time
  + With random move ordering total number of nodes examined is O(*b3m/4*)
  + Perfect move ordering is impossible, but simple ordering functions help too
  + Dynamic-move ordering schemes (trying first moves that were found to be best in past) helps a lot
    - Could come from previous exploration of current move through a process of iterative deepening
      * Search one ply deep, record ranking of moves based on evaluations
      * Then one ply deeper, using previous ranking to inform move ordering
    - Increased time from iterative deepening can be made up from better move ordering
  + **Killer move heuristic:** finding the best move first
* **Transpositions** - different permutations of the move sequence that end up in the same position (repeated states)
  + **Transposition table**: Caches heuristic value of states
  + After exploring subtree below some state, can find its backed-up minimax value, which is stored in transposition table
    - Later when we end up in same state from a different set of moves, we can look up the value instead of repeating search
* **Even with alpha-beta pruning and move ordering, minimax often doesn’t work because there are too many states to explore in the time available**
  + **Type A strategy**: Explore a wide but shallow portion of the tree
    - Consider all possible moves to a certain depth in search tree and uses heuristic evaluation to estimate utility of states at that depth
    - Most chess programs have been Type A
  + **Type B strategy:** Explore a deep but narrow portion of tree
    - Ignores moves that look bad and follow promising lines as far as possible
    - Most Go programs are Type B
    - Type B programs show higher level of play

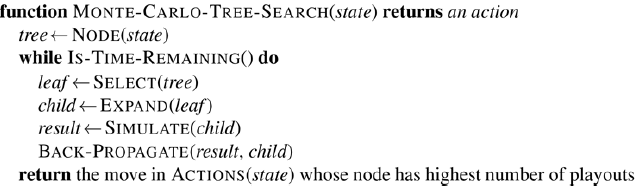




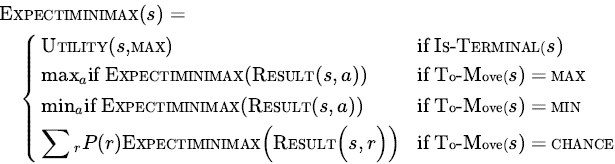
* **Heuristic Alpha-Beta Tree Search**
  + Can cut off search early and apply heuristic evaluation function to states
    - Treat nonterminal nodes as if they were terminal
  + Replace utility function with **evaluation functions**
    - Estimates a state’s utility
  + Replace terminal test with **cutoff test**
    - Decides when to cut off the search based on search dept and any property of the state that it chooses to consider
      * Must return true for terminal states



* **Heuristic evaluation function** EVAL(s, p) **returns an estimate of the expected utility of the state *s* to player *p***
  + For terminal states
    - EVAL(*s, p*) = UTILITY(*s, p*)
  + For nonterminal states
    - UTILITY(*loss*, *p*) EVAL(*s, p*) UTILITY(*win*, *p*)
  + What makes for a good evaluation function?
    - Can’t take too long
    - **Should strongly correlate with actual chances of winning**
      * If the search must be cut off at nonterminal states, algorithm will necessarily be uncertain about final outcomes of those states
      * Doesn’t need to be a linear correlation
      * **If state *s* is twice as likely to win as *s’* it’s not required that EVAL(*s*) is twice of EVAL(*s’*). Only EVAL(*s*) > EVAL(*s*’)**
  + Most eval functions work by calculating various features of the state
    - Features taken together define various equivalence classes of the state
      * States in each category have the same values for all features
  + Does not know which states are watch but returns a single value that estimates the proportion of states with each outcome.
    - **Expected value**: determined for each category of states resulting in eval function that works for any state
  + **Most eval functions compute separate numerical contributions from each feature and then combine them to find the total value**
    - E.g. material value in chess (pawn = 1, knight = 3, etc.)
  + **Weighted linear function**
    - fi = feature of the position
    - wi = weight of the position
    - Weights should be normalized so that sum is within range of zero (loss) to +1 (win)
  + Adding up values of features assumes **that the contribution of each feature is independent of the values of the other features**
    - Other games use nonlinear combinations of feature
      * E.g. a bishop in chess is worth more in the endgame
* **Cutting off search**
  + Call heuristic EVAL when it’s time to cut off the search
    - IS-TERMINAL -> IS-CUTOFF
    - UTILITY -> EVAL
  + Most straightforward approach to controlling amount of search is setting fixed depth *d*
    - Need current depth incremented on each recursive call
    - IS-CUTOFF =true for all depth greater than *d* (as well as for all terminal states)
    - *d* is chosen so that move is selected within allocated time
    - Iterative deepening would be more robust
      * When time runs out program returns move selected by deepest completed search
      * Can cache entries in transposition table
      * Can use evaluations to improve ordering
  + **Evaluation function should only be applied to quiescent positions**
    - Quiescent position = positions where there are no pending moves that would wildly swing the evaluation
    - **Quiescent search**
      * For nonquiescent positions IS-CUTOFF = false and search continues until quiescent position is reached
      * Sometimes restricted to consider only certain types of moves that quickly resolve uncertainties in the position
  + **Horizon effect** arises when program is facing opponent’s unavoidable move that causes serious damage but can be temporarily avoided using delaying tactics
    - While tactics seem like good moves right now, they just delay the inevitable
    - How can we mitigate horizon effect?
      * Allow **singular extensions**: moves that are clearly better than all other moves, even when search would normally be cut off by that point
* **Forward pruning** - pruning moves that appear to be bad but might be good
  + Strategy saves time at the risk of an error (Type B strategy)
  + Forward pruning with **beam search**
    - Consider only a beam of best *n* moves rather than all possible moves
      * Best move could be pruned away
  + **Probabilistic cut**: forward pruning alpha beta search that uses statistics gained from prior experience to lessen the chance that best move will be pruned
    - Prunes nodes that are *probably* outside current alpha beta window
    - Computes probability by
      * shallow search to compute the backed up value *v*
      * Use past experience to estimate how likely that a score of *v* at depth *d* in the tree would be outside alpha beta window.
  + **Late move reduction**: assumes move ordering has been done well so moves that appear later in the list of moves are less likely to be good moves
    - Reduce the depth to which later moves are searched
    - If reduced search comes back with a value above current alpha, can rerun search with full depth
* **Search vs. lookup -** Use table lookup rather than search for opening and ending of games
  + For openings computer usually relies on human expertise
  + Computers can gather statistics from database of played games to see which opening sequences most often lead to win
  + At the start or end of the game there are fewer possible positions, so lookup is easier
  + Endgames can be tricky
    - Computers can produce a **policy** to solve the end game
      * Policy: mapping ever possible state to the best move in that state
      * Then computer plays perfectly using lookup
    - Table is constructed by retrograde minimax search
* **Monte Carlo Tree Search (MCTS)** - instead of heuristic eval function, value of a state is estimated as average utility over a number of simulations of complete games starting from the state
  + Simulation (aka playout, rollout) chooses moves for each player until terminal position is reached
  + Rules of the game determine who won and by what score
    - For games with binary outcome average utility = win percentage
  + How to choose what moves get made during simulation?
    - Need a **playout policy** that biases moves toward good ones
  + From what positions do we start playouts? How many playouts do we allocate each position?
    - **Pure Monte Carlo search**: do *N* simulations from currente state and track which possible moves from current position has highest win %
    - Need a **selection policy** that selectively focuses computational resources on important parts of game tree. Needs to balance
      * **exploration** of states that had few playouts
      * **exploitation** of states that have done well in past playouts
  + Four steps of maintaining a search tree and growing it each iteration
    - **Selection**: starting at root of tree, choose a move guided by selection policy leading to a successor node. Repeat process moving down the tree to a leaf
    - **Expansion**: grow search tree by generating new child of selected node
    - **Simulation**: perform playout from newly generated child node, choosing moves for both players according to playout policy
      * Moves aren’t recorded in search tree
    - **Back-Propagation**: Now use result of simulation to update all search tree nodes going up to root
  + Repeat steps for set number of iterations or until allotted time expired, then return move with highest number of playouts



* + **Upper confidence bounds applied to trees (UCT)** - effective selection policy
    - Ranks each possible move based on upper confidence bound formula **UCB1**
      * **U(*n*)** - total utility of all playouts through node *n*
      * **N(*n*)** - number of playouts through *n*
      * **PARENT(*n*) -** parent node of *n* in tree
      * U(*n*) / N(*n*) is exploitation term
        + average utility of *n*
      * Term under square root is exploration term
        + *N(n)* is denominator which means term will be high for nodes that have only been explored a few times
        + Numerator has the log of number of times we’ve explored the parent of *n*
        + **If we are selecting *n* some non-zero percentage of time, exploration term goes to zero as the counts increase and eventually playouts are given to node with highest average utility**
      * *C* - constant that balances exploitation and exploration
        + Theoretically should be sqrt(2)
        + Usually multiple values are tried, and best one is chosen
    - Formula ensures that node with most playouts is almost always the node with highest win % because selection process favors win % more as number of playouts increase
  + **Time to compute a playout is linear not exponential**
    - Only one move is taken at each choice point
  + MCTS has advantage over alpha beta (AB) games where branching factor is very high or when it’s difficult to define good eval function
    - AB chooses path to a node that has highest achievable eval func score given that opponent trying to minimize score
    - If eval function is inaccurate, AB is inaccurate
    - Miscalculation can lead AB to choose/avoid a path to the best node
  + MCTS relies on aggregate of many playouts, not vulnerable to a single error like alpha beta search.
  + Can combine MCTS and eval functions
    - Do a playout for certain # of moves then truncate playout and apply eval function
  + Can Combine AB and MCts
    - Can use early playout termination
      * Stop a playout that’s taking too many moves
      * Evaluate it with heuristic function or just declare draw
  + MCTS can be applied to games where there is no experience to draw on eval function
    - All we need is rules of game
    - Selection and playout policies can be helped by expert knowledge but good policies can be learned using neural networks trained by self play
  + **MCTS has disadvantage when it’s likely that single move can change course of the game**
    - Type B pruning in MCTS means vital line of play may not be explored
  + **MCTS has disadvantage when some states are obvious wins for a side but where it’ll take many moves in playout to verify winner**
* **Stochastic games** - games that include a random element, combining luck and skill
  + Game tree in stochastic games need **chance nodes** in addition to max min nodes
    - E.g. Dice roll events
  + How can we pick the move that leads to the best position?
    - Positions don’t have definite minimax values
    - Calculate **expected value:** sum of the value over all outcomes weighted by probability of each chance action
  + **Expectiminimax** **value**: generalization of minimax (from deterministic games) for games with chance nodes



* **Evaluation functions for stochastic games must return values that are a positive linear transformation of the probability of winning**
  + Think of a game where a throw of dice precedes each move
    - There are no likely sequences of moves
    - Even the most likely move occurs only ½ of the time because for the move to take place the dice would first have to come out the right way to make it legal
  + Alpha-beta (AB) pruning can be applied to game trees with chance nodes
    - Can prune minimax and chance nodes
    - Put bounds on possible values of the utility function
      * Can arrive at bounds for the average without looking at every child of chance node
    - For games with high branching factor for chance nodes
      * Consider forward pruning that samples smaller number of possible chance branches
      * May want to opt for MCTS instead where playout includes random rolls
* **Partially observable games** are games with ‘fog of war’ e.g. don’t know where enemy is until revealed by contact
  + In deterministic partially observable games, uncertainty about state of the board arises entirely from lack of access to the choices made by opponent
    - E.g. battleship
    - E.g. Kriegspiel (partially observable variant of chess where pieces are invisible to opponent)
  + **Belief state** - set of all logically possible board states given the complete history of percepts to date
  + Keeping track of belief state as game progresses is the problem of state estimation in partially observable games
  + Instead of specifying a move to make for each possible move the opponent might make need a move for every possible percept sequence that might be received
  + **Guaranteed checkmate**: for each possible percept sequence leads to an actual checkmate for every possible board state in the current belief state regardless of how opponent moves
    - Opponent’s belief state is irrelevant, strategy has to work even if opponent can see all the pieces
  + AND-OR algorithm can be applied to belief state space to find guaranteed checkmates
  + **Probabilistic checkmate**: required to work in every board state in the belief state
    - Probabilistic with respect to randomization of the winning player’s moves
  + **Accidental checkmate**: strategy works for some of the board states in current belief state but not others
    - Can’t know it would be checkmate
    - How likely is a given strategy to win?
      * How likely is that each board state in the current belief state is the true board state?
    - Each player’s goal isn’t to just to move correctly, but also to minimize info that the opponent has about their location
  + **Optimal play in partially observable games requires a willingness to play somewhat randomly**
    - Playing predictable optimal strategy provides opponent with information
    - Occasionally selecting moves that may seem weak but gain strength from their very unpredictability because opponent is likely unprepared for it
    - Probabilities associated with board states in the current belief state can only be calculated given an optimal randomized strategy
      * Computing a strategy seems to require knowing probabilities of various states the board might be in
      * **Equilibrium** specifies optimal randomized strategy for each player
        + Can be really expensive
    - Most systems use bounded-depth look-ahead in their own belief state space ignoring opponent’s belief state
  + Card games feature **stochastic partial observability** - missing info is generated by random dealing of cards
    - Assumption: treat the start of the game as chance node with every possible deal as an outcome and then use EXPECTIMINIMAX to pick best move
      * Only root node is chance node, then game is fully observable to both players
      * “Averaging over clairvoyance”
      * Fails because it doesn't consider the belief state that the agent will be in after acting
        + Assumes every future state will automatically be one of perfect knowledge
        + Clairvoyant approach never selects actions that gather info
        + Won’t choose actions that hide info from the opponent

Because it assumes they know already

* + - Number of possible hands that can be dealt is too huge, use **abstraction** to treat similar hands as identical
    - Forward pruning: consider only a small random sample of *N* deals and calculate EXPECTIMINIMAX score
      * Can also do heuristic search with depth cutoff rather than search entire game tree
    - Consider: Is each deal equally likely for all card games?
* **Limitations of Game Search Algorithms**
  + Calculating optimal decisions in complex games is intractable. All algorithms must make some assumptions and approximations
    - AB search uses heuristic eval function to approximate
    - MCTS computes average over a random selection of playouts
      * When branching factor is high or it’s difficult to define evaluation function MCTS is preferred
  + Limitations of AB Search
    - Vulnerability to errors in heuristic function
  + Limitation of AB and MCTS
    - Designed to calculate bounds on the values of legal moves
      * Sometimes there’s one move that is obviously best (e.g. there’s only one legal move)
      * There’s no point wasting computation time to figure out value of move
      * Better implementation would use idea of **Utility of a node expansion** - selecting node expansions of high utility
        + Ones that are likely to lead to the discovery of significantly better move
        + If there are no expansions whose utility is higher than their cost the algorithm should stop searching and make a move
        + **Metareasoning** - reasoning about reasoning
  + Limitation of AB and MCTS
    - Do all their reasoning at the level of individual moves
      * Humans can reason at a higher-level goal (e.g. trap opponent’s queen in chess)
      * Use the higher level goal to selectively generate plausible plans
  + Good strategies should incorporate machine learning into the game search process