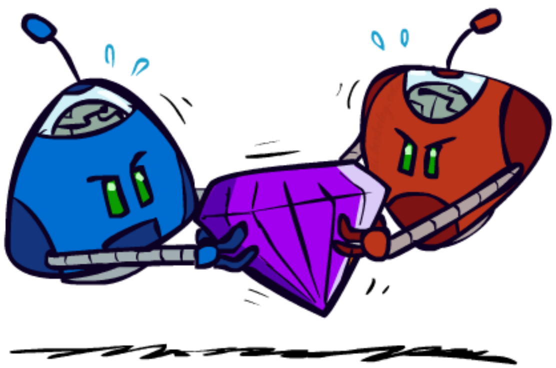
CMSC 471: Games Uncertainty and Utilities

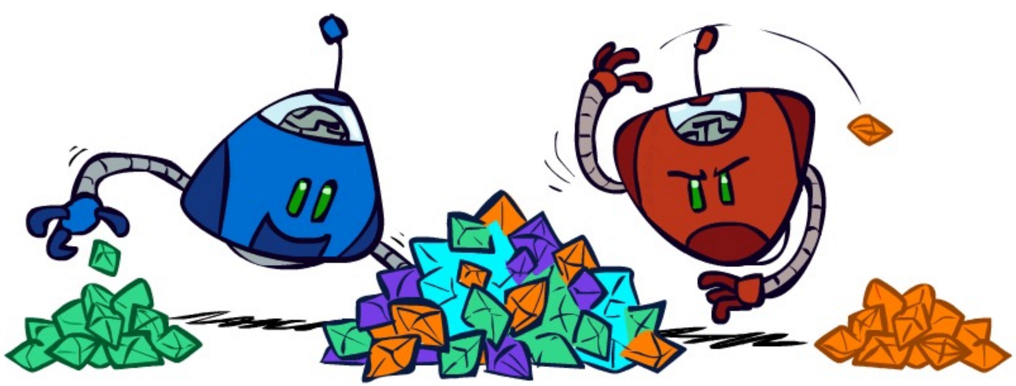
KMA Solaiman

ksolaima@purdue.edu

Slides courtesy Tim Finin and Frank Ferarro. Some material adopted from notes by Andreas Geyer-Schulz and Chuck Dyer,
Some materials adopted from slides by Dan Klein and Pieter Abbeel at UC Berkeley [http://ai.berkeley.edu]

Zero-Sum Games

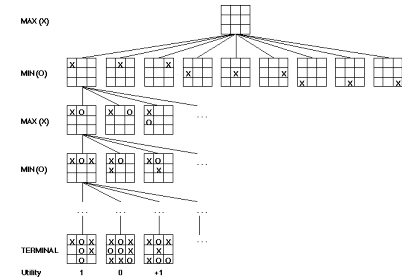




- Zero-Sum Games
  – Agents have opposite utilities (values on outcomes)
  – **Lets us think of a single value that one maximizes and the other minimizes**
  – Adversarial, pure competition

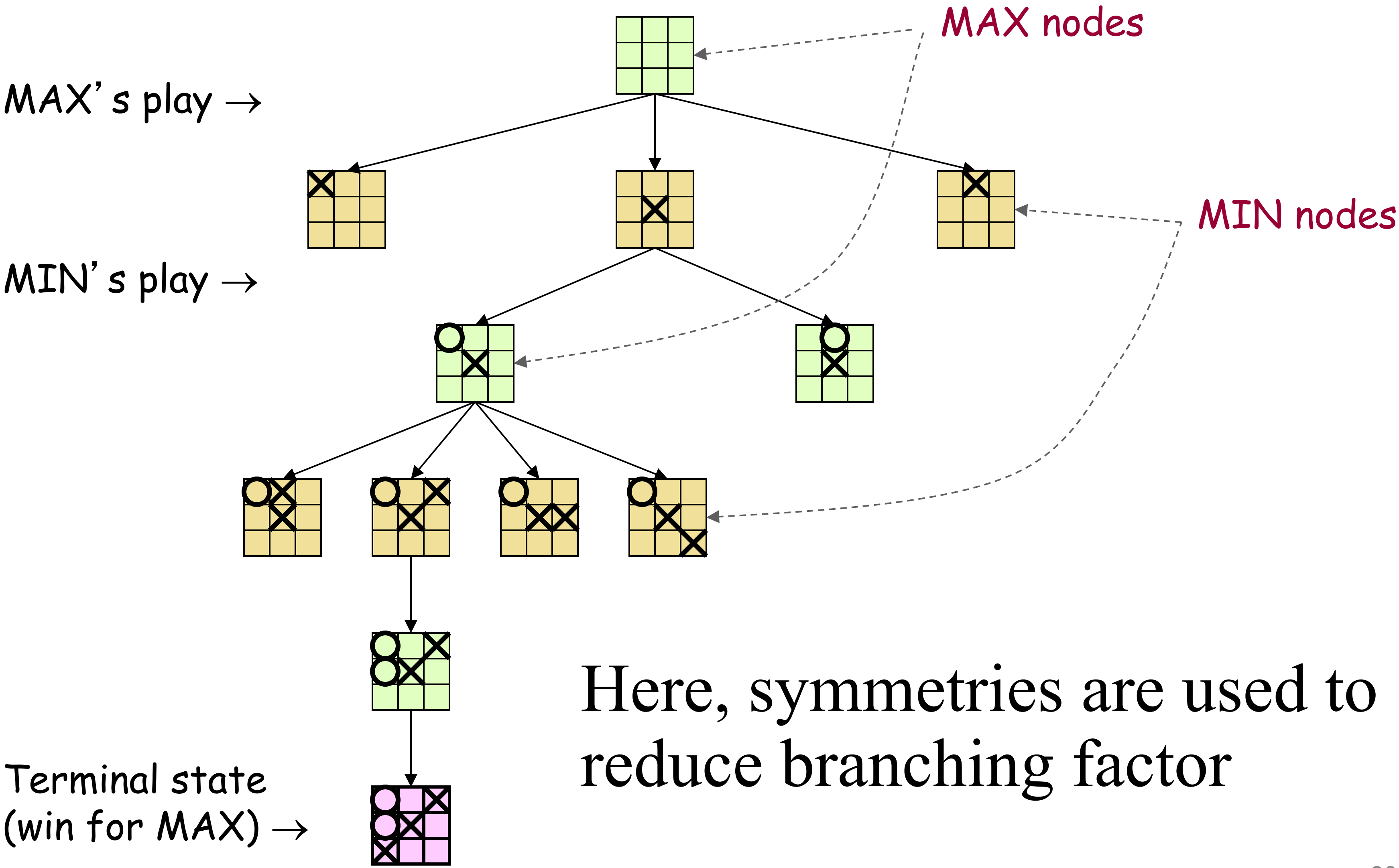
- General Games
  – Agents have independent utilities (values on outcomes)
  – Cooperation, indifference, competition, and more are all possible
  – More later on non-zero-sum games

Game trees



- Problem spaces for typical games are trees
- Root node is current board configuration; player must decide best single move to make next
- **Static evaluator function** rates board position **f(board):**real, > 0 for me; < 0 for opponent
- Arcs represent possible legal moves for a player
- If **my turn** to move, then root is labeled a "**MAX**" node; otherwise it’s a "**MIN**" node
- Each tree level’s nodes are all MAX or all MIN; nodes at level i are of opposite kind from those at level i+1

Game Tree for Tic-Tac-Toe

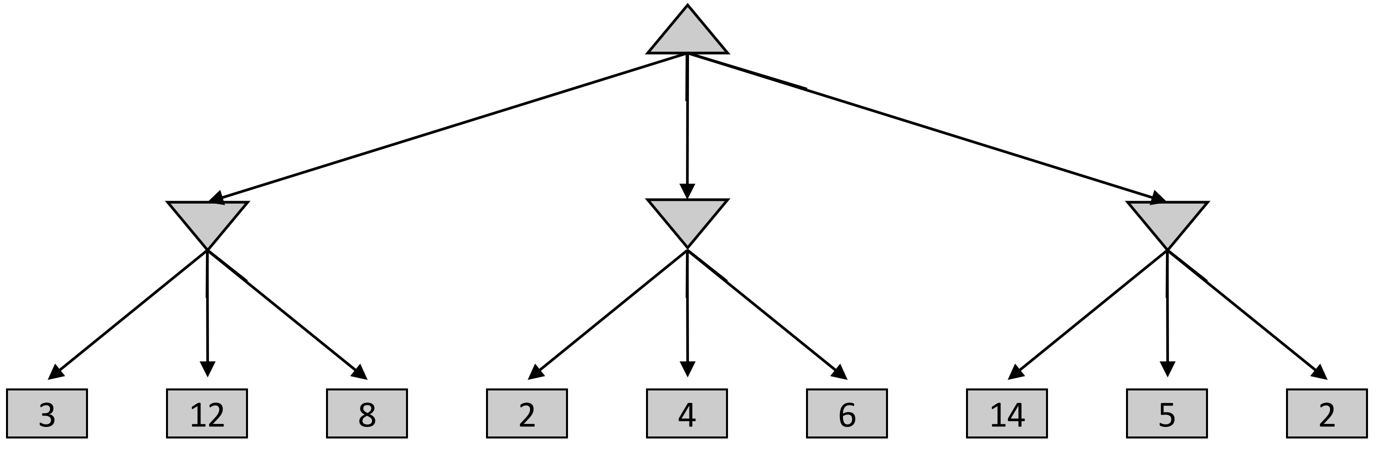


Here, symmetries are used to reduce branching factor

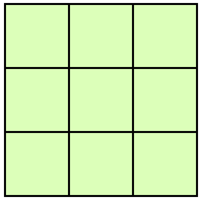
Minimax Algorithm

1. Create MAX node with current board configuration
2. *Expand nodes to some* ***depth*** *(a.k.a.* ***plys****) of* ***lookahead*** *in game*
3. Apply evaluation function at each **leaf** node
4. ***Back up*** values for each non-leaf node until value is computed for the root node
   - At MIN nodes: value is **minimum** of children's values
   - At MAX nodes: value is **maximum** of children's values
5. Choose move to child node whose backed-up value determined value at root

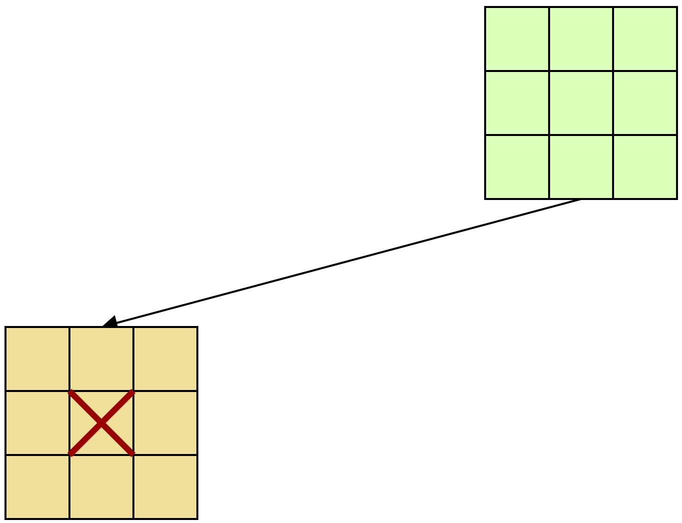
Minimax Example



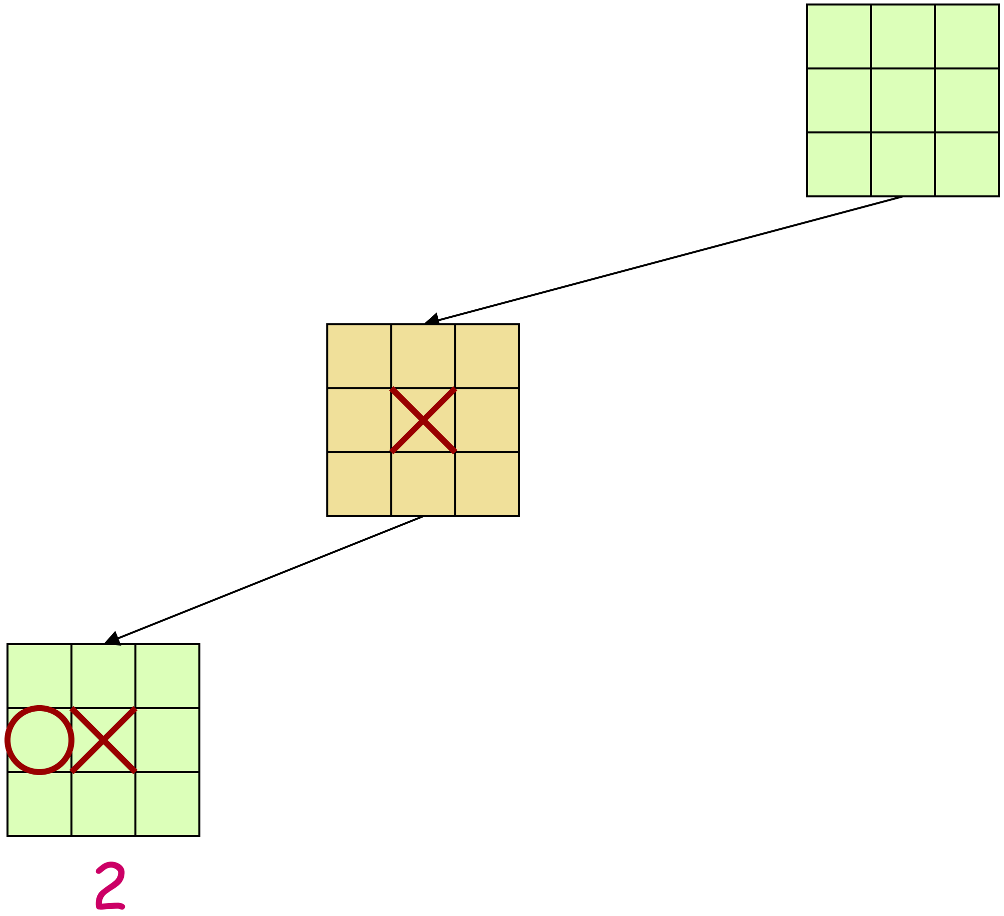
Alpha-Beta Tic-Tac-Toe Example



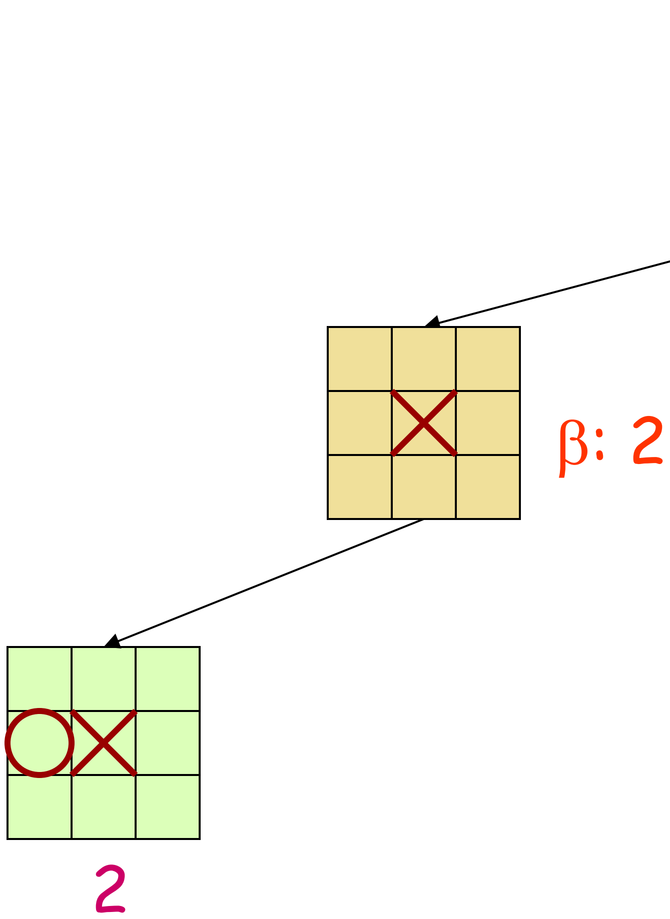
Alpha-Beta Tic-Tac-Toe Example



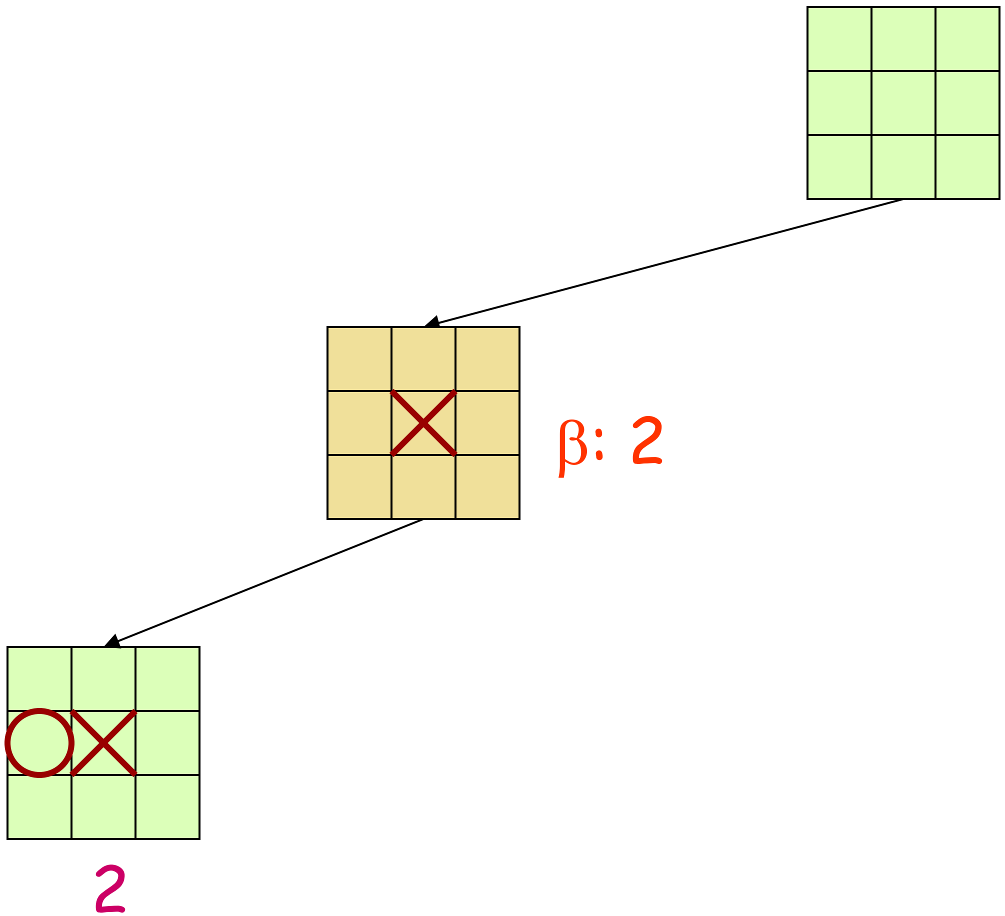
Alpha-Beta Tic-Tac-Toe Example



Alpha-Beta Tic-Tac-Toe Example

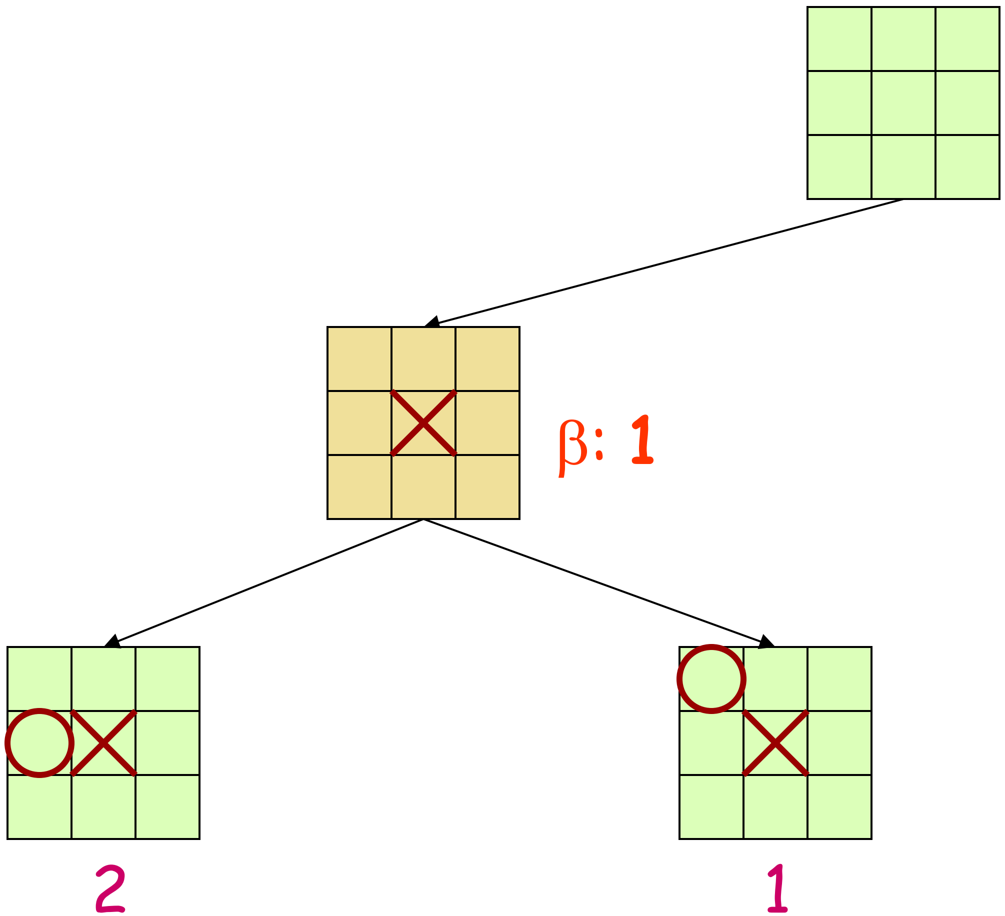


Alpha-Beta Tic-Tac-Toe Example

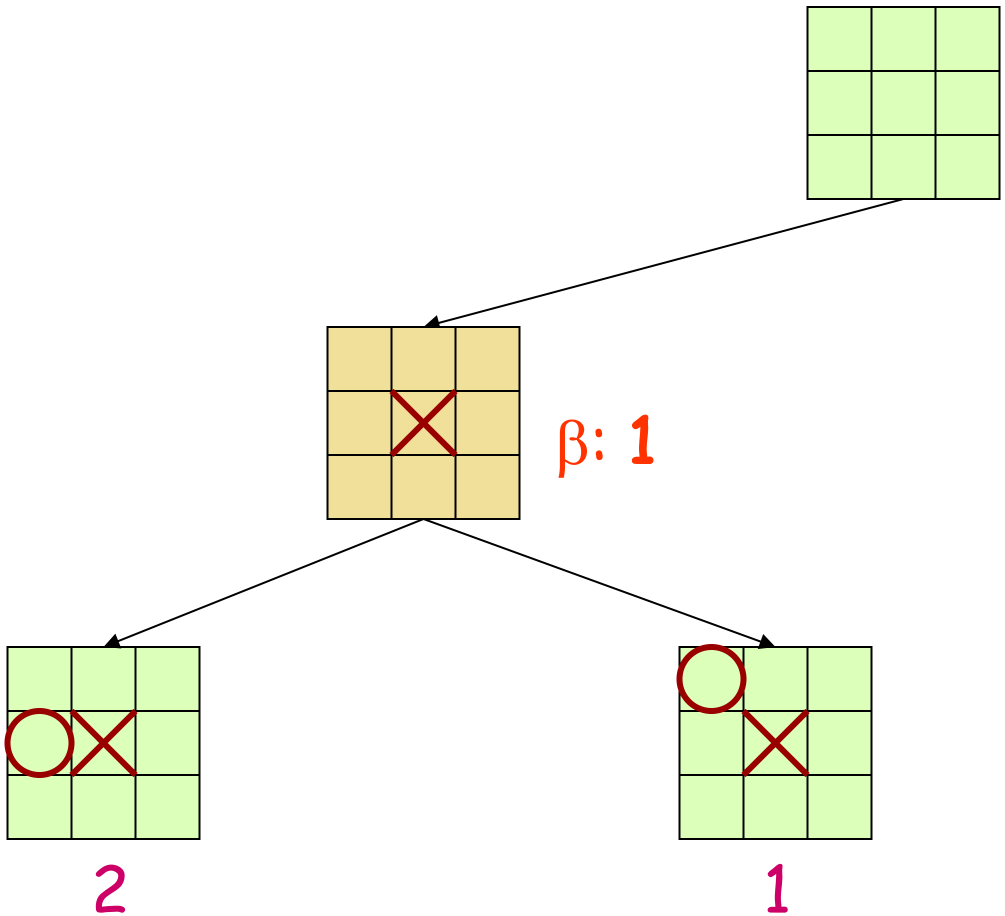


Beta value of a MIN node is **upper** bound on final backed-up value; it can never increase

Alpha-Beta Tic-Tac-Toe Example

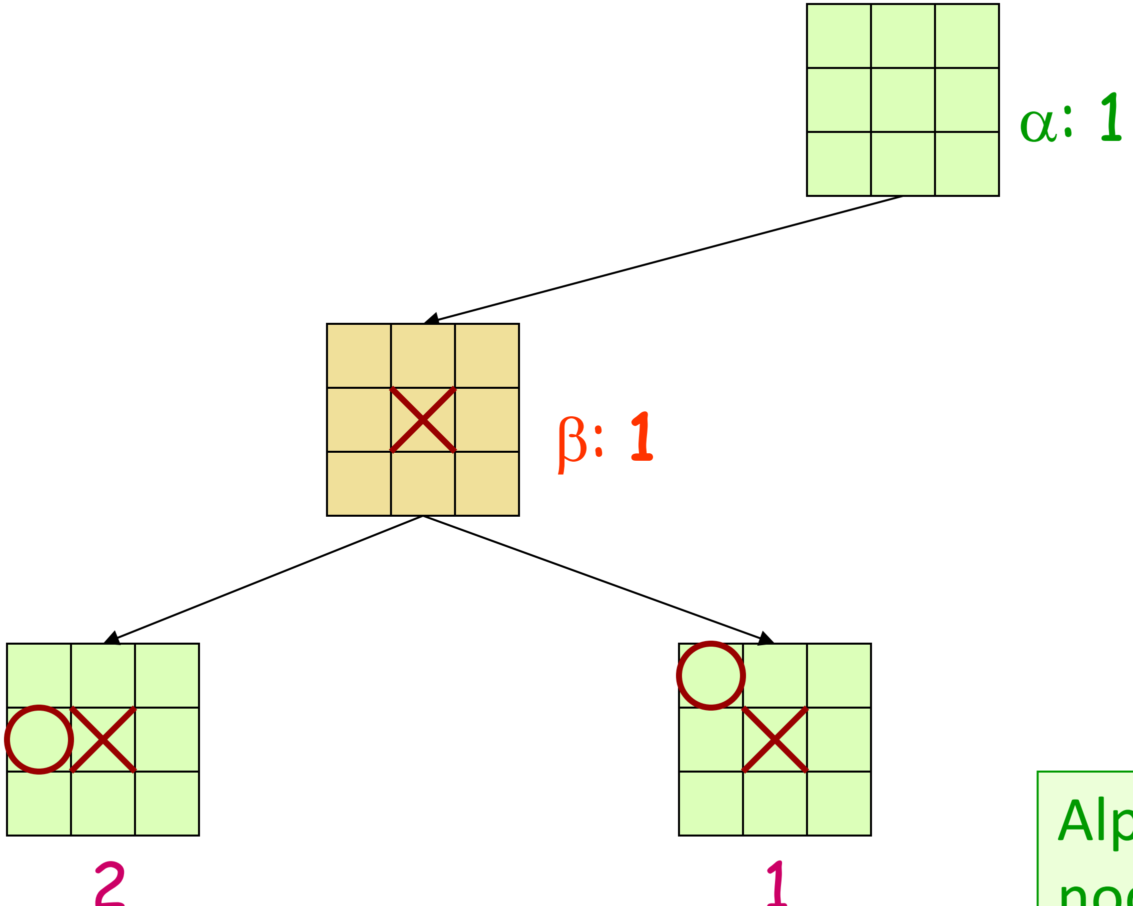


Alpha-Beta Tic-Tac-Toe Example



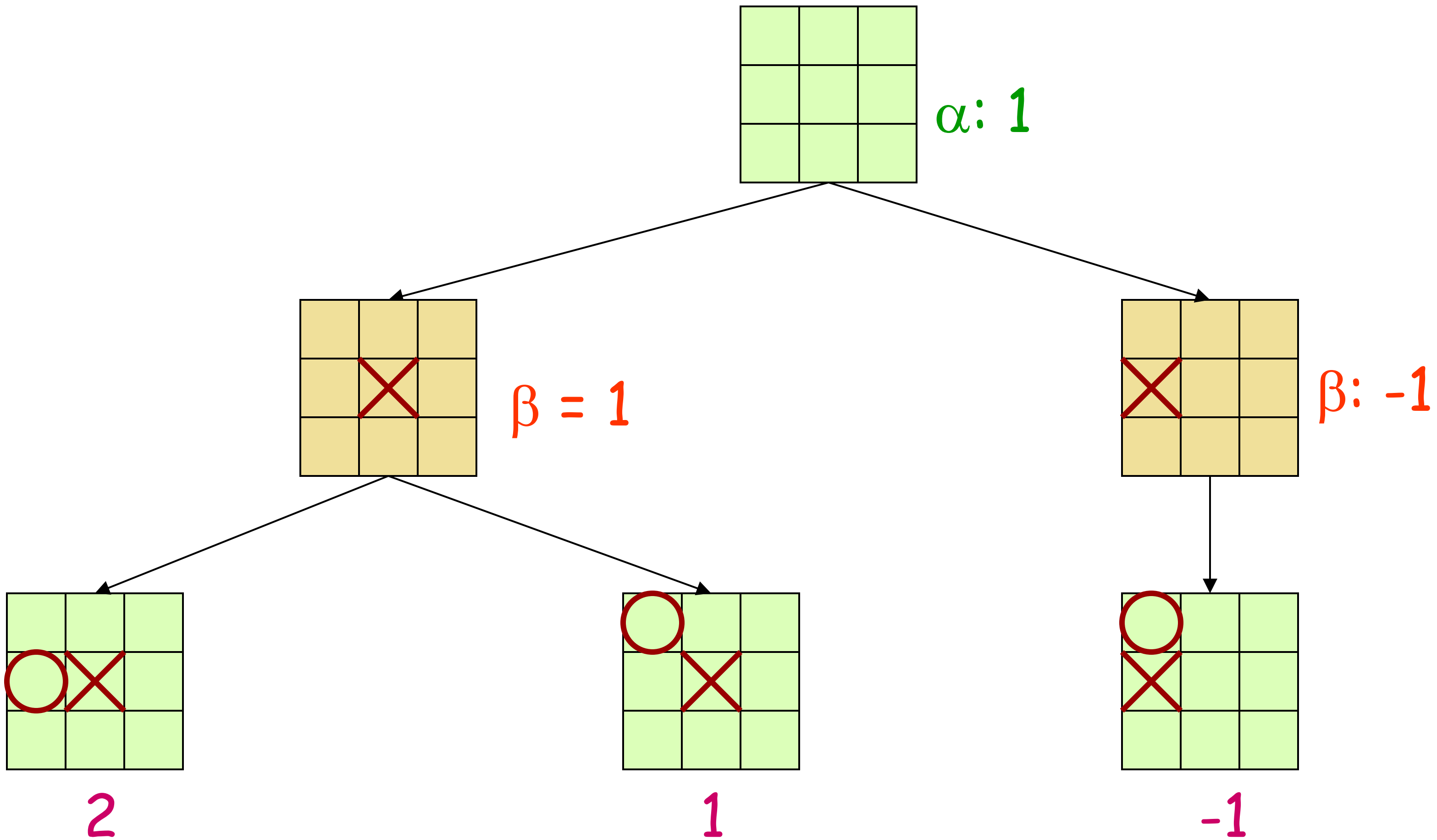
Beta value of a MIN node is **upper** bound on final backed-up value; it can never increase

Alpha-Beta Tic-Tac-Toe Example

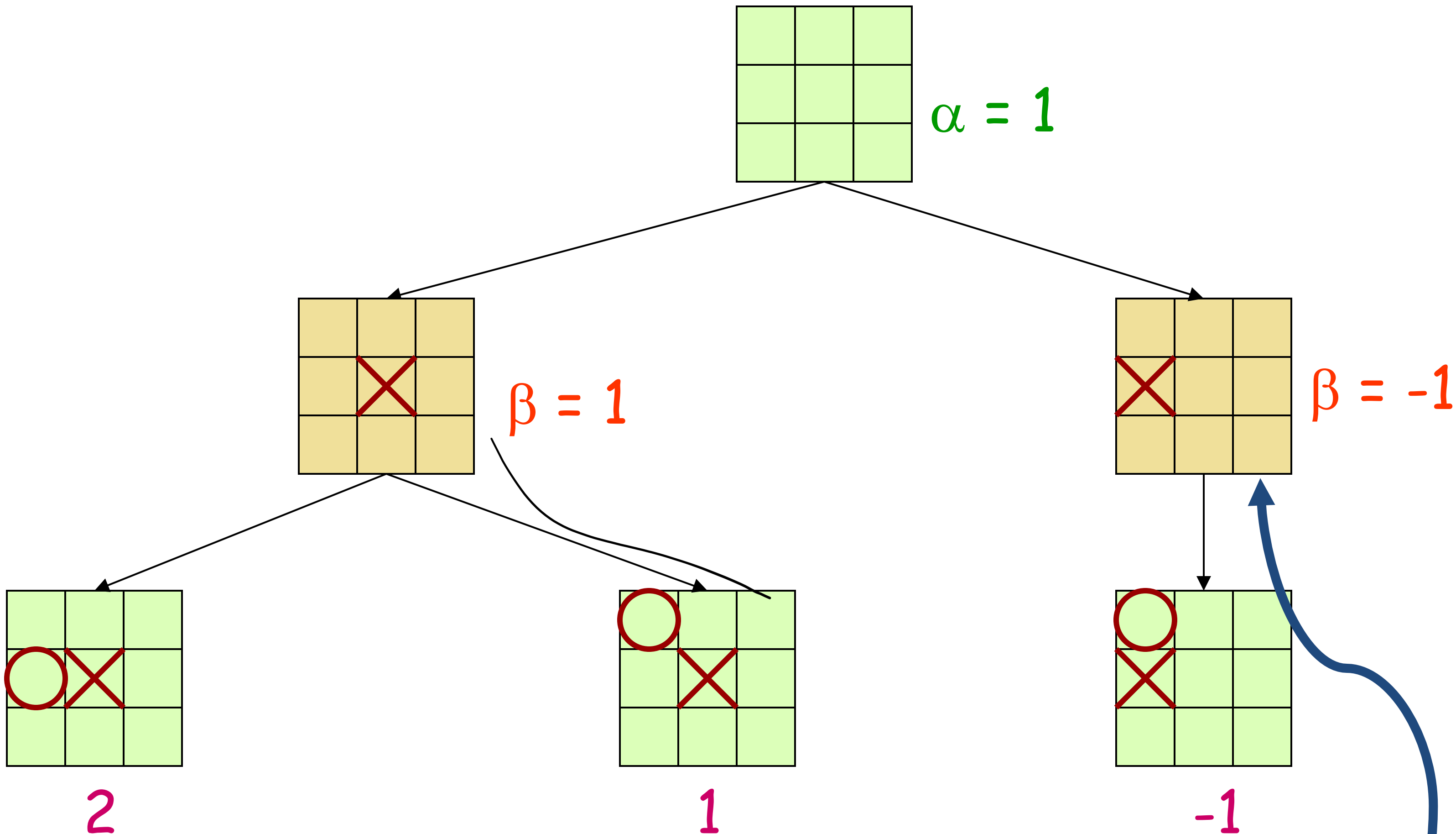


Alpha value of MAX node is **lower** bound on final backed-up value; it can never decrease

Alpha-Beta Tic-Tac-Toe Example



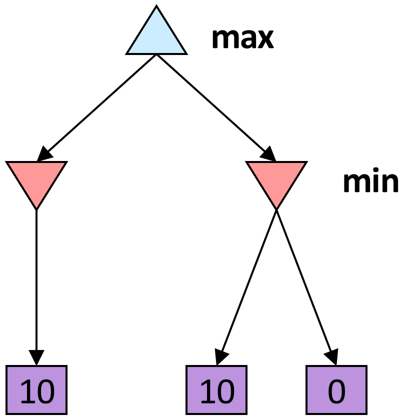
Alpha-Beta Tic-Tac-Toe Example



Discontinue search below a MIN node whose beta value ≤ alpha value of one of its MAX ancestors

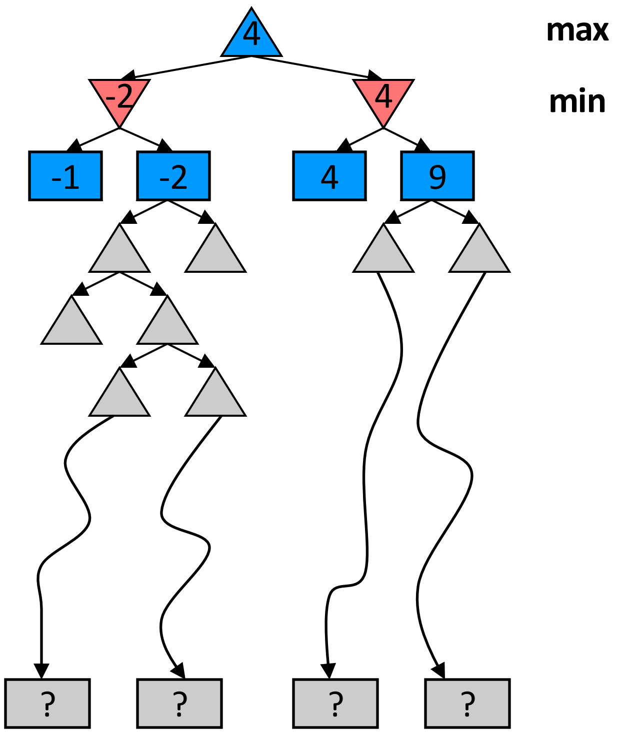
Alpha-Beta Pruning Properties

- This pruning has no effect on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - **Time complexity drops to $O(b^{m/2})$**
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…
- This is a simple example of metareasoning (computing about what to compute)



Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - α-β reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm



Evaluation function

- **Evaluation function** or **static evaluator** used to evaluate the "goodness" of a game position

  Contrast with heuristic search, where evaluation function estimates **cost** from start node to goal passing through given node

- Zero-sum assumption permits **single function to describe goodness of board for both players**
  - **f(n) >> 0**: position n good for me; bad for you
  - **f(n) << 0**: position n bad for me; good for you
  - **f(n) near 0**: position n is a neutral position
  - **f(n) = +infinity**: win for me
  - **f(n) = -infinity**: win for you

Evaluation function examples

- **For Tic-Tac-Toe**

  f(n) = [# my open 3lengths] - [# your open 3lengths]

  Where 3length is complete row, column or diagonal that has no opponent marks

- **Alan Turing's function for chess**
  - **f(n) = w(n)/b(n)** where w(n) = sum of point value of white's pieces and b(n) = sum of black's
  - Traditional piece values: pawn:1; knight:3; bishop:3; rook:5; queen:9

Evaluation function examples

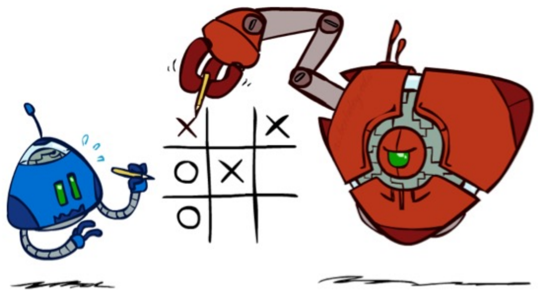
- Most evaluation functions specified as a weighted sum of positive features

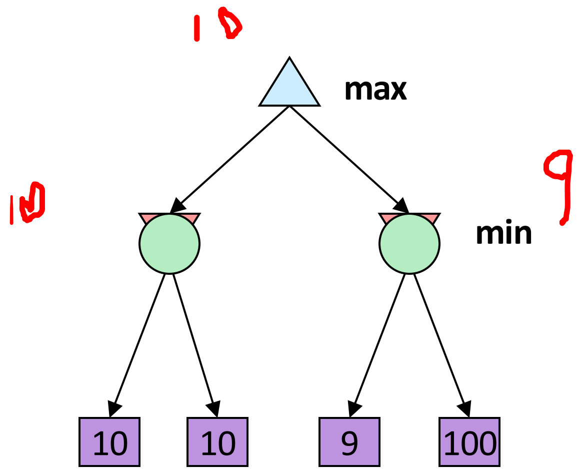
  $f(n) = w_1 * feat_1(n) + w_2 * feat_2(n) + ... + w_n * feat_k(n)$
- Example chess features are piece count, piece values, piece placement, squares controlled, etc.
- IBM's chess program Deep Blue (circa 1996) had **>8K features** in its evaluation function

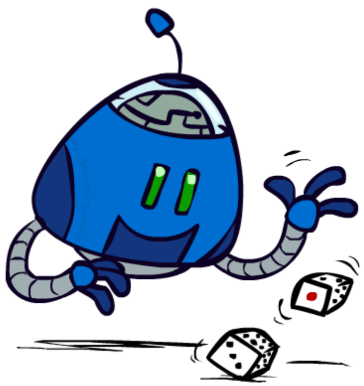
Depth Matters

- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation

Worst-Case vs. Average Case







Idea: Uncertain outcomes controlled by chance, not an adversary!

Stochastic Games

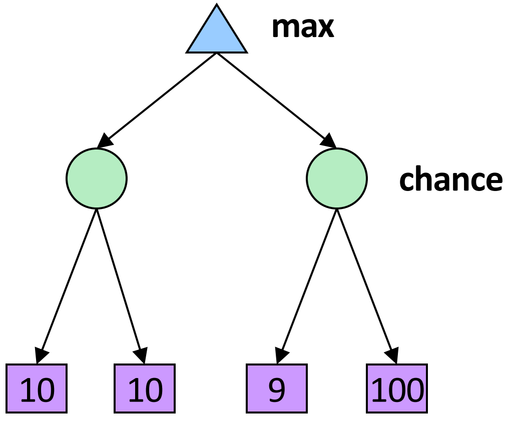
- In real life, unpredictable external events can put us into unforeseen situations
- Many games introduce unpredictability through a random element, such as the throwing of dice
- These offer simple scenarios for problem solving with adversaries and uncertainty

Why can’t we use MiniMax?

- Before a player chooses a move, she rolls dice and only then knows exactly what moves are possible
- The immediate outcome of each move is also known
- But she does not know what moves she or her opponent will have available in the future
- Need to adapt MiniMax to handle this

Expectimax Search

- Why wouldn't we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Actions can fail: when moving a robot, wheels might slip
- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes
- Expectimax search: compute the average score under optimal play
  - Max nodes as in minimax search
  - Chance nodes are like min nodes but the outcome is uncertain
  - Calculate their expected utilities
  - I.e. take weighted average (expectation) of children
- Later, we'll learn how to formalize the underlying uncertain-result problems as Markov Decision Processes



Expectimax Pseudocode

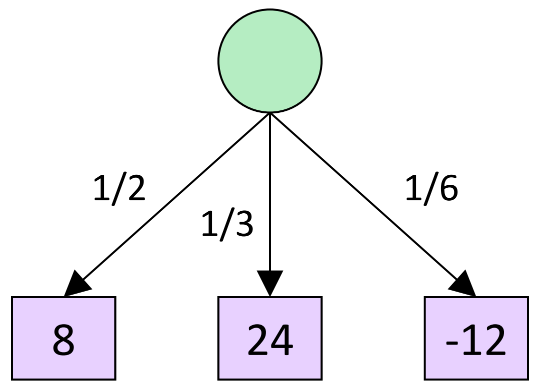
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

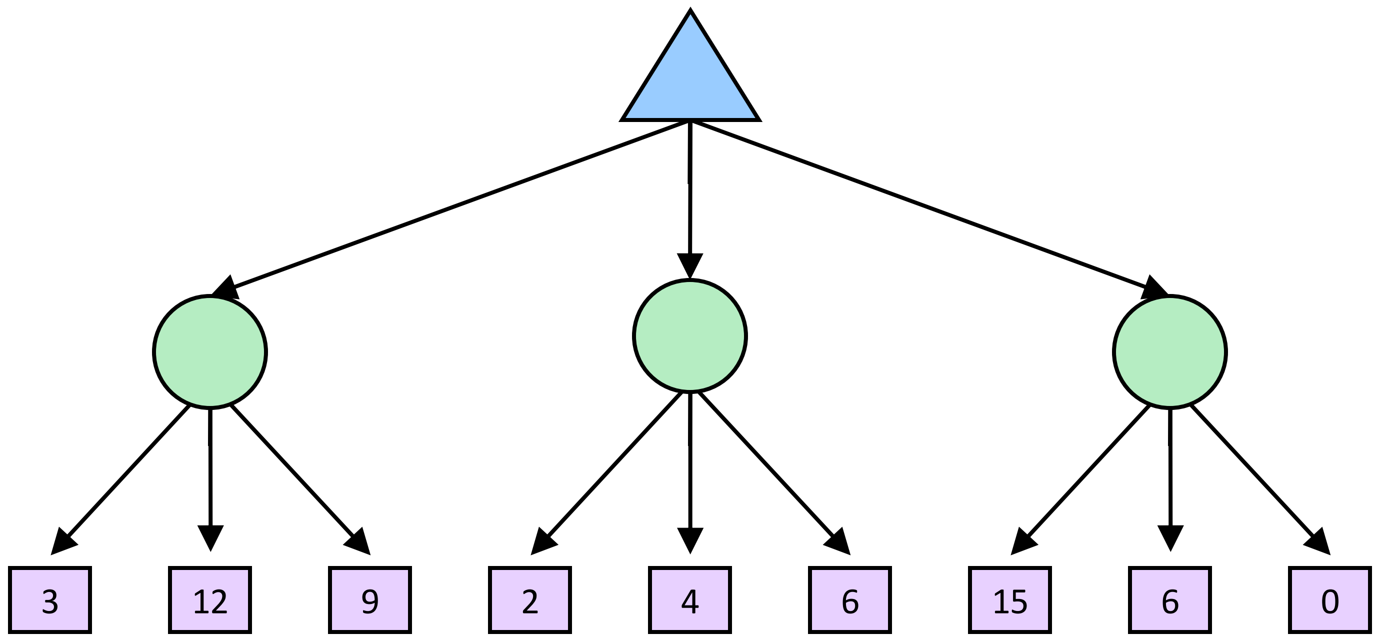
Expectimax Pseudocode

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```



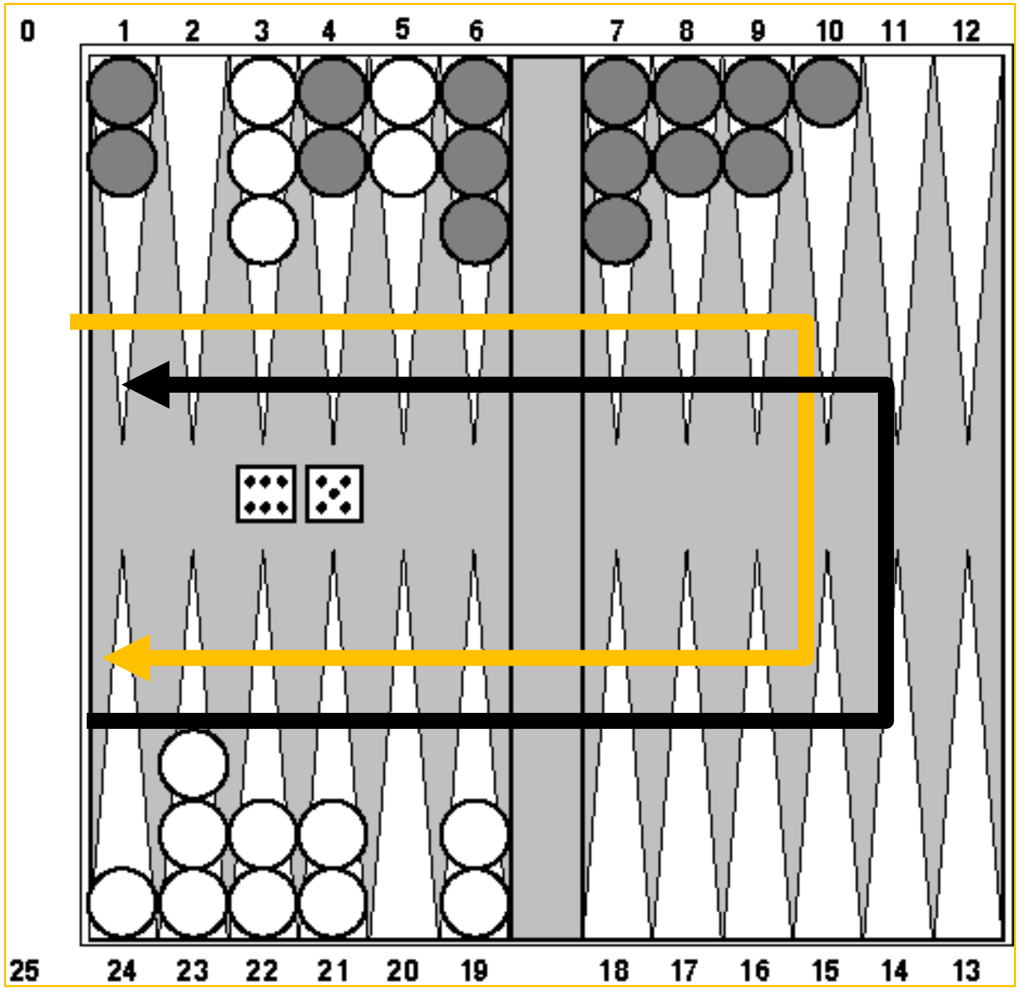
v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10

Expectimax Example

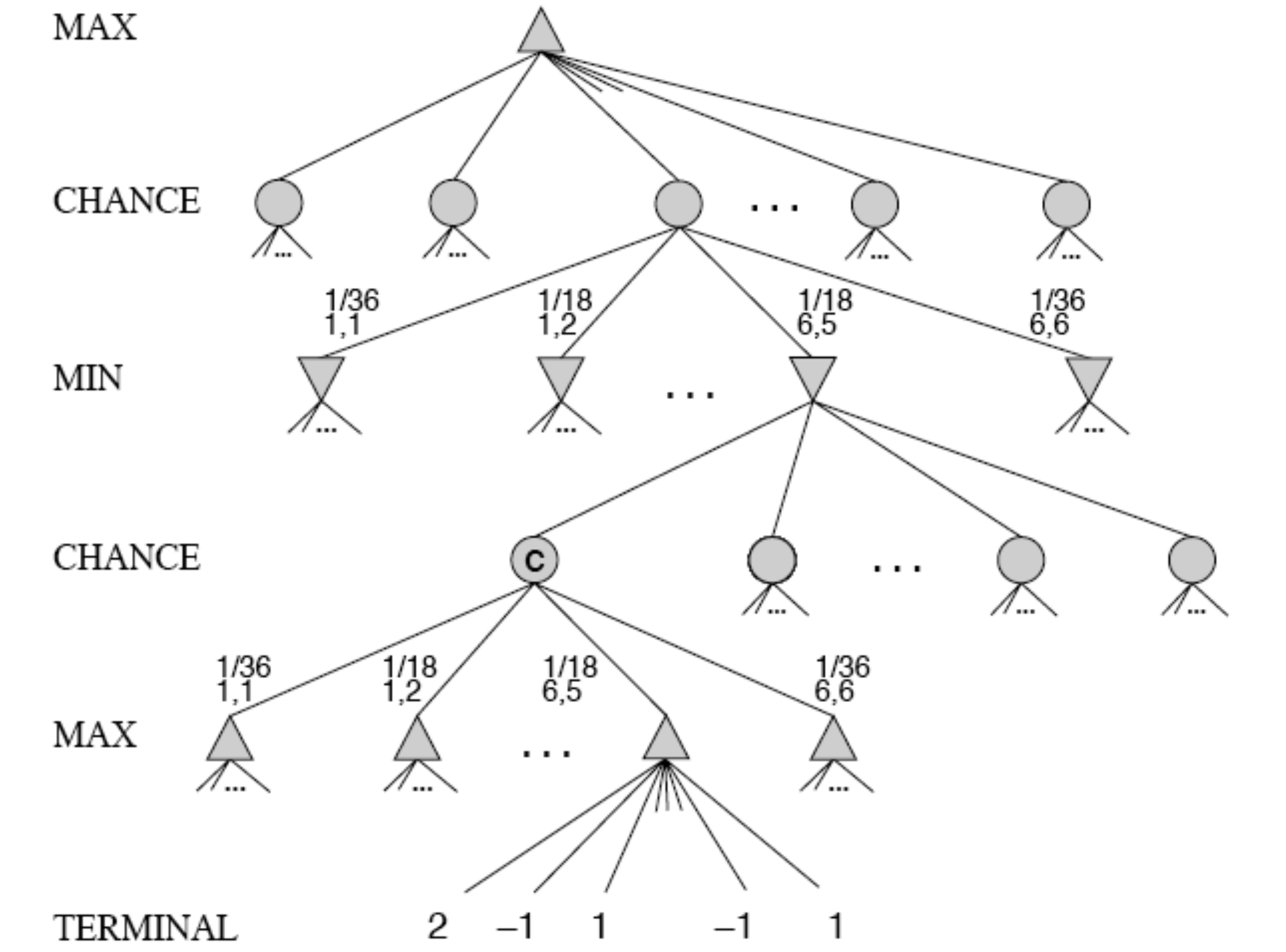


Example: [Backgammon]()

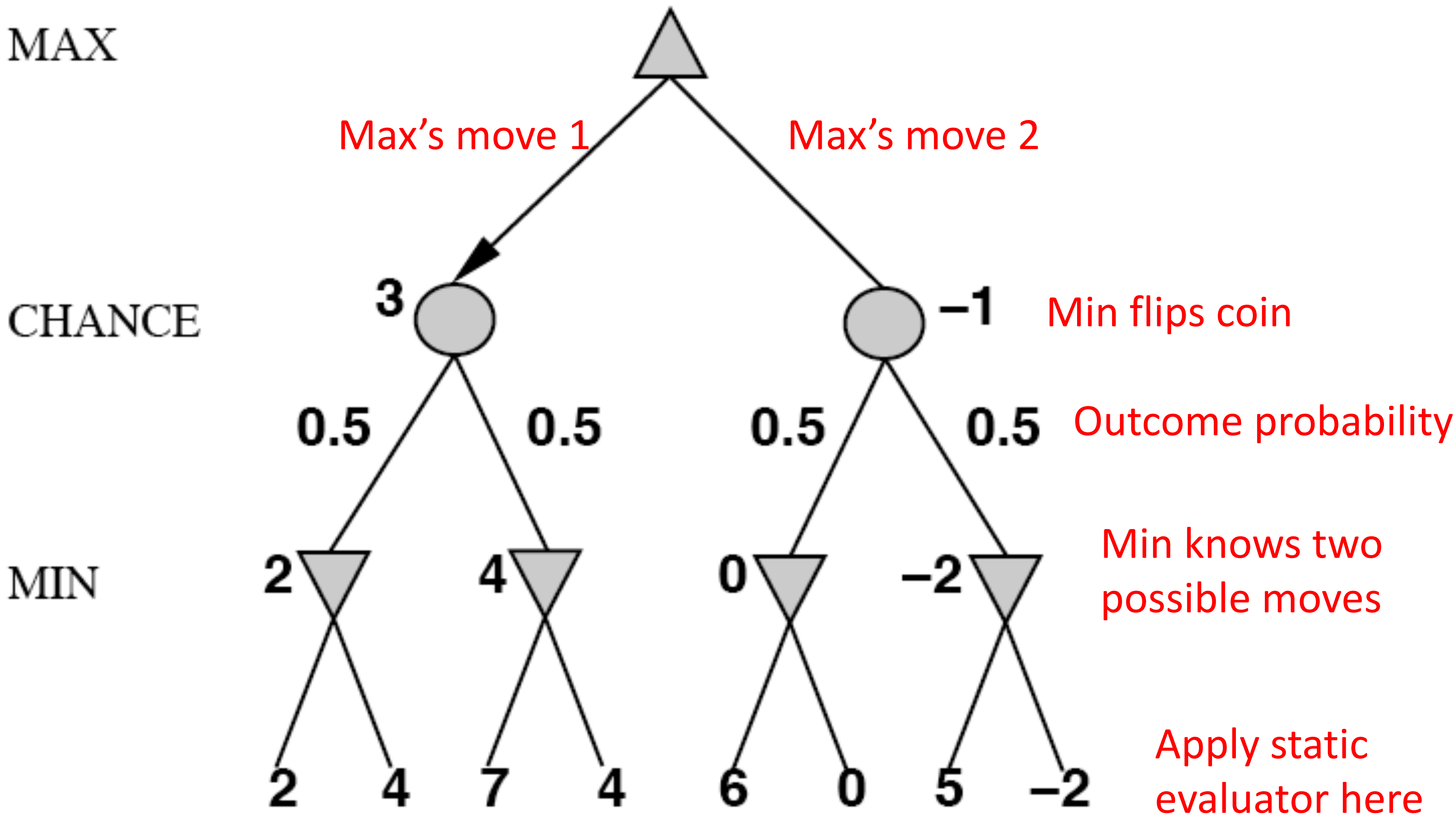
- Popular two-player game with uncertainty
- Players roll dice to determine what moves can be made
- White has just rolled 5 & 6, giving four legal moves:
  - 5-10, 5-11
  - 5-11, 19-24
  - 5-10, 10-16
  - 5-11, 11-16
- Good for exploring decision making in adversarial prob-lems involving skill **and** luck



MiniMax trees with Chance Nodes



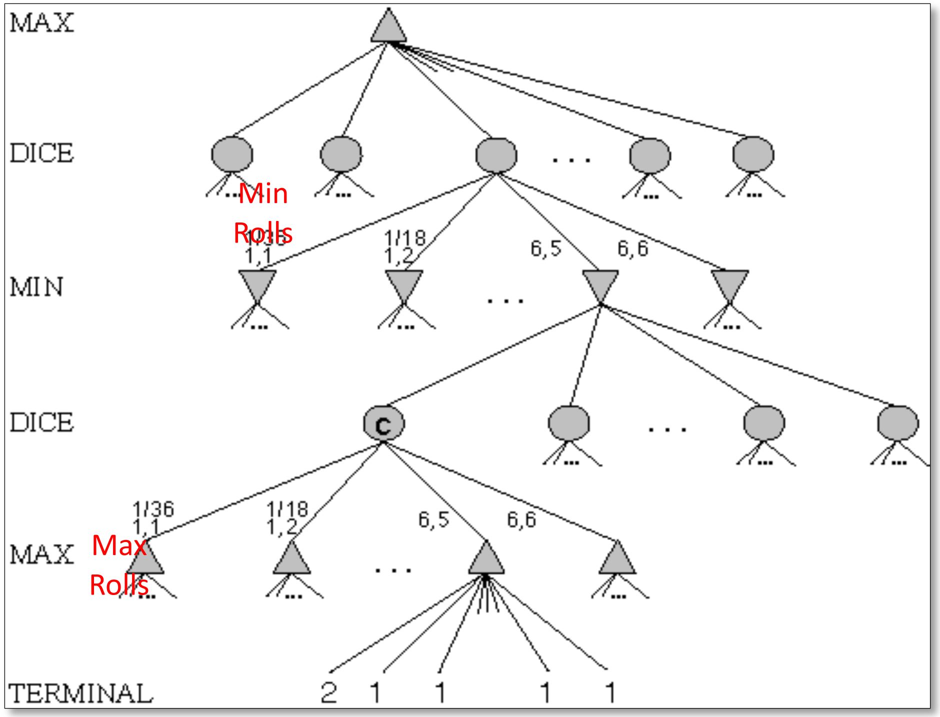
Understanding the notation



Board state includes chance outcome determining available moves

Game trees with chance nodes

- Chance nodes (circles) represent random events
- For random event with N outcomes, chance node has N children, each with a probability
- 2 dice: 21 distinct outcomes
- Use minimax to compute values for MAX and MIN nodes
- Use expected values for chance nodes

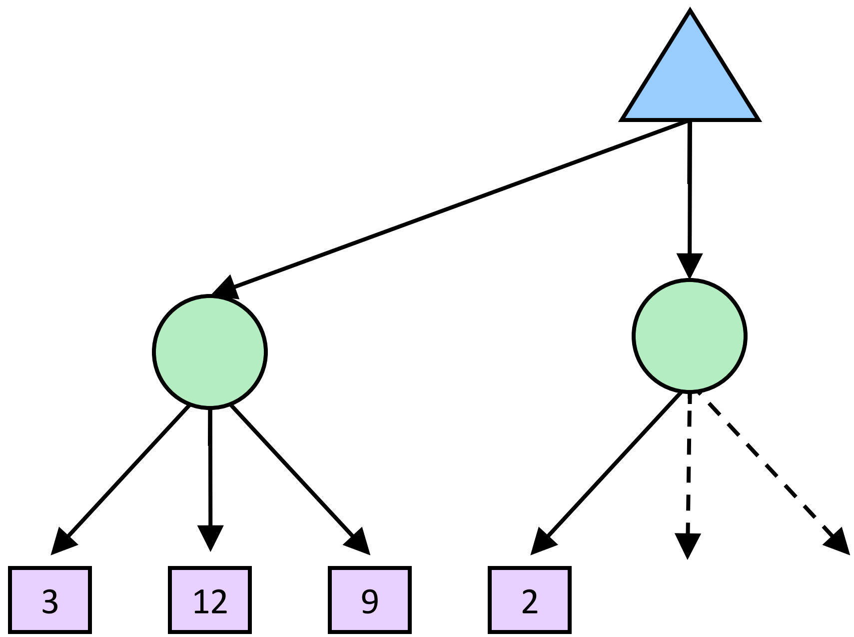


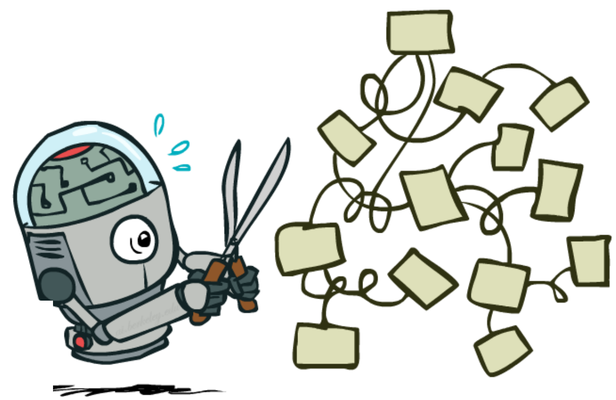
- Chance nodes over max node: **expectimax(C)** = $\sum_i(P(d_i)*\text{maxval}(i))$
- Chance nodes over min node: **expectimin(C)** = $\sum_i(P(d_i)*\text{minval}(i))$

Impact on lookahead

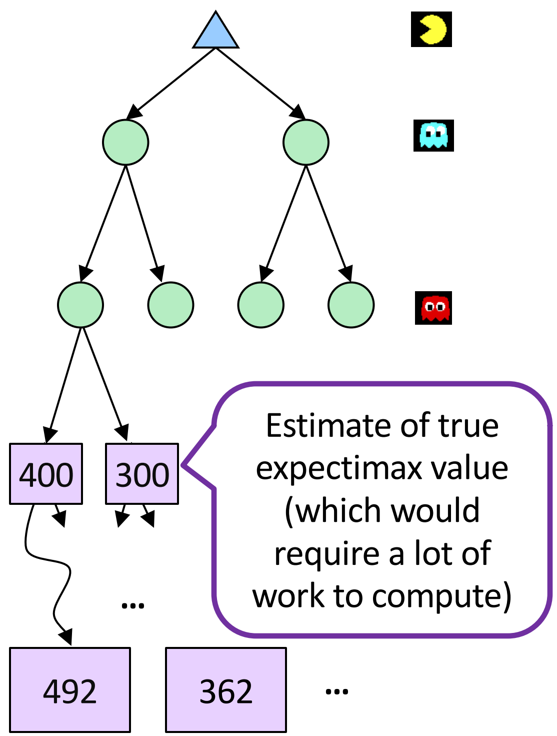
- Dice rolls **increase branching factor**
  – There are 21 possible rolls with two dice
- Backgammon: ~20 legal moves for given roll
  ~6K with 1-1 roll (get to roll again!)
- At depth 4: 20 * (21 * 20)**3 ≈ 1.2B boards
- As depth increases, probability of reaching a given node shrinks
  – lookahead's value diminished and alpha-beta pruning is much less effective
- TDGammon used depth-2 search + good static evaluator + Reinforcement Learning to achieve world-champion level

Expectimax Pruning?

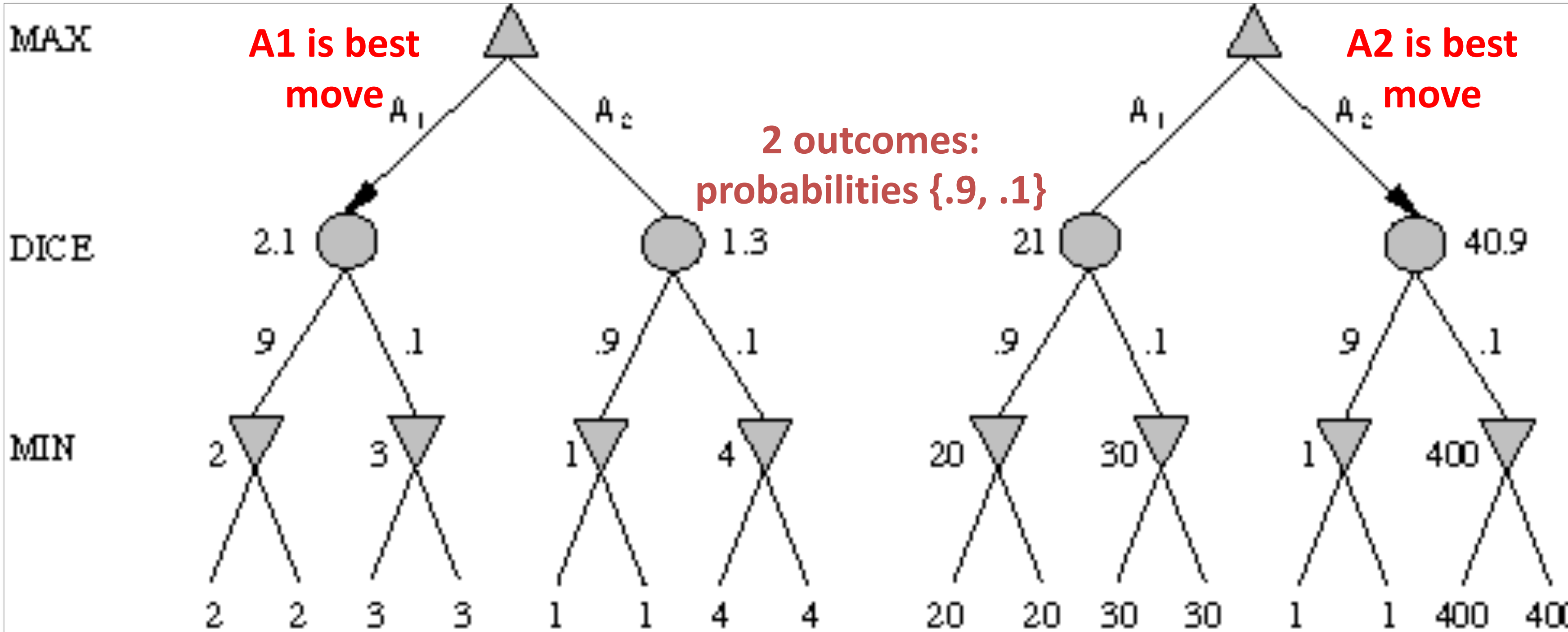




Depth-Limited Expectimax



Meaning of the evaluation function



- With probabilities & expected values we must be careful about meaning of values returned by static evaluator
- Relative-order preserving change of static evaluation values doesn't change minimax decision, but could here
- Linear transformations are OK

Games of imperfect information

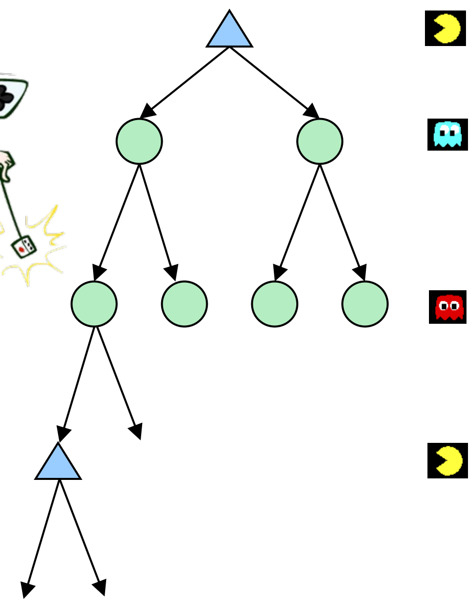
- E.g. card games where opponent's initial hand unknown
  - **Can calculate probability for each possible deal**
  - Like having one big dice roll at beginning of game
- Possible approach: minimax over each action in each deal; choose action with highest expected value over all deals
- Special case: if action optimal for all deals, it's optimal
- GIB bridge program, approximates this idea by

1. Generating 100 deals consistent with bidding
2. Picking action that wins most tricks on average

What Probabilities to Use?

- In expectimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state
  - Model could be a simple uniform distribution (roll a die)
  - Model could be sophisticated and require a great deal of computation
  - We have a chance node for any outcome out of our control: opponent or environment
  - The model might say that adversarial actions are likely!

- For now, assume each chance node magically comes along with probabilities that specify the distribution over its outcomes



Having a probabilistic belief about another agent's action does not mean that the agent is flipping any coins!

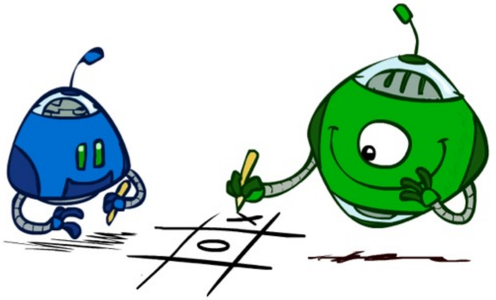
High-Performance Game Programs

- Many programs based on alpha-beta + iterative deepening + extended/singular search + transposition tables + huge databases + ...
- Chinook searched all checkers configurations with ≤ 8 pieces to create endgame database of 444 billion board configurations
- Methods general, but implementations improved via many specifically tuned-up enhancements (e.g., the evaluation functions)

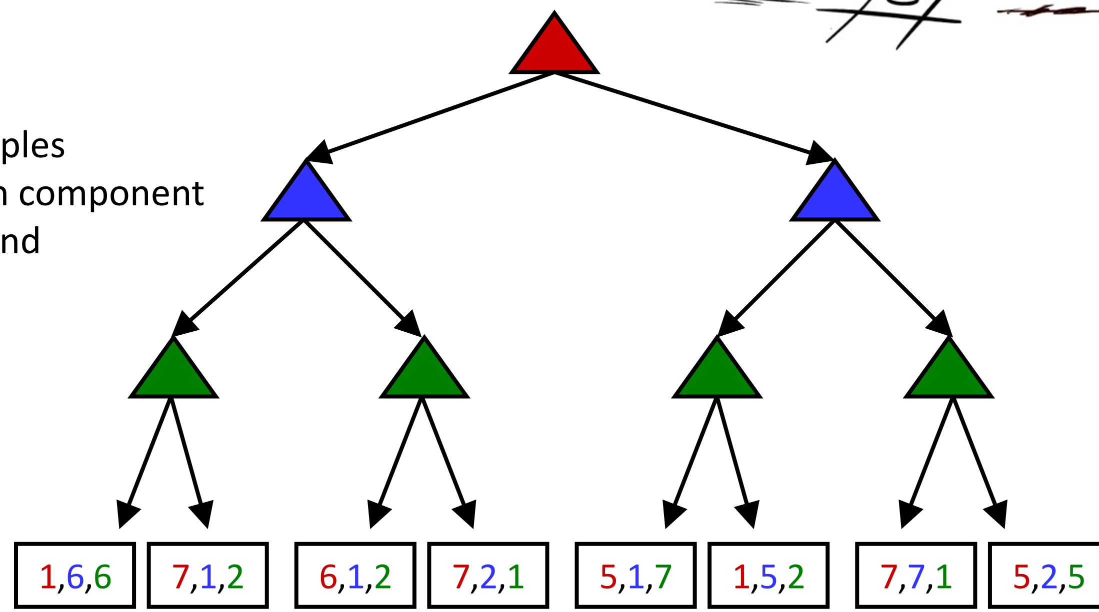
Other Issues

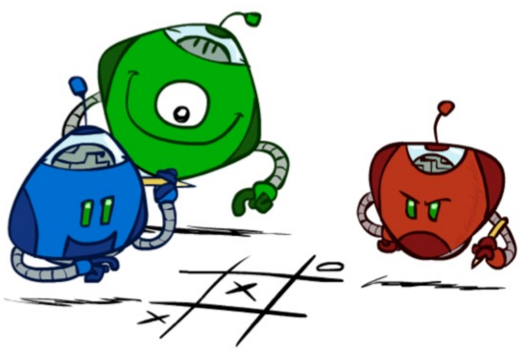
- Multi-player games, no alliances
  - E.g., many card games, like Hearts
- Multi-player games with alliances
  - E.g., Risk
  - More on this when we discuss game theory
  - Good model for a social animal like humans, where we must balance cooperation and competition

Multi-Agent Utilities



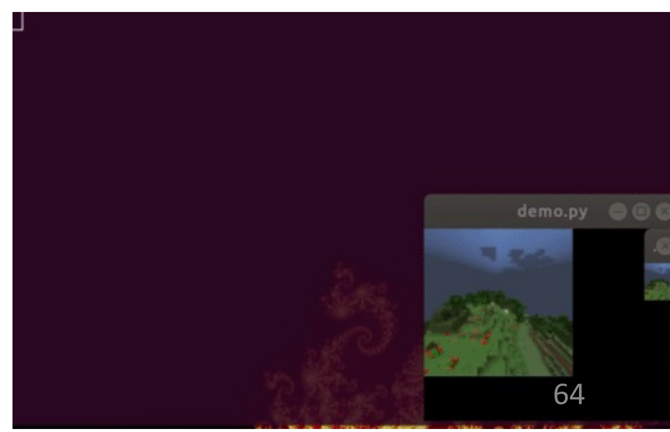
- What if the game is not zero-sum, or has multiple players?
- Generalization of minimax:
  - Terminals have utility tuples
  - Node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to cooperation and competition dynamically…





AI and video Games

- Many games include agents run by the game program as
  - Adversaries, in first person shooter games
  - Collaborators, in a virtual reality game
  - E.g.: AI bots in Fortnite Chapter 2
- Some games used as AI/ML challenges or learning environments
  - MineRL: train bots to play Minecraft
  - MarioAI: train bots for Super Mario Bros



AlphaGO

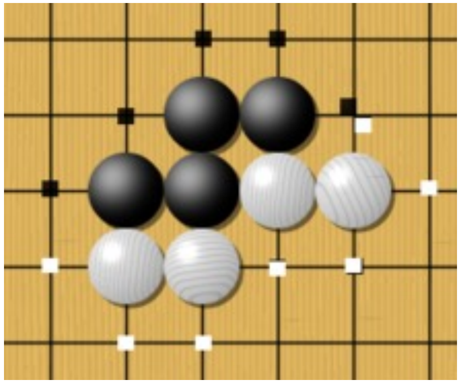
- Developed by Google's DeepMind
- Beat top-ranked human grandmasters in 2016
- Used Monte Carlo tree search over game tree

  expands search tree via random sampling of search space
- *Science* Breakthrough of the year runner-up

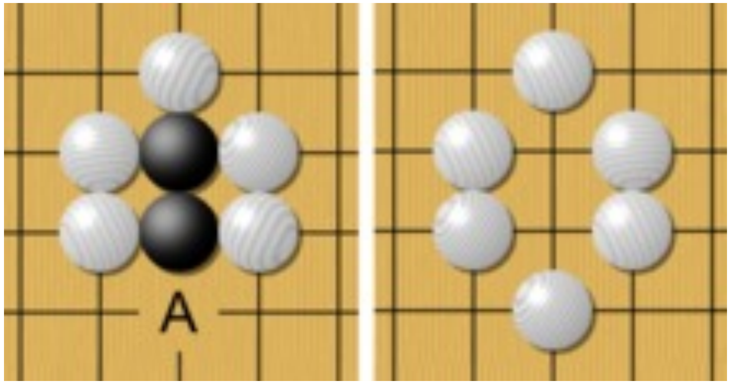
  Mastering the game of Go with deep neural networks and tree search, Silver et al., *Nature*, 529:484–489, 2016
- Match with grandmaster Lee Sedol in 2016 was subject of award-winning 2017 AlphaGo

Go - the game

- Played on 19x19 board; black vs. white stones
- Huge state space $O(b^d)$: chess:~35^{80}, go: ~250^{150}
- Rule: Stones on board must have an adjacent open point ("liberty") or be part of connected group with a liberty. Groups of stones losing their last liberty are removed from the board.



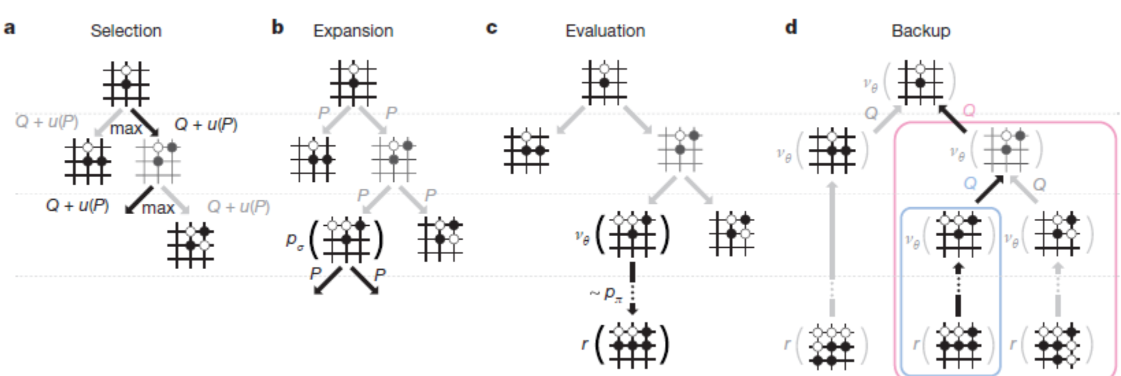
liberties



capture

AlphaGo implementation

- Trained deep neural networks (13 layers) to learn **value function** and **policy function**
- Performs Monte Carlo game search
  - –explore state space like minimax
  - –random "rollouts"
  - –simulate probable plays by opponent according to policy function



AlphaGo implementation

- Hardware: 1920 CPUs, 28O GPUs
- Neural networks trained in two phases over 4-6 weeks
- **Phase 1:** supervised learning from database of 30 million moves in games between two good human players
- **Phase 2:** play against versions of self using reinforcement learning to improve performance