COMS W4701: Artificial Intelligence

Lecture 8: Heuristic Game Playing

Tony Dear, Ph.D.

Department of Computer Science

School of Engineering and Applied Sciences

Today

- Depth limits and evaluation functions

- Stochastic games

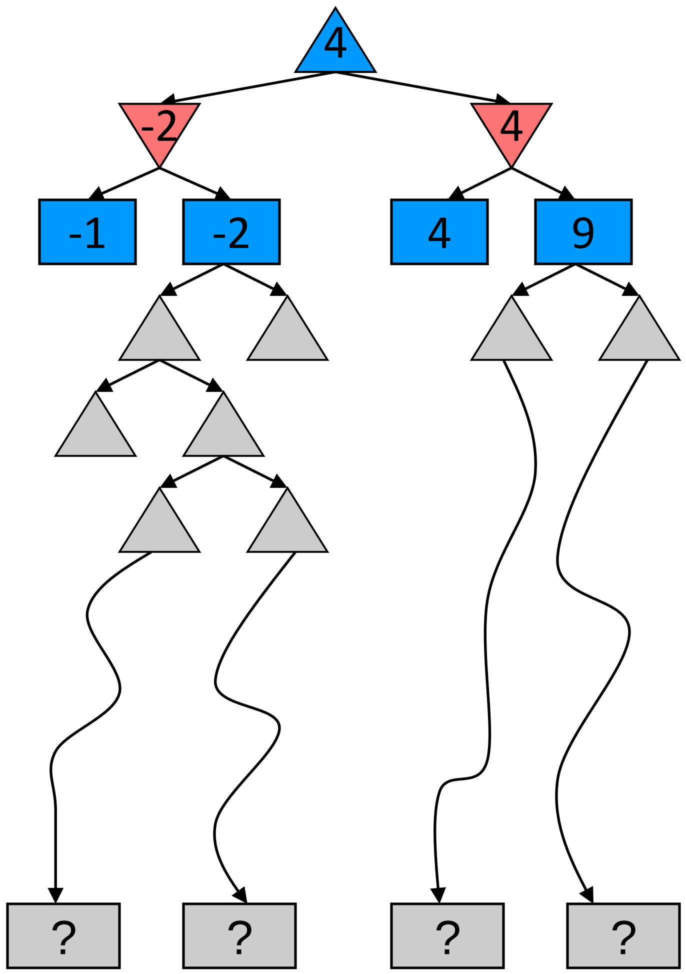
- Monte Carlo tree search

Imperfect Decisions

- Problem: Most game trees still too big
- α-β can help but still need to find terminal nodes

- Heuristic: Treat non-terminal nodes as terminals!
- **Evaluation function** returns an *estimate* of this "terminal" state's utility

- **Cutoff test** decides when to do this
- No more guarantee of optimality



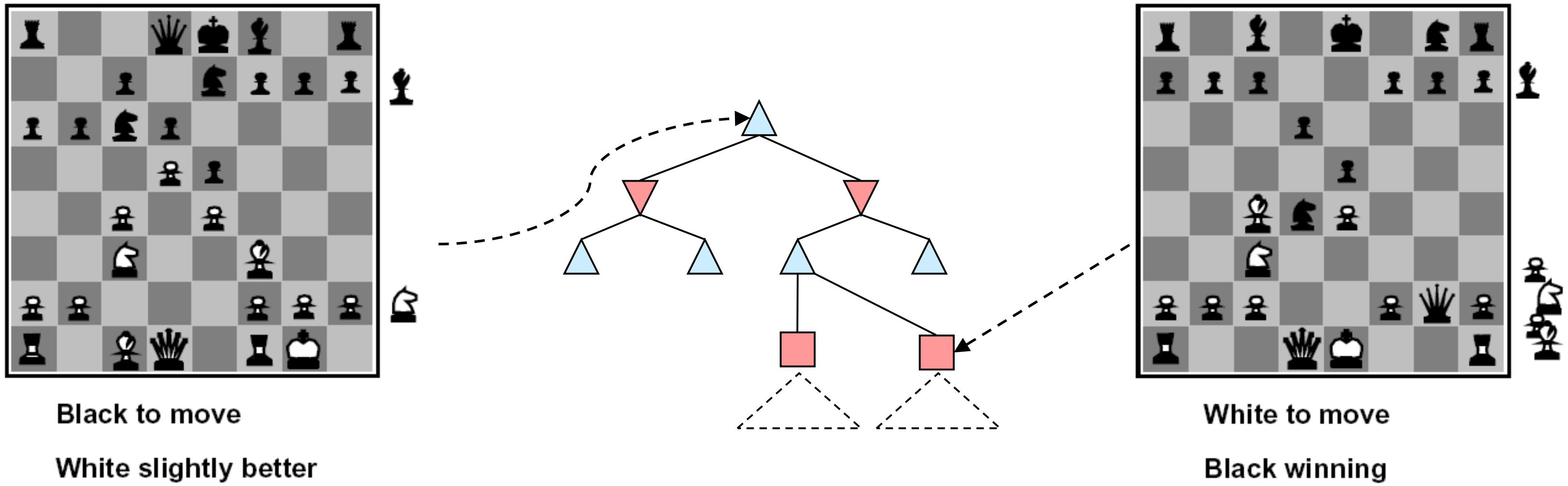
Evaluation Functions

- Evaluation functions are *estimates* of a state's utility

- Evaluation functions…
  - Should be equal to utility values for terminal nodes
  - Should be reflective of minimax values for nonterminal nodes
  - Should be efficient to compute and based on game knowledge/rules

- One common eval function: weighted linear sum of game **features**
- Features represent categories or equivalence classes of states

Example: Chess



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

- Features may be derived from expert knowledge of common *categories* of states
- E.g., one feature for each type of piece, attack formations, king safety positions, etc.
- Weights correspond to *material values* of each feature
- Linear weighting assumes features are independent of each other

Depth Limits

- What cutoff test to use?
- Simple approach: Use a fixed depth limit, or use iterative deepening

- Problem: Cutting off search at unstable positions can hide valuable info
- **Horizon effect**: Agent may favor moves that push danger "over the horizon", appears to have been mitigated but actually delayed

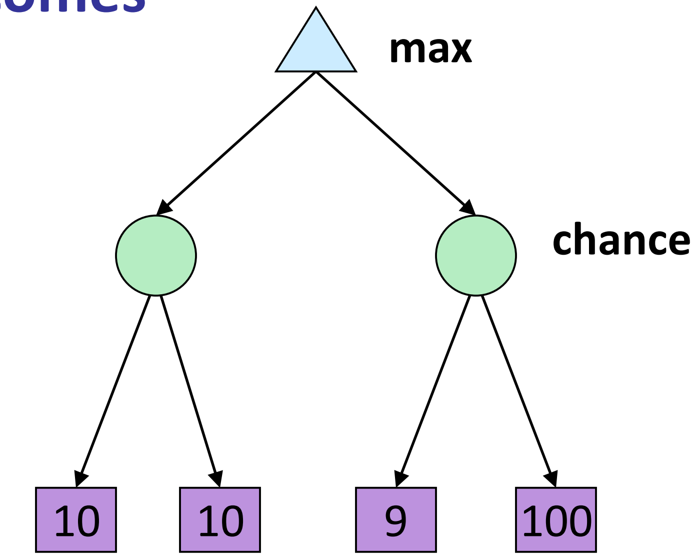
- **Quiescence search**: Evaluate states based on their stability ("quiescence"), e.g., chess moves that do not lead to imminent captures

Stochastic Games

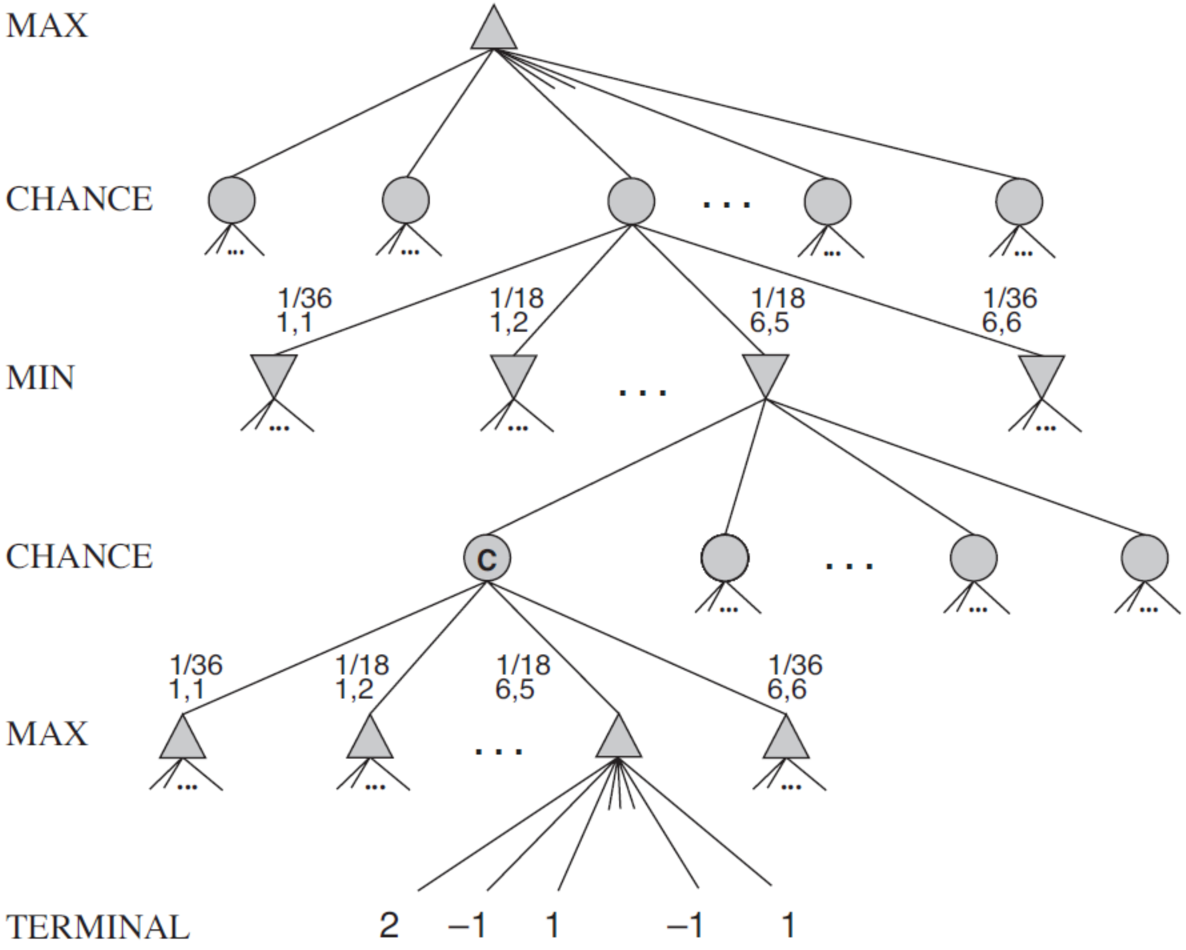
- Many games contain elements of *stochasticity*
- Opponents playing suboptimally (e.g., inexperienced) or randomly
- Explicit random elements (e.g., dice rolling)

- Instead of worst-case scenarios, we consider **expected outcomes**
- Instead of minimax, we compute **expectiminimax values**

$$\textsc{Expectiminimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \\ \sum_r P(r)\textsc{Expectiminimax}(\textsc{Result}(s,r)) & \text{if } \textsc{Player}(s) = \textsc{chance} \end{cases}$$



Example: Backgammon

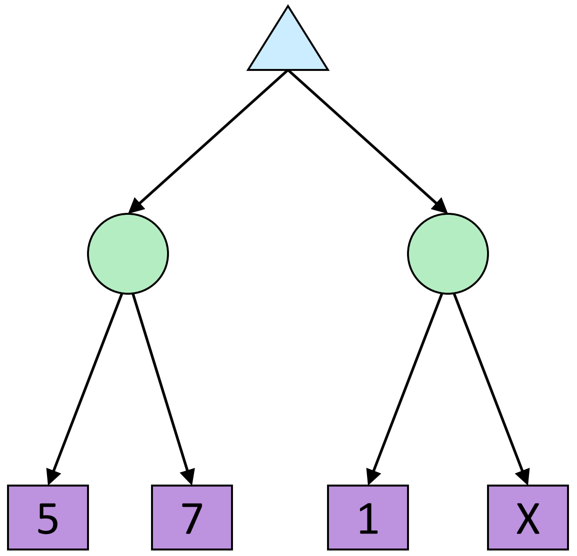


Heuristics for Expectiminimax

- With deterministic games, α-β can cleverly prune the game tree and ignore outcomes that cannot occur

- With stochastic games, *any* outcomes may occur
- Even the unlikeliest outcomes must be examined

- We may be able to prune if we can *bound* utility values
- We can also *forward-prune* by ignoring bad looking moves a priori (as in beam search)



All utilities ≤ 10

Monte Carlo Tree Search

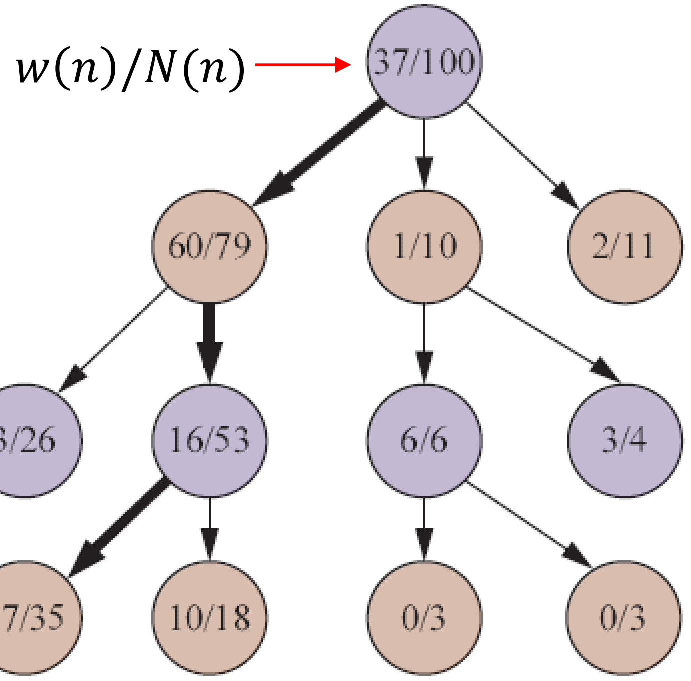
- Minimax and its variants are **type A** strategies: search wide but shallow
- For games like Go with large branching factors, **type B** strategies like MCTS work better: search deep but narrow

- *Traverse* and *expand* new nodes in promising portions of the game tree
- For a partial tree, *simulate* the game from a leaf node to a terminal
- *Backpropagate* the result up to all nodes along traversed path

- **MCTS**: Repeat the above many times and choose the most traversed move

Game Tree Structure

- As in minimax and α-β, we build and expand a game tree starting from current state
- Proceed by iteration: One or more nodes added to the tree each time
- Every node keeps track of its value and N
- After each MCTS iteration, some node values will be updated



$w(n)$ = number of wins from n

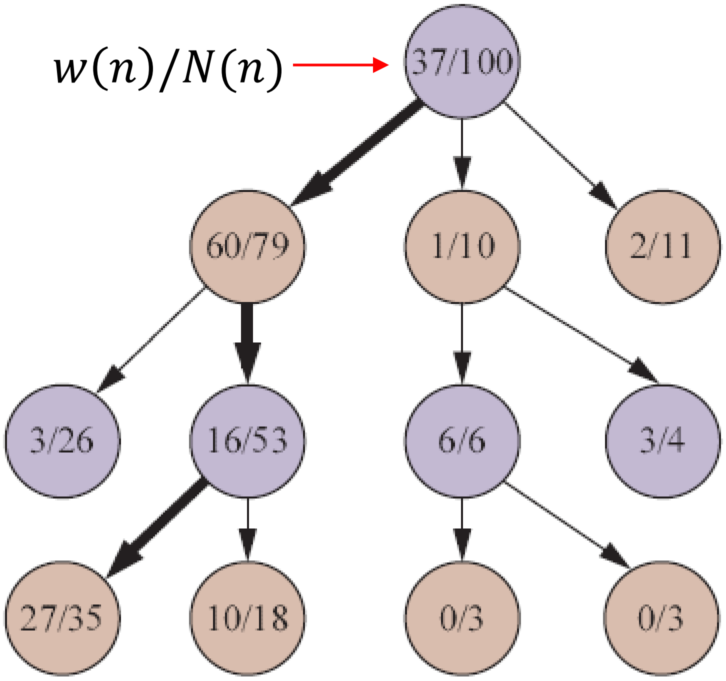
$N(n)$ = number of rollouts from n

Node value = $w(n)/N(n)$

Selection

- We traverse the tree until arriving at a terminal node or one with successors not yet in the tree
- A **selection policy** balances *exploitation* (highest values) with *exploration* (less visited nodes)
- **UCT** method for traversal: From each node, move to successor maximizing the following:

$$UCT(n) = \underbrace{\frac{w(n)}{N(n)}}_{\text{“exploitation”}} + \alpha \underbrace{\sqrt{\frac{\ln N\big(parent(n)\big)}{N(n)}}}_{\text{“exploration”}}$$



$w(n)$ = number of wins from n

$N(n)$ = number of rollouts from n

α = exploration parameter

Upper Confidence Bound

- The *exploration* or *uncertainty* term of the UCT value is $\alpha\sqrt{\frac{\ln N(parent(n))}{N(n)}}$

- $\alpha \geq 0$: Tunable parameter controlling weight of this term
- $\sqrt{\ln(parent(N))}$: Proportional to number of rollouts of parent node, can just treat as a constant since same value for all other children nodes

- $N(n)$ is the number of rollouts containing n, or *samples* for estimating the value of n
- $1/\sqrt{N(n)}$ is proportional to the standard deviation of this estimate
- Each rollout through n increments $N(n)$ and *decreases* its uncertainty

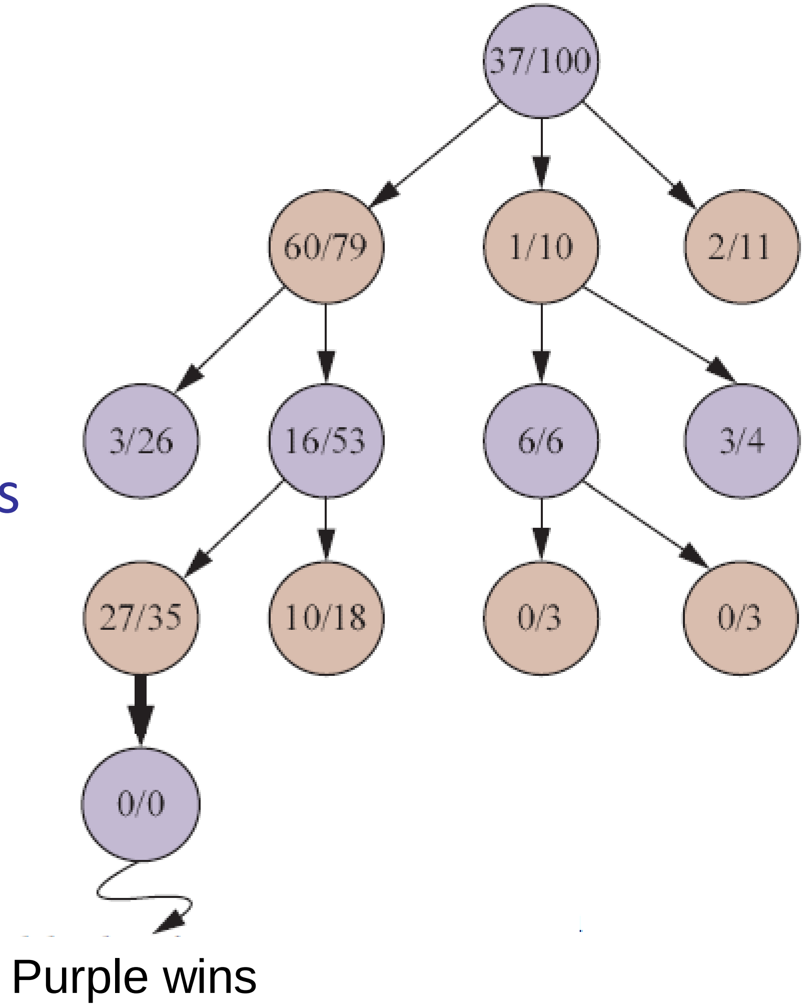
Expansion and Simulation

- Expansion: Add a child of the selected node to the tree

- **Playout policy** simulates a *rollout* from the new child
- Basic example: Select moves at random

- Better policies guide moves toward good or clever ones
- May incorporate game-specific heuristics
- May be obtained from deep learning (e.g. AlphaGo)

- Record the game result, but *not* the game states seen

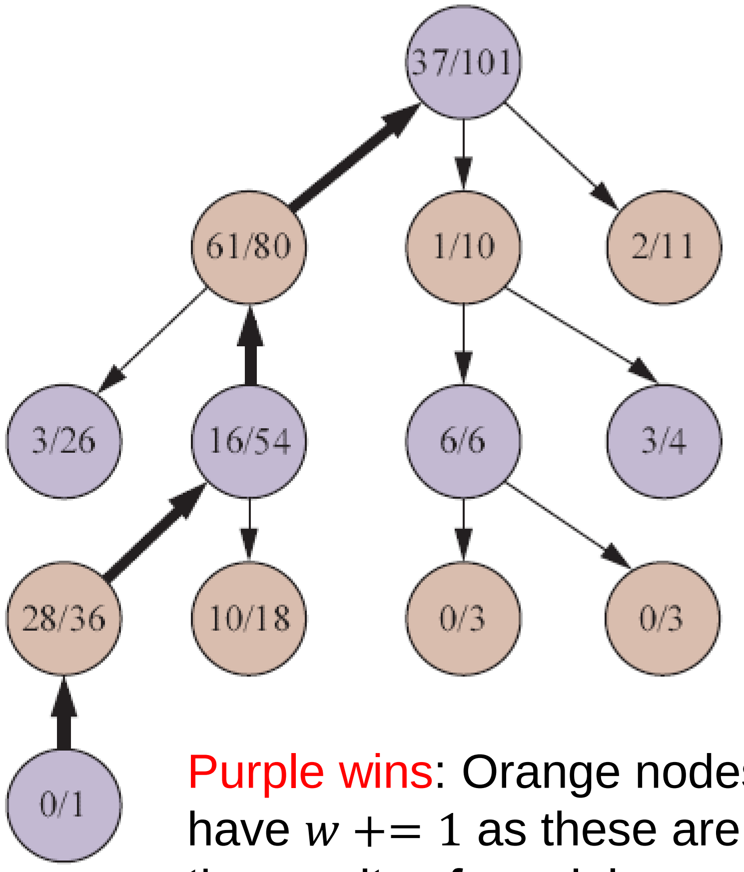


Backpropagation

- Once simulation finishes, pass information back up to *all* nodes along path back to root

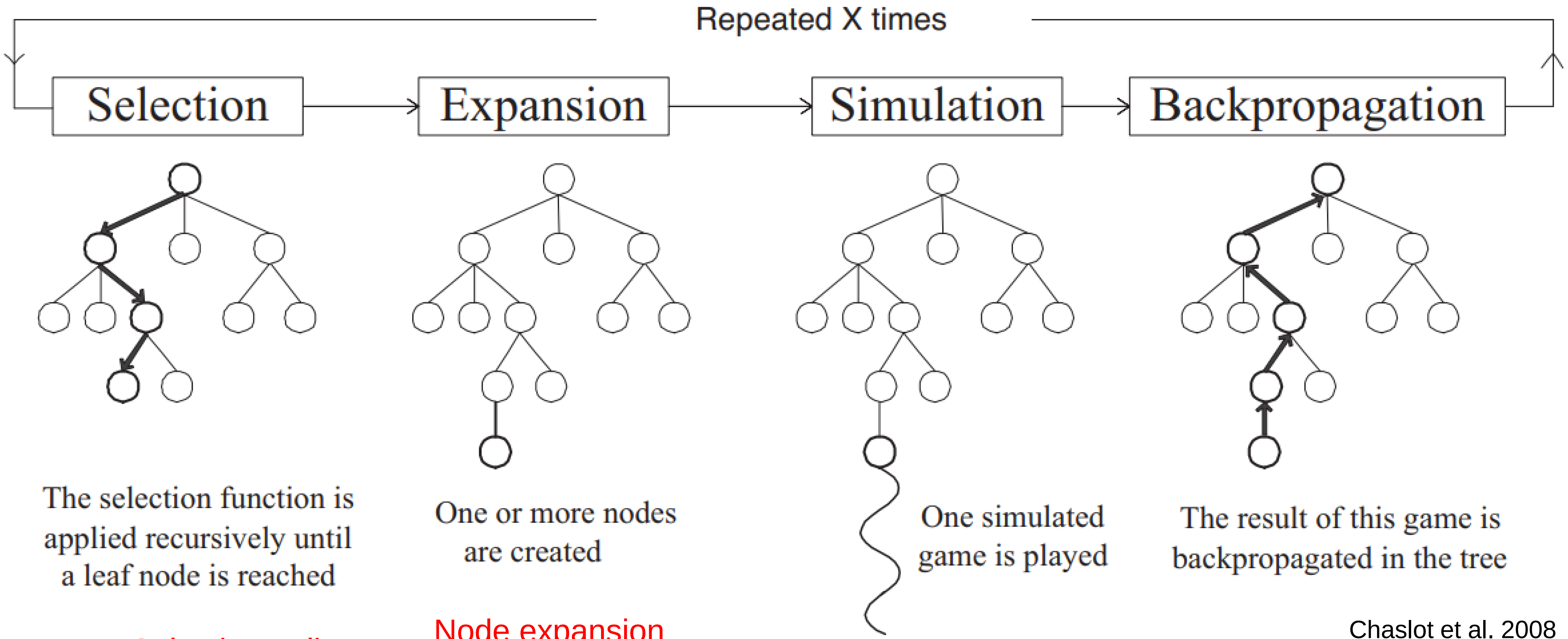
- Increment N value of all nodes on path
- If tracking wins, increment $w(n)$ of nodes whose *parent* is the winning player

- After sufficient rounds of MCTS, the root chooses the most frequent action (highest N) as the move to play
- More robust and less variability than the move with the highest value (if different)



Purple wins: Orange nodes have w += 1 as these are the results of purple's moves

Monte Carlo Tree Search



NEW: Selection policy

Node expansion (as before)

NEW: Playout (rollout) policy

Chaslot et al. 2008

MCTS vs Alpha-Beta

- MCTS simulations are linear in game depth
- For a game with branching factor of 32 and depth of 100, alpha-beta search down to 12 ply deep is equivalent to 10^7 MCTS simulations

- MCTS tends to do better when branching factor is high
- Also less sensitive than α-β to inaccurate eval functions
- Also good for brand new games with no predefined eval functions at all!

- Stochastic nature of MCTS: no guarantee of exploring all good moves

Summary

- Practical adversarial search often means exploring portions of game tree
- Alpha-beta search can be cut off at finite depth limits by applying evaluation functions to non-terminal states

- Games with stochasticity can be handled with expectiminimax search

- Games with high branching factors can be searched deep, not wide
- MCTS builds a partial tree over many iterations, simulating many playouts to the end of the game