AI基础

Lecture 5：Adversarial Search and Games

Bin Yang
School of Data Science and Engineering
byang@dase.ecnu.edu.cn

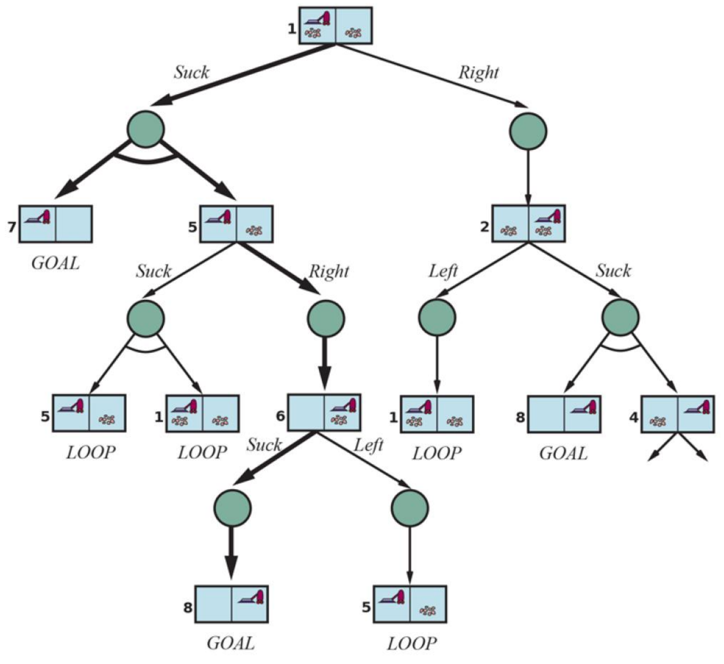
[Some slides adapted from Roberto Sebastiani, Trento; Marco Chiarandini, SDU]

Lecture 4 ILOs

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x}),$$

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x}) \nabla f(\mathbf{x}),$$

- Search in complex environments
  - Local search in continuous spaces
    - Steepest ascent hill climbing, Newton method
  - Search with nondeterministic actions
    - AND-OR search, conditional plan
  - Search in partially observable environments
    - Sensorless problems
      - Belief-state space, ordinary search, sequences of actions in belief-state
    - Partially observable environments
      - AND-OR search, conditional plan
  - Online search agents and unknown environments



The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

- A solution for an AND-OR search problem is a subtree of the complete search tree that
  - Has a goal node at every leaf
  - Specifies one action at each of its OR nodes
  - Includes every outcome branch at each of its AND nodes

Lecture 5 ILOs

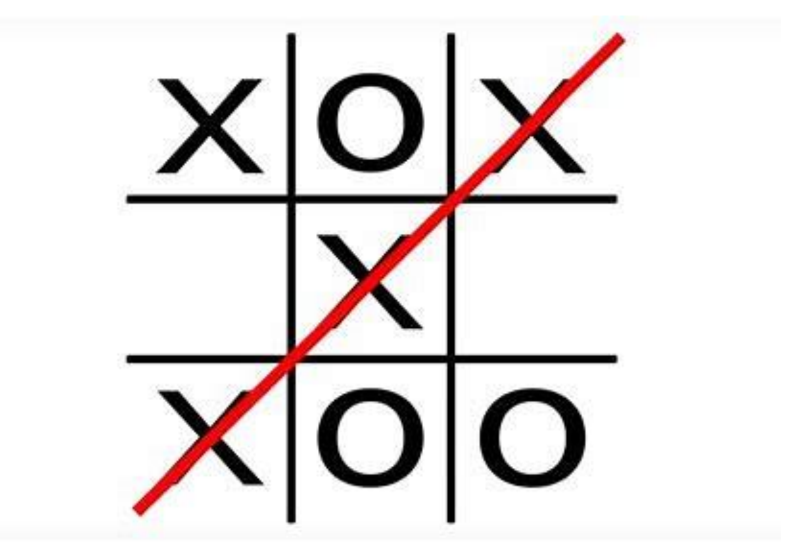
- Adversarial Search and Games
    - Game theory, problem settings
    - Optimal Decisions in games, minimax search
    - Alph-Beta Pruning, Heuristic Alph-Beta Tree Search
    - Stochastic games
    - Monte Carlo Tree Search

Problem settings

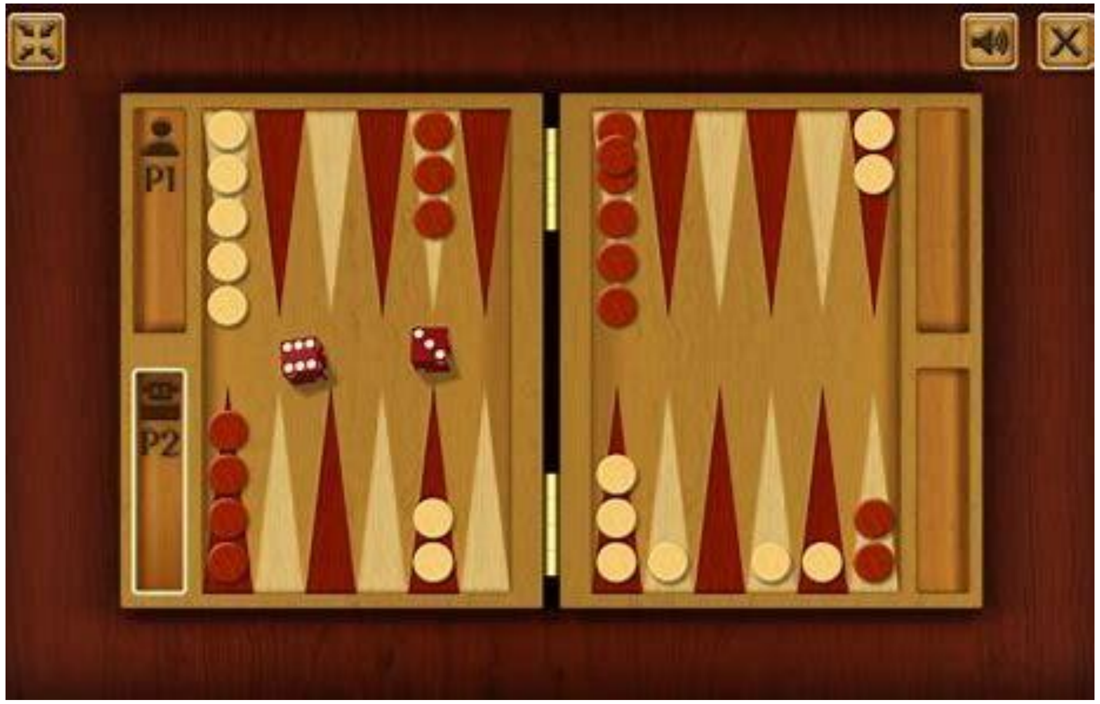
- Competitive environments
  - Two or more agents have conflicting goals
  - Giving rise to adversarial search
- Multi-agent environments, 3 categories
  - Economy: when there are a very large number of agents, consider them in the aggregate as an economy.
    - Allowing us to do things like predict that increasing demand will cause prices to rise
    - Often do not predict the action of any individual agent.
  - Consider adversarial agents as just a part of the environment—a part that makes the environment nondeterministic.
    - But if we model the adversaries in the same way that, say, rain sometimes falls and sometimes doesn’t, we miss the idea that our adversaries are actively trying to defeat us, whereas the rain supposedly has no such intention.
  - Explicitly model the adversarial agents with the techniques of adversarial game-tree search.

Adversarial search vs. search

- Search (with no adversary)
  - Solution is a sequence of actions or a conditional plan
  - Admissible heuristics techniques can help find optimal solutions
  - Evaluation function: estimate of cost from start to goal through given node
  - Examples: path planning, scheduling activities, …
- Games (with adversary), a.k.a., adversarial search
  - Solution a is strategy (specifies move for every possible opponent reply)
  - Evaluation function (utility): evaluate “goodness” of game position
  - Examples: tic-tac-toe, chess, checkers, Othello, backgammon, …
  - Often time limits force an approximate solution


X O X
X
X O O

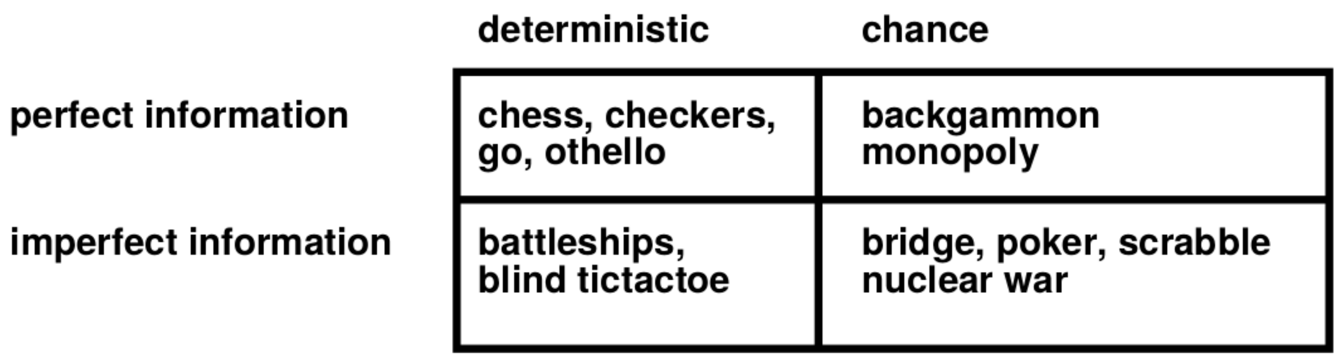
A B C D E F G H
1 2 3 4 5 6 7 8

GoodEasy

P1
P2

Outline

- Two-player zero-sum games
- Optimal Decisions in games, minimax search
  - Two players
  - Multiple players
- Alpha-Beta Pruning
- Heuristic Alpha-Beta Tree Search
- Stochastic games
- Monte Carlo Tree Search

Many different kinds



	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

- Relevant features:
  - deterministic vs. stochastic (with chance)
  - one, two, or more players
  - zero-sum vs. general games
    - Zero-sum means that what is good for one player is just as bad for the other
    - Zero-sum games have no “win-win” outcomes
  - perfect information (can you see the state?) vs. imperfect
    - Fully vs. partially observable
- Most common: deterministic, turn-taking, two-player, zero-sum games, of perfect information

- Want algorithms for calculating a strategy (a.k.a. policy):
  - recommends a move from each state: policy : S->A

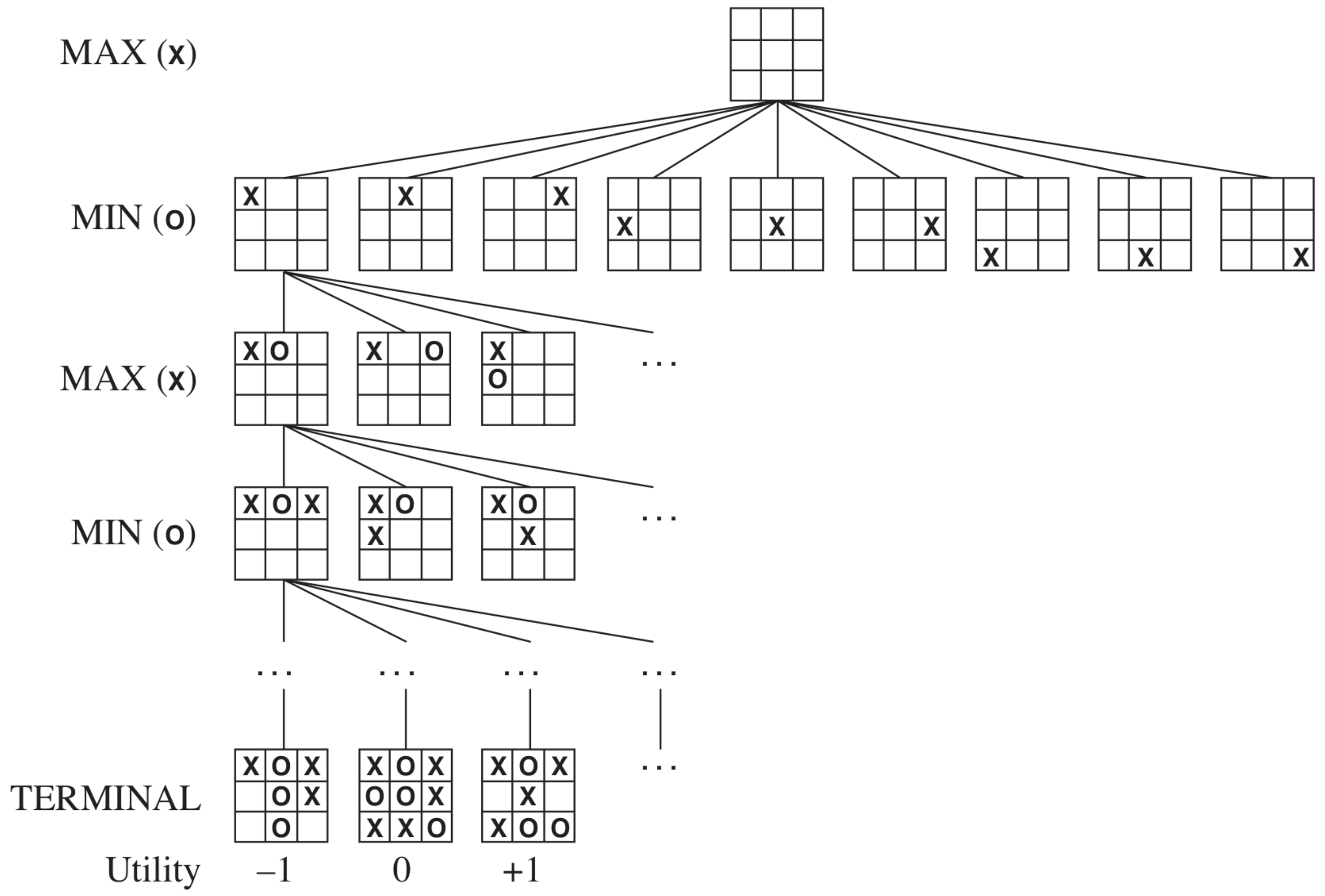
Two-player zero-sum games

- Two players: MAX and MIN
    - MAX moves first
    - The players take turns moving until the game is over.
    - At the end of the game, points are awarded to the winning player and penalties are given to the loser.
    - Move: Action
    - Position: State

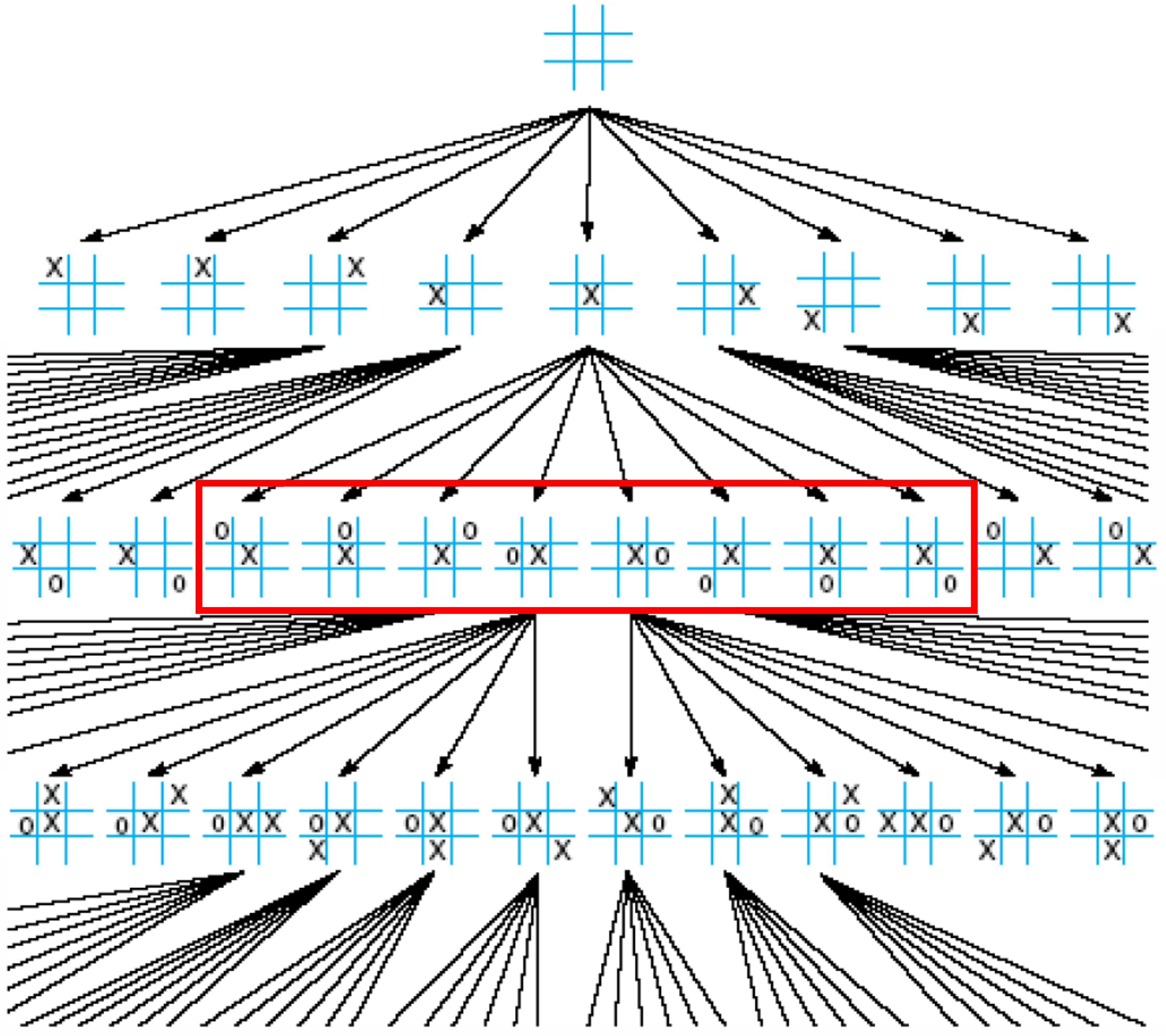
Games and game trees

- A game can be formally defined with the following elements
  - S_0: initial state, specifies how the game is set up at the start
  - To-Move(s): the player whose turn it is to move in state s
  - Actions(s): returns the set of legal moves in state s
  - Result(s; a): the transition model, defines the result of taking action a at state s
  - Is-Terminal(s): true iff the game is over (if so, s is a terminal state)
  - Utility(s, p): A utility function (a.k.a. objective function or payoff function) defines the **final numeric value** for a game ending in **terminal** state s for **player** p
    - Example: chess: win 1, loss 0, draw ½
  - S_0, Actions(s) and Result(s, a) recursively define the **game tree**
    - Nodes are states, edges are actions

Example game tree, tic-tac-toe, 2-player, turn-taking, deterministic



Mini-quiz: how many terminal states in the game tree?



8 possible positions to draw a circle

$9! = 362\ 880$

Zero-sum games vs general games

- General Games
  - agents have independent utilities
  - cooperation, indifference, competition, and more are all possible
- Zero-Sum Games: the total payoff to all players is the **same** for each game instance
  - adversarial, pure competition
  - agents have opposite utilities (values on outcomes)
  - Example: chess: win 1, loss 0, draw ½
- Idea: With two-player zero-sum games, we can use one single utility value
  - one agent maximizes it, and the other minimizes it
  - optimal adversarial search as minimax search

Outline

- Two-player zero-sum games
- Optimal Decisions in games, minimax search
    - Two players
    - Multiple players
- Alpha-Beta Pruning
- Heuristic Alpha-Beta Tree Search
- Stochastic games
- Monte Carlo Tree Search

Optimal decisions in games

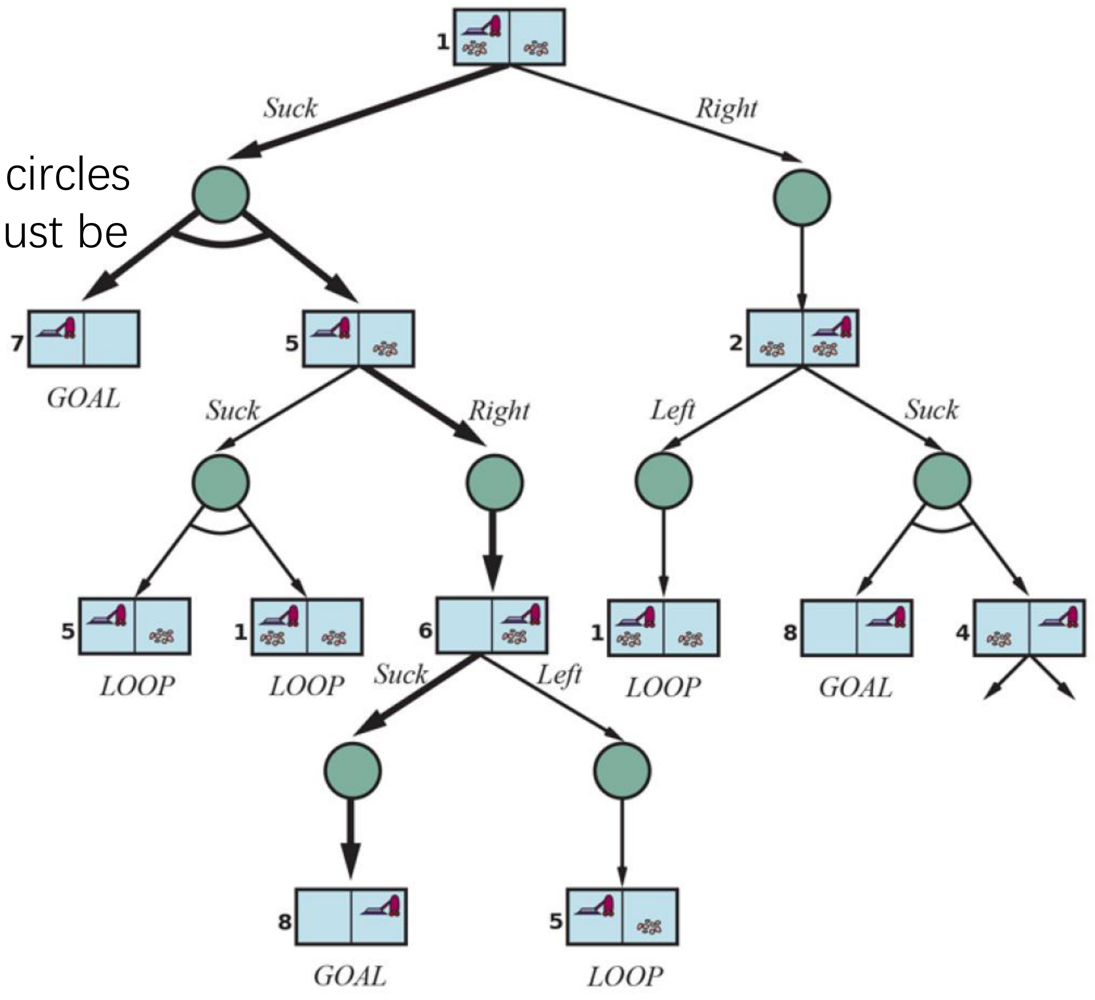
- MAX wants to find a sequence of actions leading to a win.
- MAX’s strategy must be a conditional plan
  - Specifying a response to each of MIN’s possible moves.
- In games that have a binary outcome (win or lose), we could use AND-OR search to generate the conditional plan.
  - MAX playing the role of OR
  - MIN playing the role of AND
- For games with multiple outcome scores, we need a slightly more general algorithm called **minimax search**

AND-OR search trees

- OR nodes
  - The nodes that we have seen in search trees in deterministic environments.
  - Branching is introduced by the agent's own choices in each state.
- AND nodes
  - Branching is also introduced by the environment's choice of outcome for each action
- Two kinds of nodes alternate, leading to an AND-OR tree

MAX: OR Node, rectangles
Some action must be chosen

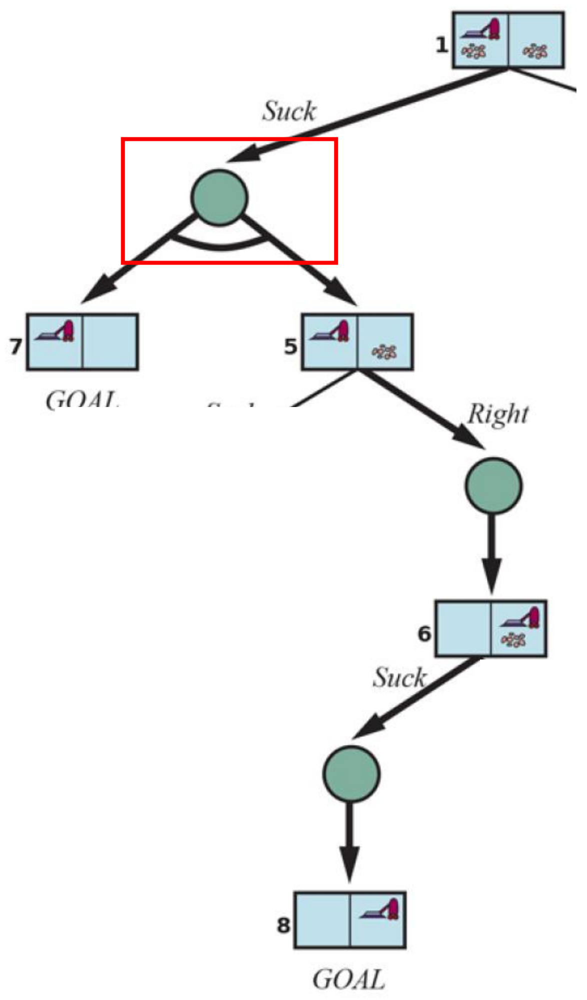
MIN: AND Node, circles
Every outcome must be handled



The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

AND-OR search trees

- A solution for an AND-OR search problem is a subtree of the complete search tree that
  - Has a goal node at every leaf
  - Specifies one action at each of its OR nodes
  - Includes every outcome branch at each of its AND nodes



The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

Minimax search

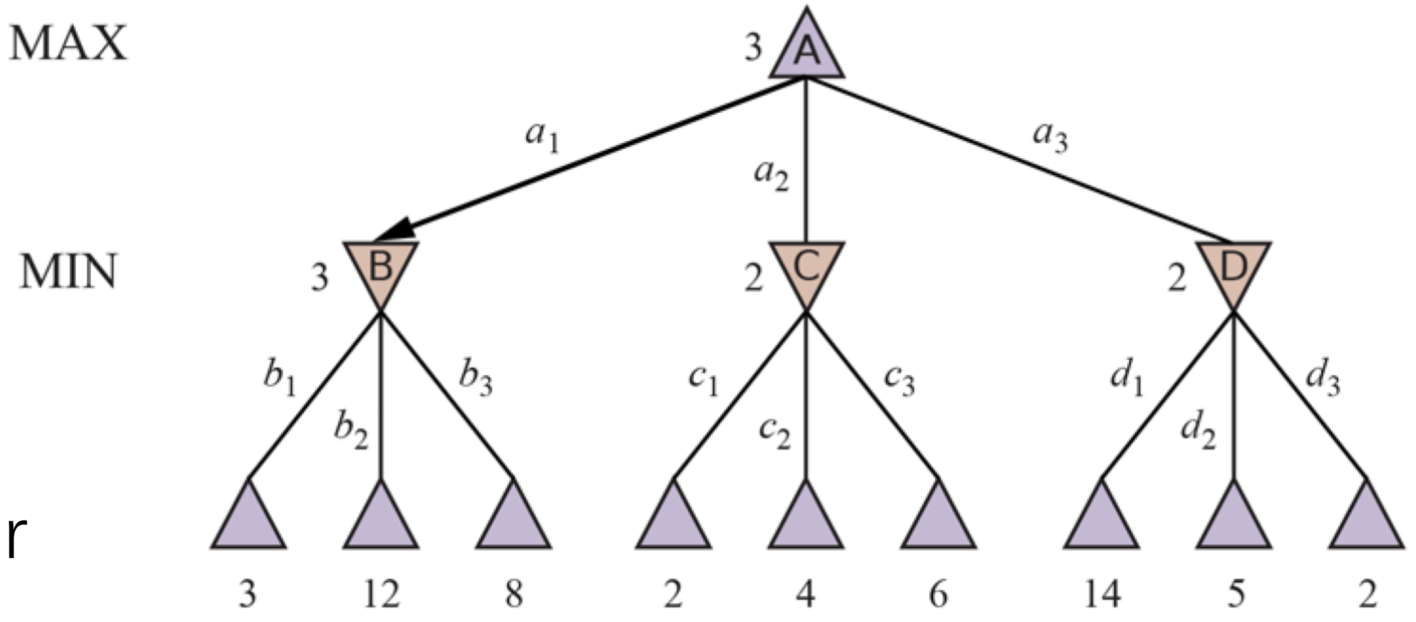
- Optimal strategy is determined by working out the minimax value of each state in the tree, denoted by minimax(s)

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s, \textsc{max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s, a)) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s, a)) & \text{if } \textsc{To-Move}(s) = \textsc{min} \end{cases}$$

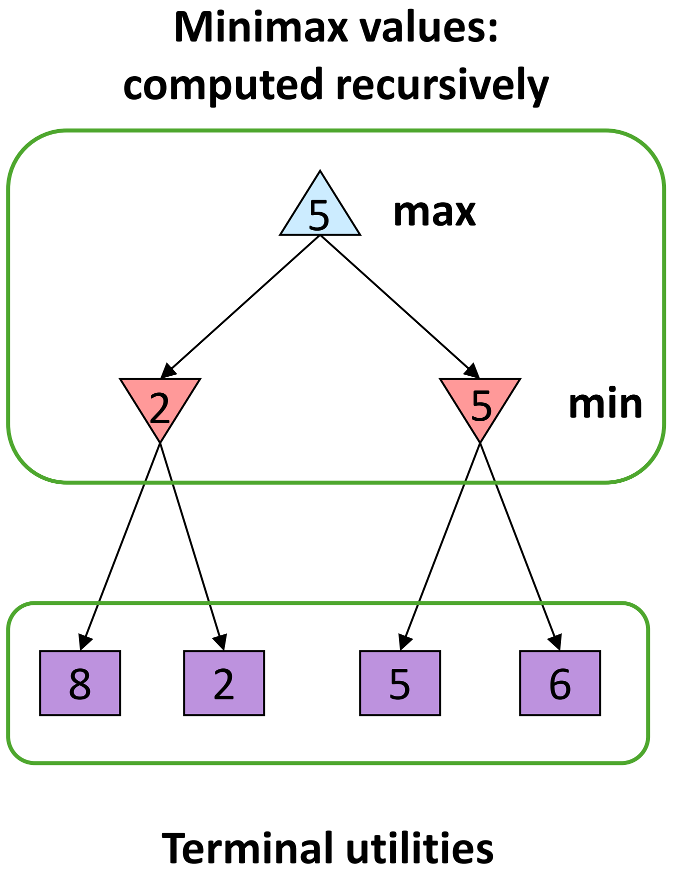
- Minimax value for a terminal state is just its utility.
- In a non-terminal state
  - when it is MAX’s turn to move, MAX prefers to move to a state of maximum value
  - when it is MIN’s turn to move, MIN prefers a state of minimum value (that is, minimum value for MAX and thus maximum value for MIN).

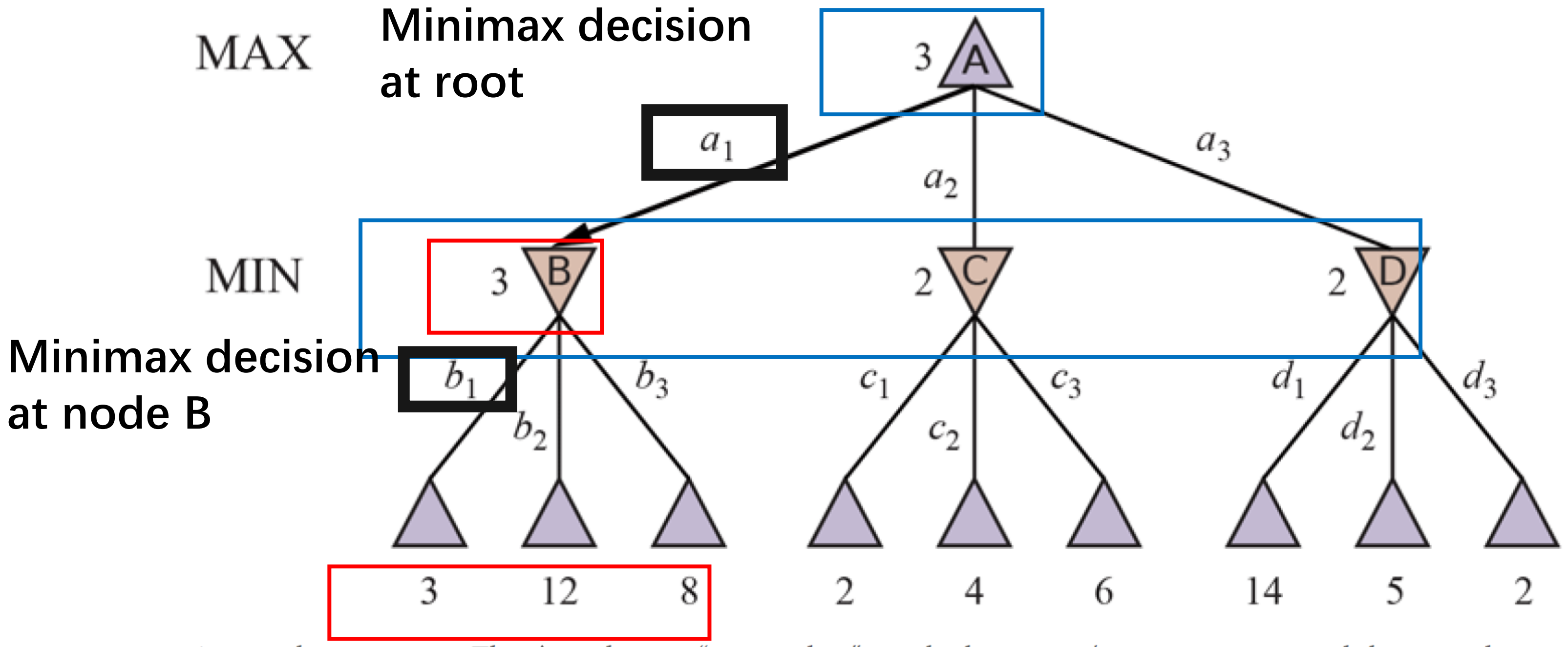
Minimax Example

- MAX has 3 possible moves
  - a_1, a_2, a_3
- The possible replies to a_1 for MIN are
  - b_1, b_2, b_3
- The game ends after one move each by MAX and MIN.
- Move: both players play
- ply: one move by one player



A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.





A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s,\textsc{max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_{a\in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_{a\in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{min} \end{cases}$$

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

The minimax search algorith

- Depth-first exploration of the game tree
- Completeness?
  - Yes, if tree is finite
- Time complexity?
  - $O(b^m)$
- Space complexity?
  - $O(bm)$
- Chess has a branching factor of 35, and the average game has depth of about 80 ply
  - $35^{80}=10^{123}$ states

```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

MAX goes to a state with maximum

MIN goes to a state with minimum

Whether or not MIN plays optimally

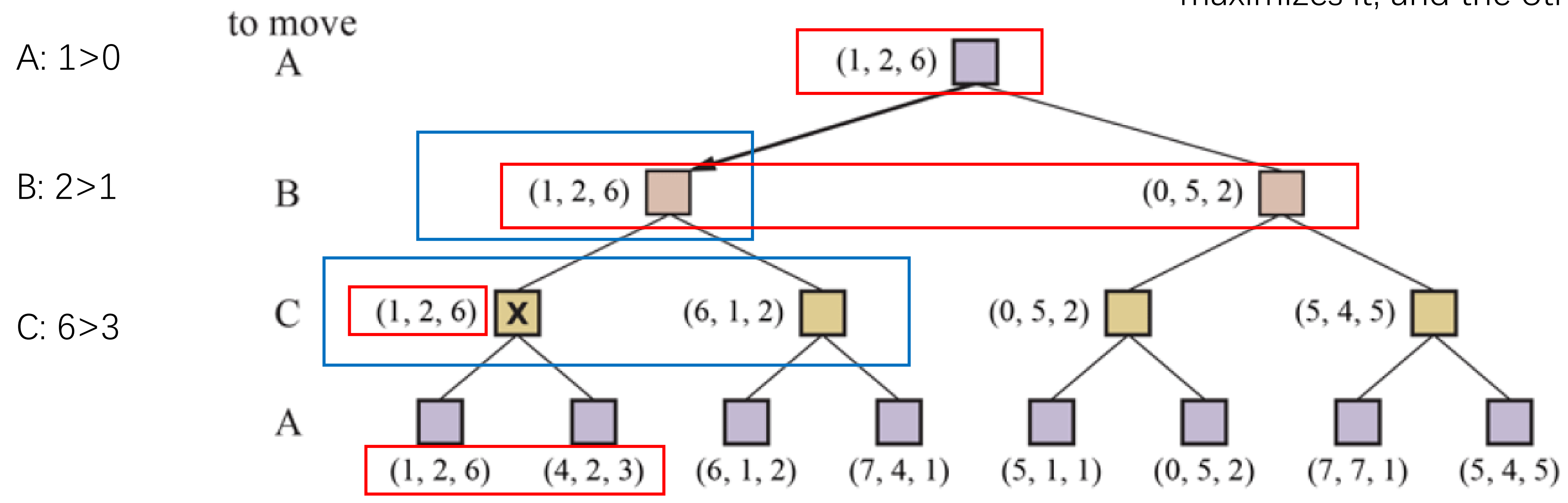
- This definition of optimal play for MAX assumes that MIN also plays optimally.
- What if MIN does not play optimally?
    - Then MAX will do at least as well as against an optimal player, possibly better.
- However, that does not mean that it is always best to play the minimax optimal move when facing a suboptimal opponent.
    - Consider a situation where optimal play by both sides will lead to a draw.
    - But there is one risky move for MAX that leads to a state in which there are 10 possible response moves by MIN that all seem reasonable, but 9 of them are a loss for MIN and one is a loss for MAX.
    - If MAX believes that MIN does not have sufficient computational power to discover the optimal move, MAX might want to try the risky move, on the grounds that a 9/10 chance of a win is better than a certain draw.

Optimal decisions in multiplayer games

- Three players A, B, and C
  - Three players move in turns, choose the action with highest value for themselves
- Vector <v_A, v_B, v_C> is associated with each node
- For terminal states
  - The vector gives the utility of the terminal state from each player's viewpoint
- For non-terminal states
  - A player chooses the move with the highest value from the player's viewpoint

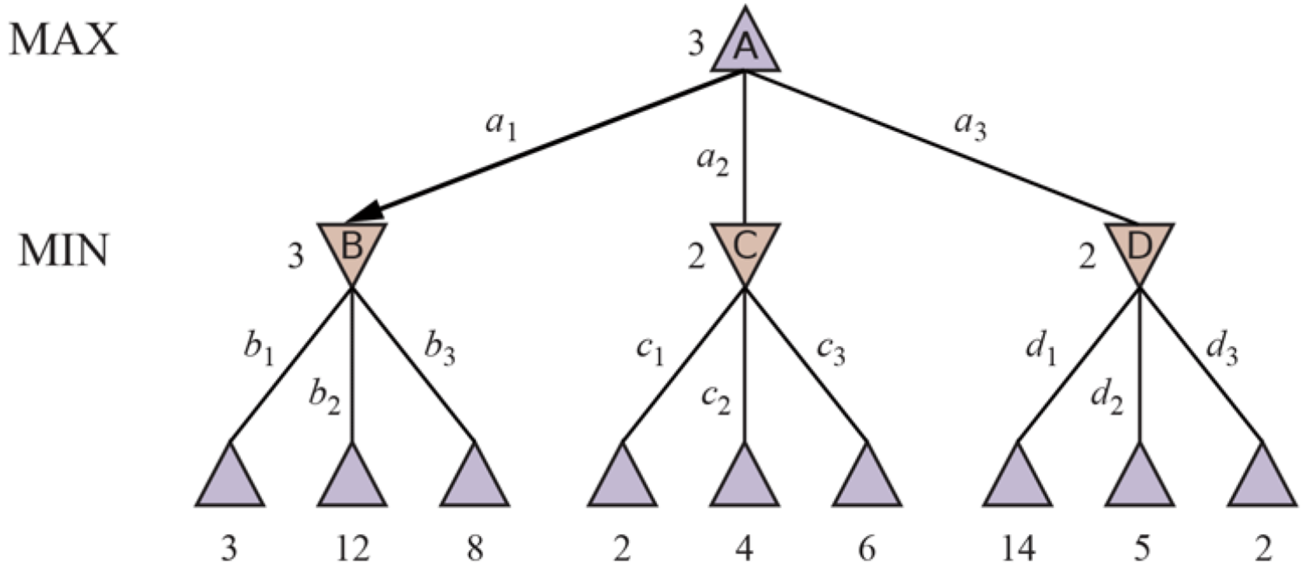
Optimal decisions in multiplayer games

With two-player zero-sum games, we can use one single utility value: one agent maximizes it, and the other minimizes it.



The first three ply of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Miniquiz: can you transform this two-player game tree to a game tree in the multiplayer games (a vector of two values with two players A and B)?



A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Multiplayer games

- Alliances
- For example, suppose A and B are in weak positions and C is in a stronger position. Then it is often optimal for both A and B to attack C rather than each other, lest C destroy each of them individually.
- In this way, collaboration emerges from purely selfish behavior. Of course, as soon as C weakens under the joint onslaught, the alliance loses its value, and either A or B could violate the agreement.
- When the game is not zero-sum, then collaboration can also occur with just two players. Suppose, for example, that there is a terminal state with utilities <1000, 1000> and that 1000 is the highest possible utility for each player. Then the optimal strategy is for both players to do everything possible to reach this state—that is, the players will automatically cooperate to achieve a mutually desirable goal.
  - Zero-sum games have no “win-win” outcomes

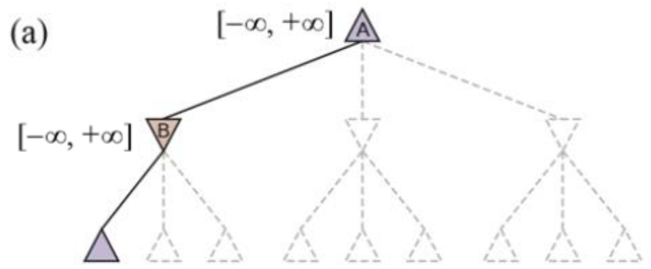
Outline

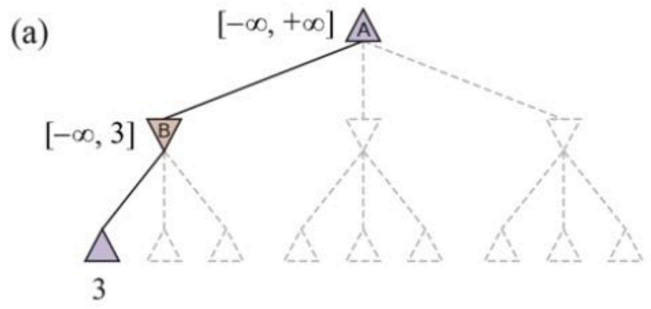
- Two-player zero-sum games
- Optimal Decisions in games, minimax search
    - Two players
    - Multiple players
- Alpha-Beta Pruning
- Heuristic Alpha-Beta Tree Search
- Stochastic games
- Monte Carlo Tree Search

Alpha-beta pruning

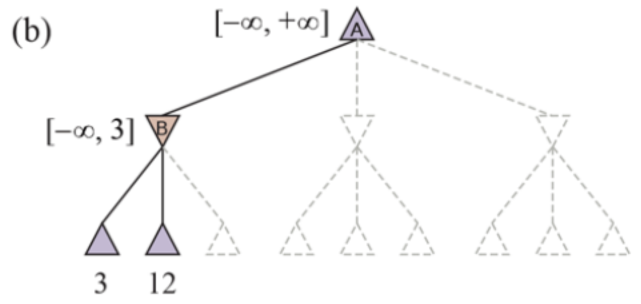
- Recall that minimax search has a time complexity of O(b^m)
- Chess has a branching factor of 35, and the average game has depth of about 80 ply
  - $35^{80}=10^{123}$ states
- Infeasible to solve use minimax search. Can we speed up?
- Pruning the tree
  - Without the need to examine every state

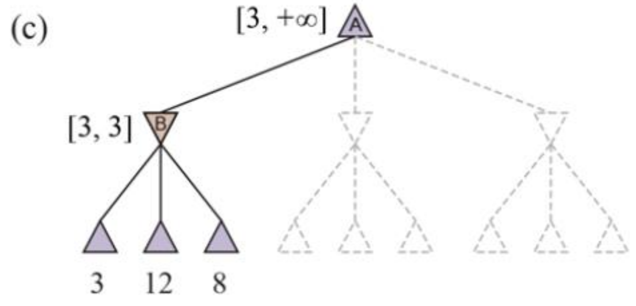
Pruning example



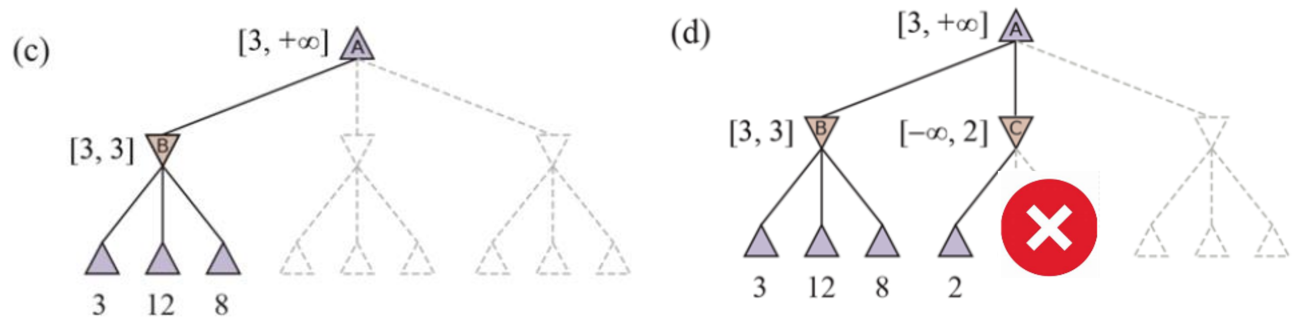


- Let [min, max] track the currently-known bounds for the search
- In the beginning, all with [-infinity, infinity]
- When visiting the first leaf with utility 3, we have [-infinity, 3] for B, at most 3
  - Since B is MIN playing, MIN only cares of reducing the upper bound
- We have examined all leaves of B, so we have [3, 3] for B.
- Now we also know for node A, at least we have 3, so [3, infinity]





Pruning example

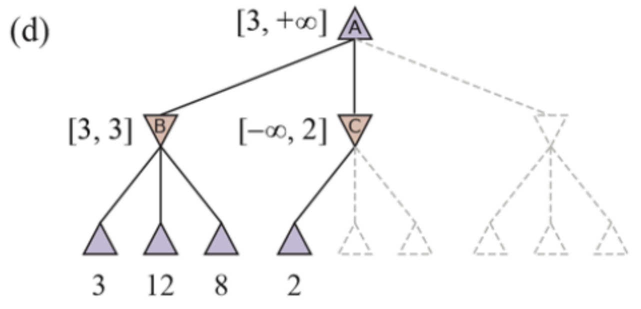


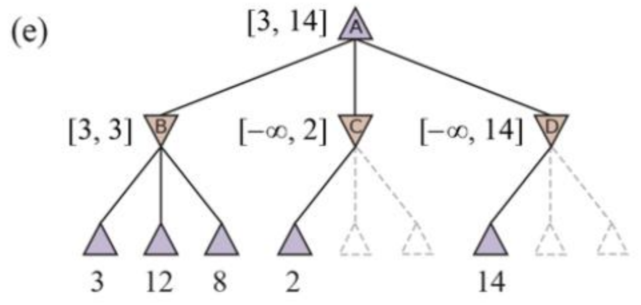
- Now we go to node C
- After visiting the first leaf of C, we have [-infinity, 2] for C, at most 2
- Is it needed to continue checking the other leaves under C?
  - NO! The upper bound is 2. Since MIN is playing, checking the remaining leaves cannot increase the current upper bound 2, whereas B has an upper bound 3 which is larger than 2.
  - No matter what the remaining leaves of C will give, A will not choose to go to C given that B has a larger upper bound 3.
  - So pruning the remaining leaves under C.

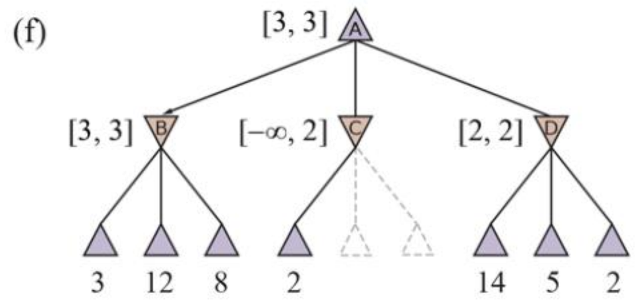
Pruning example

- We go to the first leaf of D
  - [-infinity, 14] for D, and [3, 14] for A
- Visit the second leaf of D
  - Unchanged
- Visit the third leaf of D
  - [2, 2] for D, and [3, 3] for A

$$
\begin{aligned}
\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3, z, 2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\
&= 3.
\end{aligned}
$$

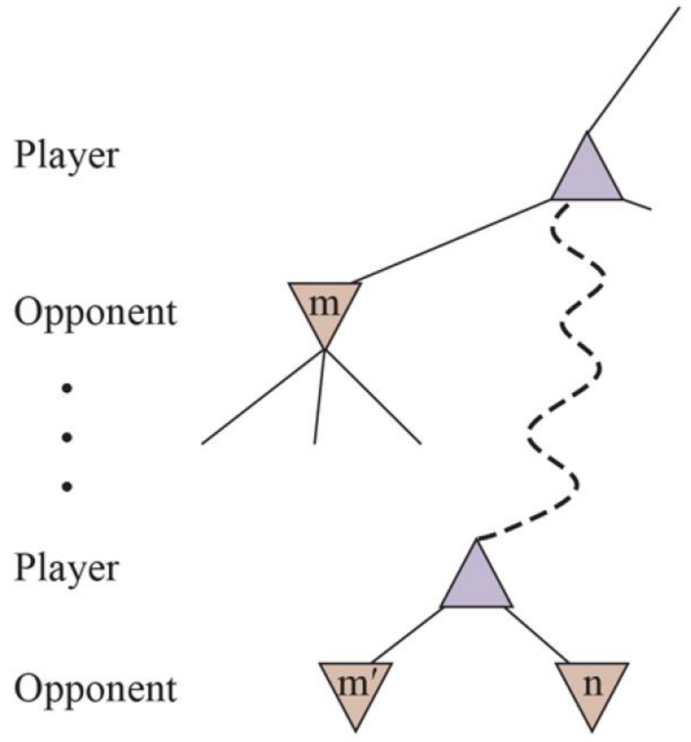






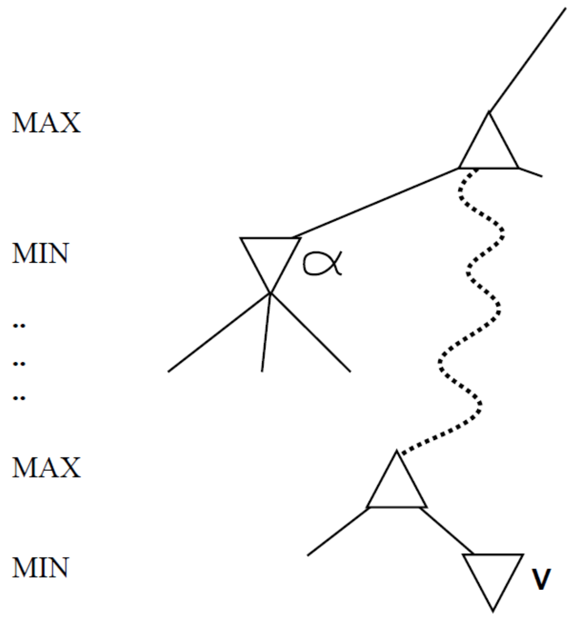
Alpha-beta pruning for Minimax search

- Consider a node n somewhere in the tree
  - If player has a better choice either at the same level, e.g., m', or at any point higher up in the tree, e.g., m, then the player will never move to n.
  - If we know enough of n to draw this conclusion, we can prune n



Alpha-beta pruning for Minimax search

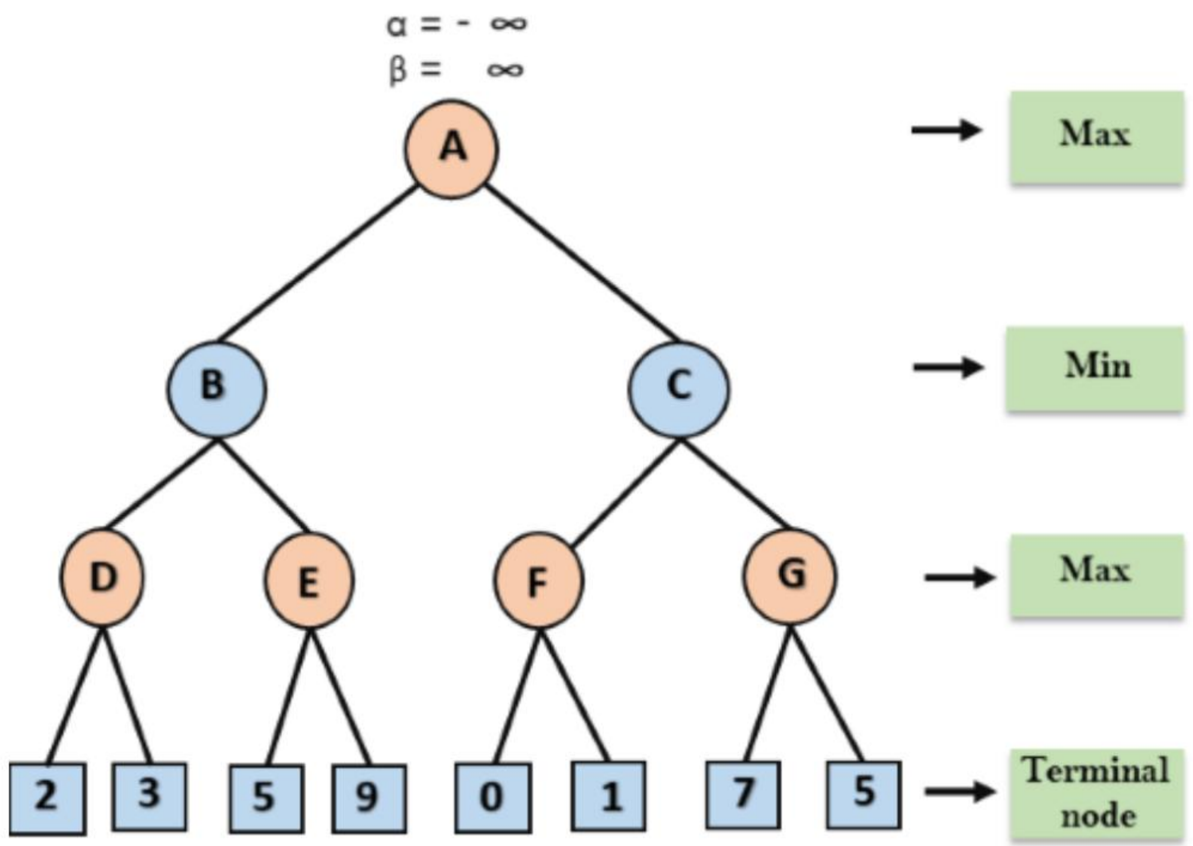
- Alpha: the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
  - Think: “at least.”
- Beta: the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.
  - Think: “at most.”



α is the best value (to MAX) found so far along the current path
If V is worse ($<$) than α, MAX will avoid it $\Rightarrow$ prune that branch
Define β similarly for MIN

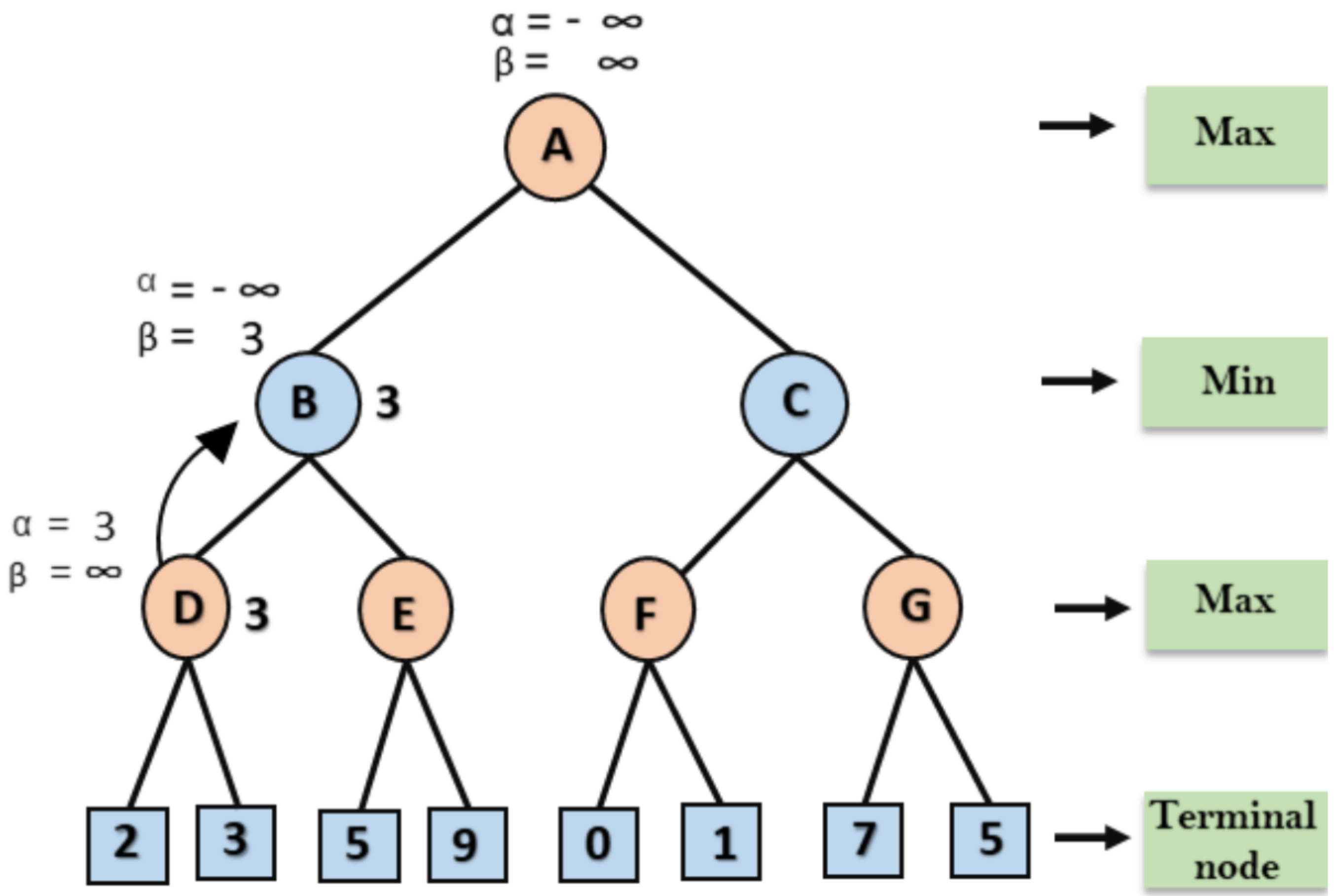
Step 1: At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.

Alpha, MAX, "at least"
Beta, MIN, "at most"



Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.

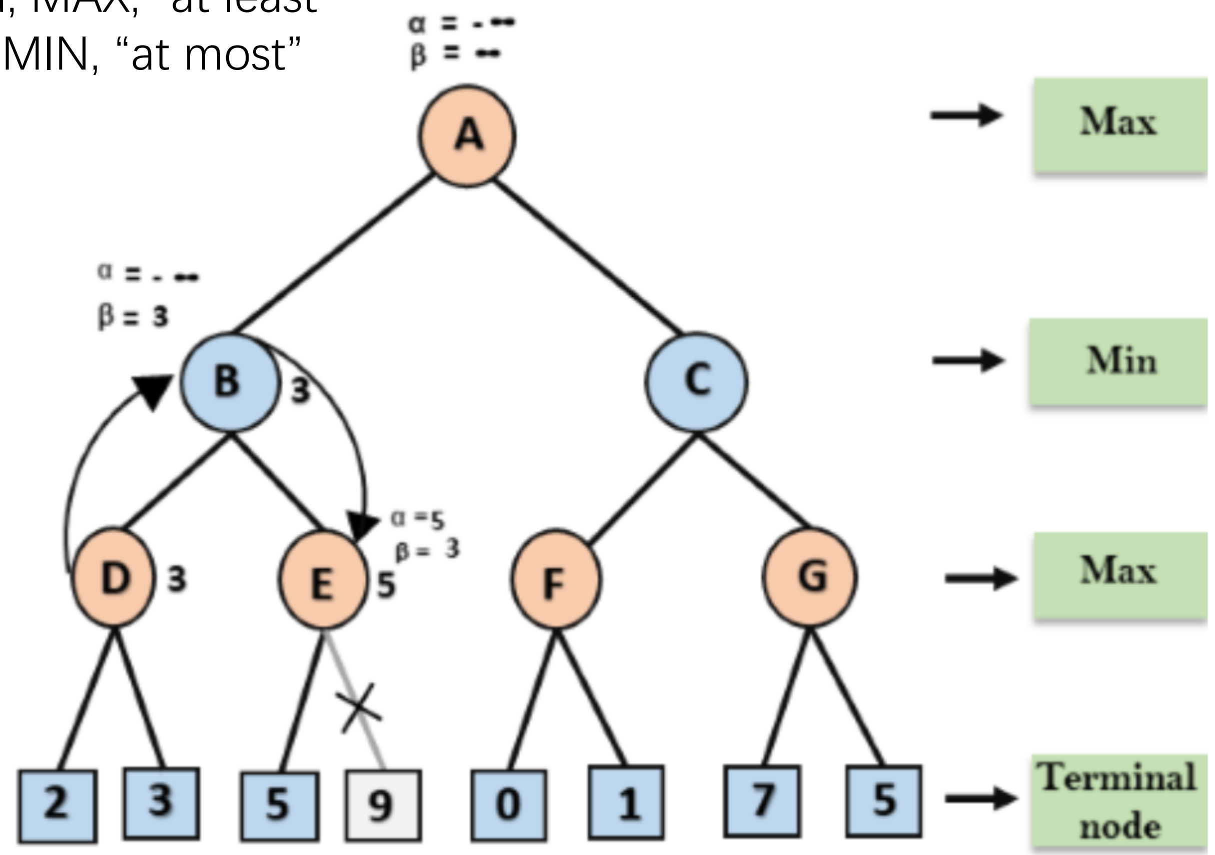


Alpha, MAX, "at least"
Beta, MIN, "at most"

In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.

Alpha, MAX, "at least"
Beta, MIN, "at most"

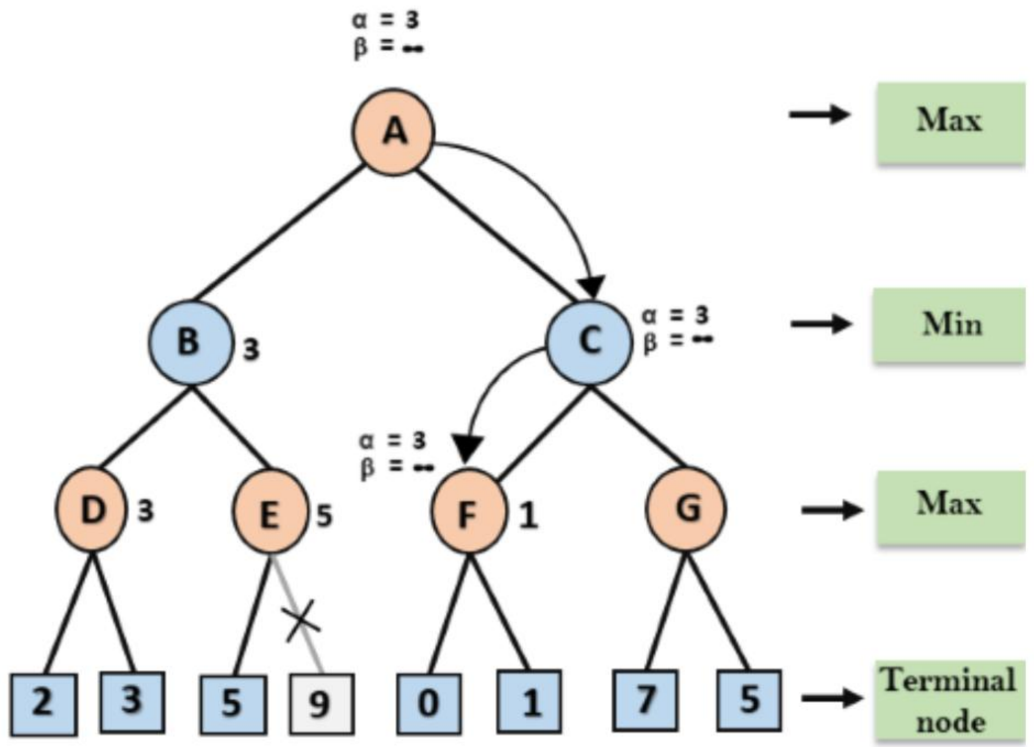


- At node E
  - MAX plays, alpha=5, it is at least 5.
- At node B
  - MIN plays, beta=3, it has at most 3.
- Alpha>=Beta, so E will not be chosen by MIN in node B.

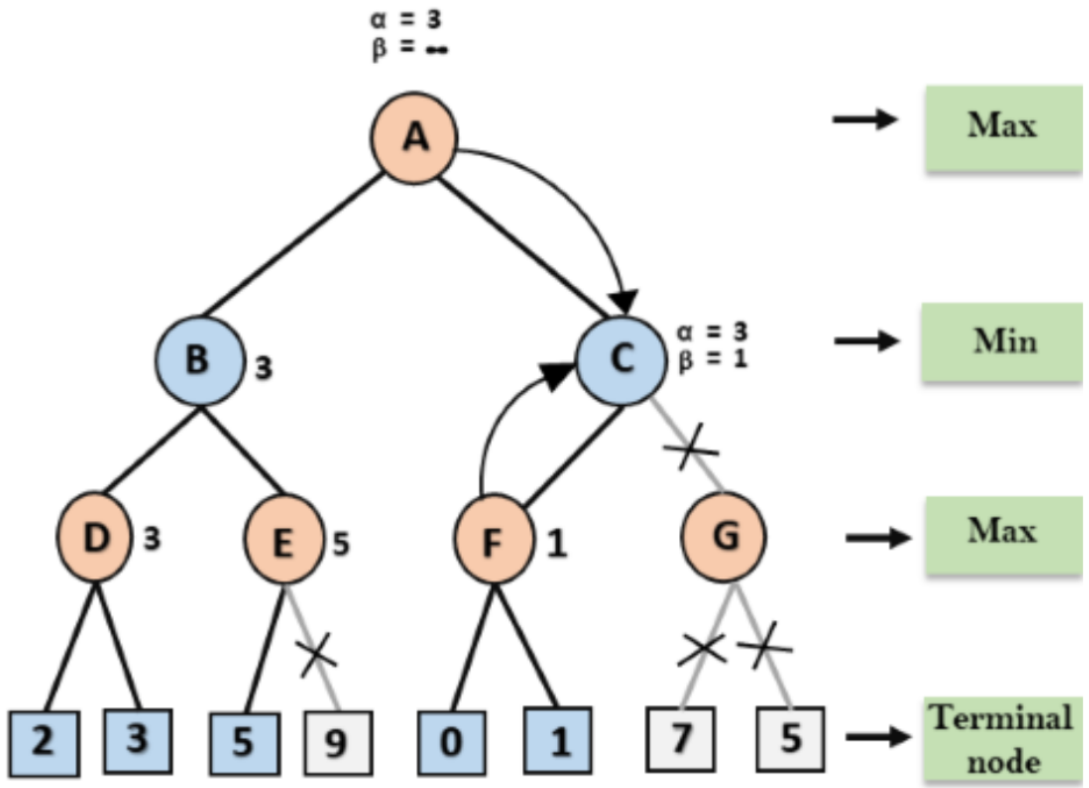
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

At node C, α=3 and β= +∞, and the same values will be passed on to node F.

Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.

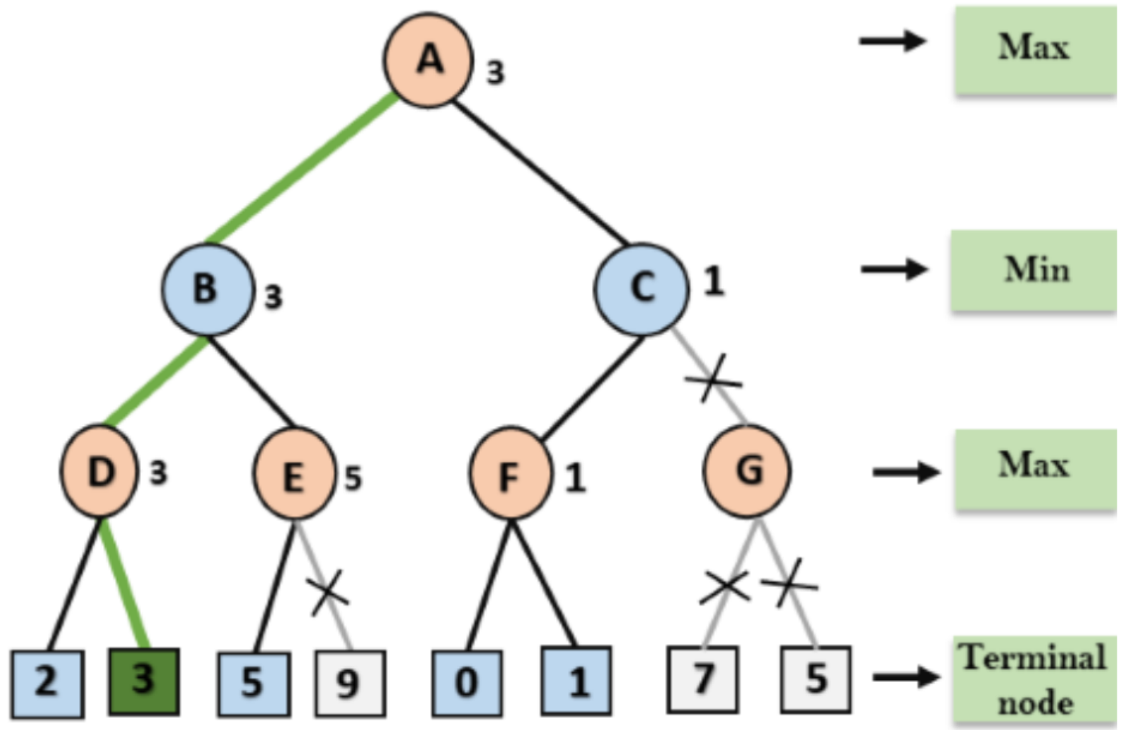


Step 7: Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



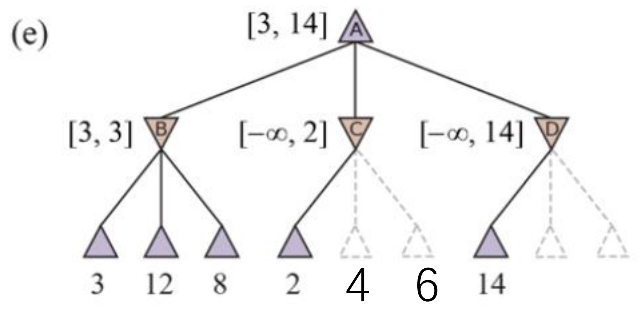
- At node C
  - MIN plays, Beta=1, it is at most 1.
- At node A,
  - MAX plays, alpha=3, it has at least 3.
- Alpha>=Beta, so C will not be chosen by MAX in node A.

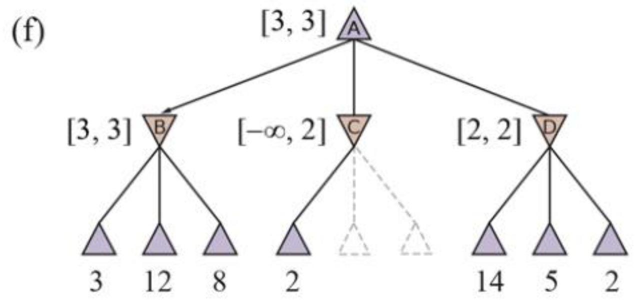
Step 8: C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



Move ordering

- Good move ordering improves effectiveness of pruning
  - If MIN expands the 3rd child of D first, the others are pruned
  - If MIN expands the 2nd or 3rd child of C first, then we cannot prun
- With “perfect” ordering, time complexity reduces to O($b^{m/2}$)
  - doubles solvable depth!
- With “random” ordering, time complexity reduces to O($b^{3m/4}$)





Outline

- Two-player zero-sum games
- Optimal Decisions in games, minimax search
  - Two players
  - Multiple players
- Alpha-Beta Pruning
- Heuristic Alpha-Beta Tree Search
- Stochastic games
- Monte Carlo Tree Search

Heuristic Alpha-Beta Tree Search

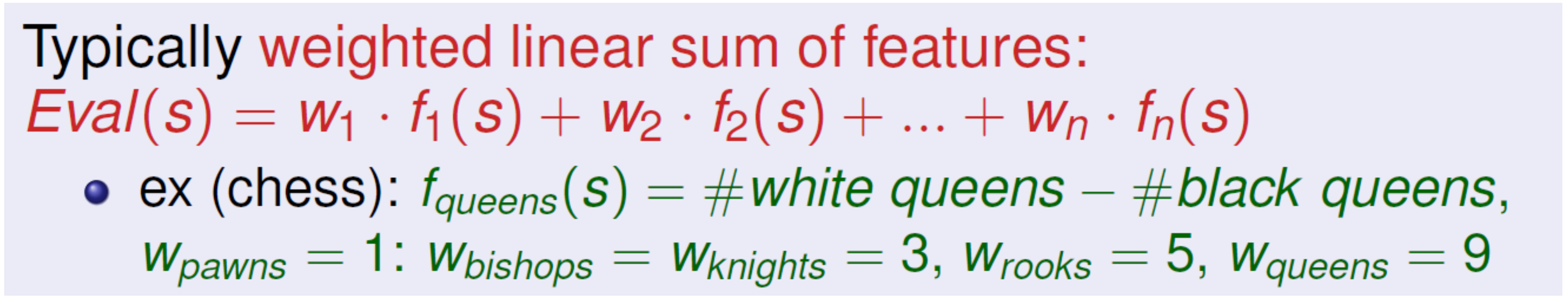
- Adversarial search with resource limits
- In realistic games, full search is impractical, even with Alpha-Beta pruning.
- In limited computation time, we cut off the search early and apply a heuristic evaluation function to states
    - cut off with limited depth
    - Treating nonterminal nodes as if they were terminal
- Changes w.r.t. minimax search
    - Terminal check -> cutoff check
    - Utility function -> heuristic evaluation function

$$\textsc{Minimax}(\mathrm{s}) = \begin{cases} \textsc{Utility}(s,\textsc{max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,\ a)) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,\ a)) & \text{if } \textsc{To-Move}(s) = \textsc{min} \end{cases}$$

$$\text{H-}\textsc{Minimax}(s,d) = \begin{cases} \textsc{Eval}(s,\textsc{max}) & \text{if } \textsc{Is-Cutoff}(s,d) \\ \max_{a \in Actions(s)} \text{H-}\textsc{Minimax}(\textsc{Result}(s,\ a), d+1) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \text{H-}\textsc{Minimax}(\textsc{Result}(s,\ a), d+1) & \text{if } \textsc{To-Move}(s) = \textsc{min}. \end{cases}$$

Evaluation function

- A heuristic evaluation function Eval(s, p) returns an estimate of the expected utility of state s to player p, just as the heuristic functions in informed search return an estimate of the distance to the goal.
- For terminal states, it must be Eval(s, p)=Utility(s, p)
- Utility(loss, p)<= Eval(s, p)<= Utility(win, p)
- Should be relatively cheap to compute
- Often use some features of the state


Typically weighted linear sum of features:

$Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + \ldots + w_n \cdot f_n(s)$

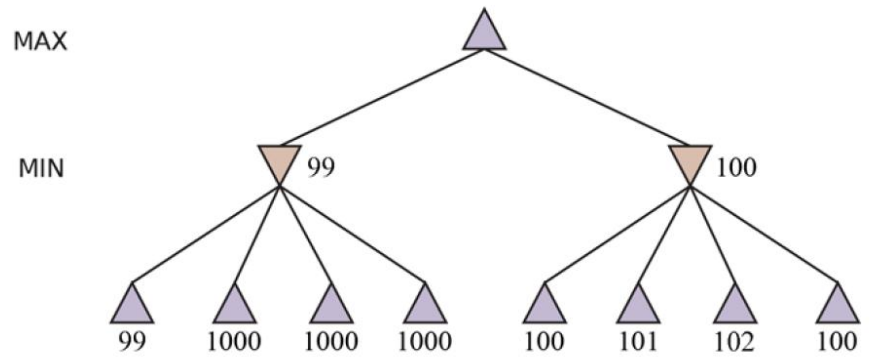
- ex (chess): $f_{queens}(s) = \#white\ queens - \#black\ queens$, $w_{pawns} = 1$: $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$

Cutting off search

- Set a fixed depth limit d so that Is-Cuttoff() returns true for all depth greater than d
- Or apply iterative deepening
  - d starts from 1, 2, 3, ⋯
  - When time runs out, returns the move selected by the deepest completed search

Limitations of Game Search Algorithms

- Alpha–beta search is vulnerable to errors in the heuristic function.
  - MAX chooses to go to right
  - If the evaluation has errors, left may be better
- Hand-craft evaluation functions
  - machine learning into evaluation functions.
- Individual moves, not higher level goals



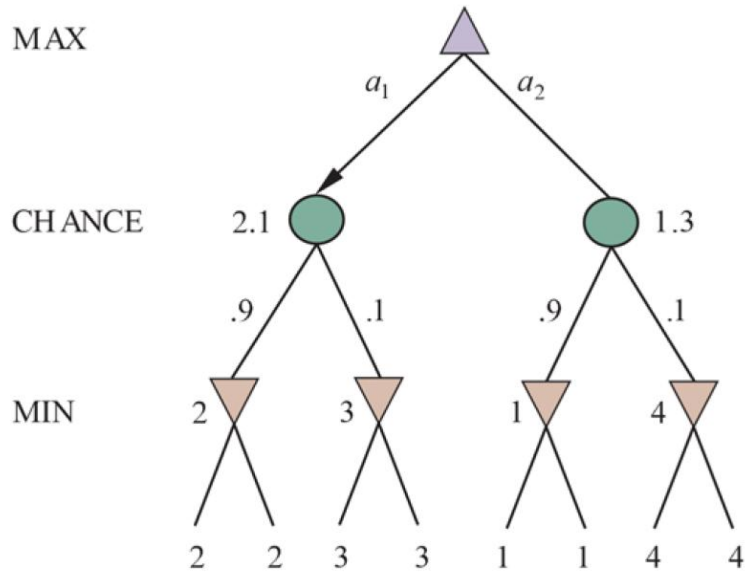
Outline

- Two-player zero-sum games
- Optimal Decisions in games, minimax search
    - Two players
    - Multiple players
- Alpha-Beta Pruning
- Heuristic Alpha-Beta Tree Search
- Stochastic games
- Monte Carlo Tree Search

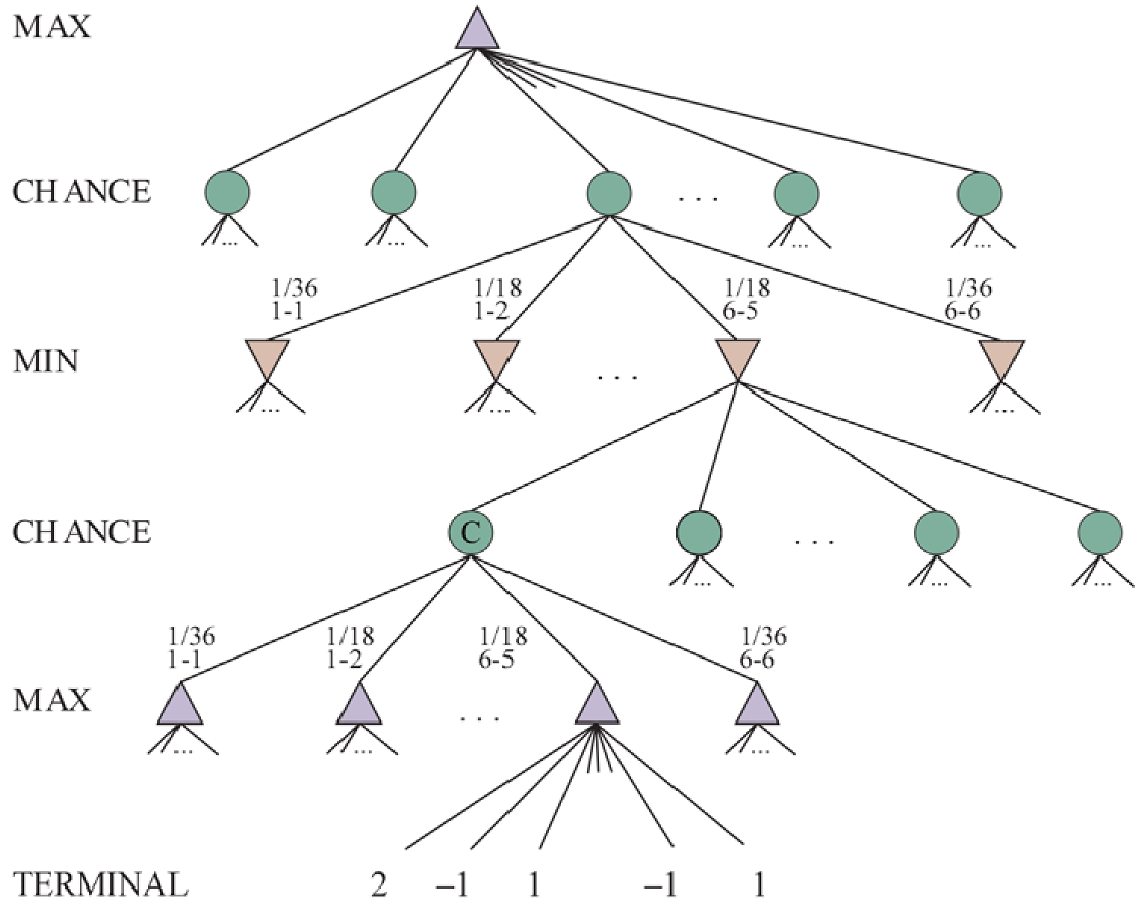
Stochastic games

- In real life, unpredictable external events may occur
- Stochastic Games mirror unpredictability by random steps:
    - e.g. dice throwing, card-shuffling, coin flipping, tile extraction, …
- Cannot calculate definite minimax value, only expected values

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s,\text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_a \text{if EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_a \text{if EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MIN} \\ \sum_r P(r)\text{EXPECTIMINIMAX}\Big(\text{RESULT}\big(s,r\big)\Big) & \text{if TO-MOVE}(s) = \text{CHANCE} \end{cases}$$



Game tree for a backgammon



Schematic game tree for a backgammon position.

Outline

- Two-player zero-sum games
- Optimal Decisions in games, minimax search
    - Two players
    - Multiple players
- Alph-Beta Pruning
- Heuristic Alph-Beta Tree Search
- Stochastic games
- Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS)

- Weaknesses of alph-beta tree search, illustrated by Go
  - Go has a **branching factor** that starts at 361, which means alpha–beta search would be limited to only 4 or 5 ply.
  - It is difficult to define a **good evaluation function** for Go because material value is not a strong indicator and most positions are in flux until the endgame.
- Main idea of MCTS
  - Does not use a heuristic evaluation function to estimate utility value
  - The value of a state is estimated as **the average utility** over a number of simulations of complete games starting from the state.
  - A simulation (also called a playout or rollout) chooses moves first for one player, than for the other, repeating until a terminal position is reached.
  - At that point the rules of the game (not fallible heuristics) determine who has won or lost, and by what score. For games in which the only outcomes are a win or a loss, “average utility” is the same as “win percentage.”

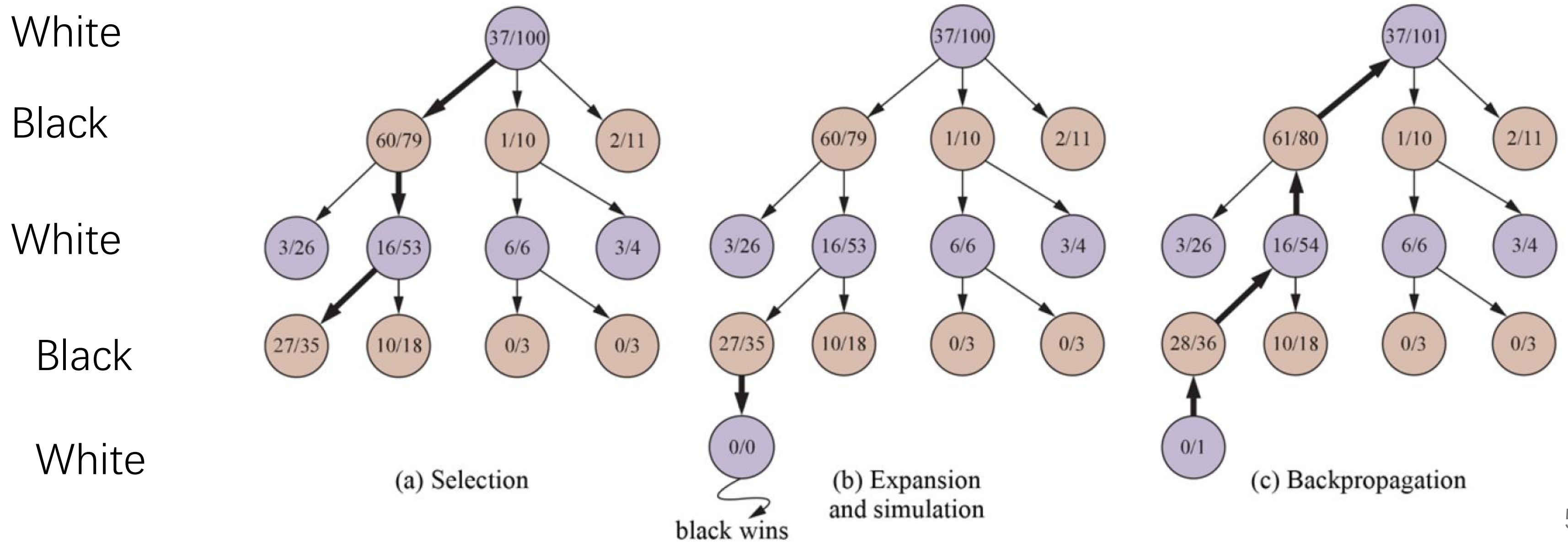
Exploration vs exploitation

- Selection policy
  - Use limited computational resources on the important parts of the game tree
- Exploration of states
  - that have had few playouts
- Exploitation of states
  - that have done well in past playouts, to get a more accurate estimate of their value

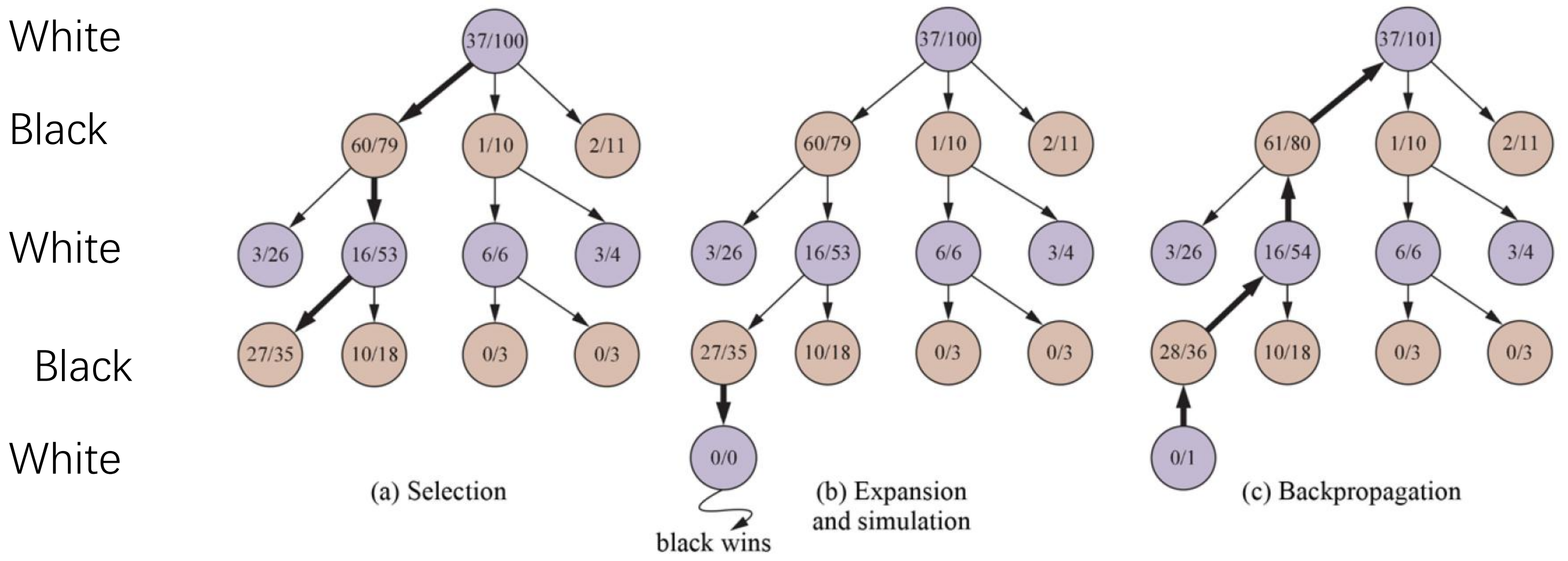
- MCTS: Selection, Expansion, Simulation, and backpropagation

Four steps in MCTS

- Selection: Starting at the root of the search tree, we choose a move (guided by the selection policy), leading to a successor node, and repeat that process, moving down the tree to a leaf.
- Expansion: We grow the search tree by generating a new child of the selected node.



- Simulation: We perform a playout from the newly generated child node, choosing moves for both players according to the playout policy. These moves are not recorded in the search tree.
    - In the figure, the simulation results in a win for black.
- Backpropagation: We now use the result of the simulation to update all the search tree nodes going up to the root.
    - Since black won the playout, black nodes are incremented in both the number of wins and the number of playouts, so 27/35 becomes 28/26, and 60/79 becomes 61/80. Since white lost, the white nodes are incremented in the number of playouts only, so 16/53 becomes 16/54, and the root 37/100 becomes 37/101.



MCTS

$$UCB1(n) = \boxed{\frac{U(n)}{N(n)}} + C \times \boxed{\sqrt{\frac{\log N(\textsc{Parent}(n))}{N(n)}}}$$

Exploitation

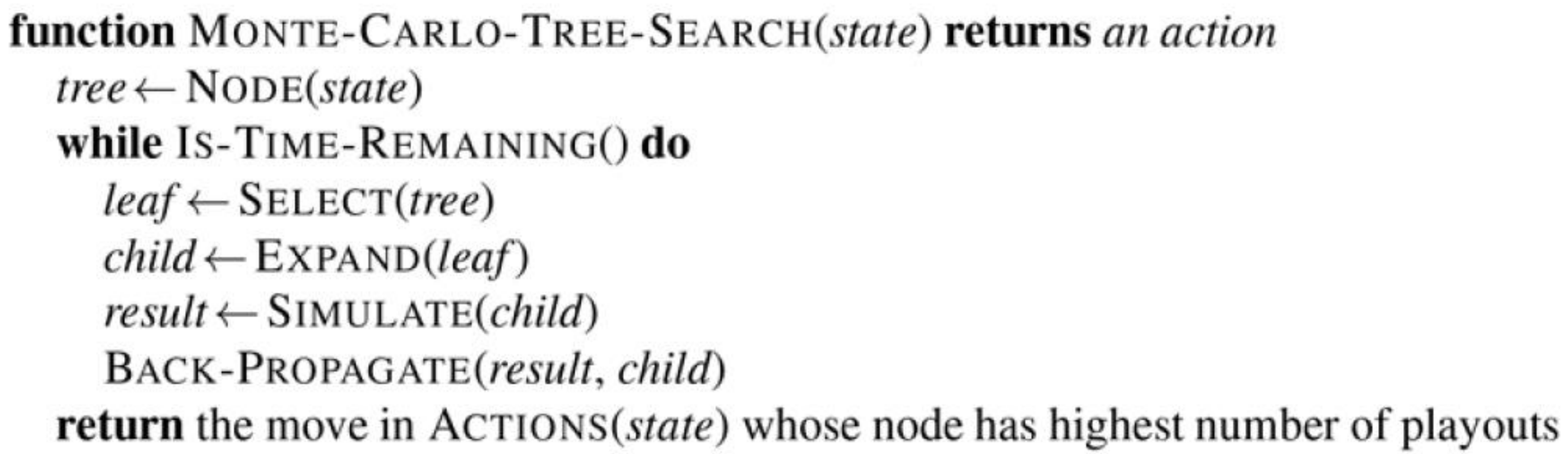
Exploration

- We repeat these four steps either for a set number of iterations, or until the allotted time has expired, and then return the move with **the highest number of playouts**.
- Selection policy: Upper Confidence bounds applied to Trees (UCT)
  - Ranks each possible move based on an upper confidence bound formula UCB1.
  - For a node n,
    - U(n) is the total utility
    - N(n) is the number of playouts through node n
    - Exploitation term: U(n)/N(n), the average utility of n.
      - Higher average utility means more promising.
    - Exploration term: square root part
      - N(n) on the denominator: when N(n) is small (it has only been explored a few times), has more chance to be selected.
    - C is a constant that balances the exploitation and exploration terms.

Final choice of move

- The move with the highest number of playouts is returned.
- What about the node with the highest average utility?
    - The idea is that a node with 65/100 wins is better than one with 2/3 wins, because the latter has a lot of uncertainty.
- In any event, the UCB1 formula ensures that the node with the most playouts is almost always the node with the highest win percentage, because the selection process favors win percentage more and more as the number of playouts goes up.

MCTS



```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
    tree ← NODE(state)
    while IS-TIME-REMAINING() do
        leaf ← SELECT(tree)
        child ← EXPAND(leaf)
        result ← SIMULATE(child)
        BACK-PROPAGATE(result, child)
    return the move in ACTIONS(state) whose node has highest number of playouts
```

Lecture 5 ILOs

- Adversarial Search and Games
  - Game theory, problem settings
    - # players, deterministic vs. stochastic, zero-sum vs. general games
  - Optimal Decisions in games
    - binary outcome (win or lose): AND-OR tree search
    - Multiple outcomes: minimax search
    - Stochastic games: minimax search with chance nodes computing expected utilities
  - Alpha-Beta Pruning
  - Heuristic Alpha-Beta Tree Search
    - Cutoff, use heuristic evaluation
  - Monte Carlo Tree Search
    - Selection, Expansion, Simulation, and backpropagation
    - Exploration and exploitation