Alpha-Beta Search

Two-player games

- The object of a search is to find a path from the starting position to a goal position
- In a *puzzle-type* problem, you (the searcher) get to choose every move
- In a two-player competitive game, you alternate moves with the other player
- The other player doesn't want to reach *your* goal
- Your search technique must be very different

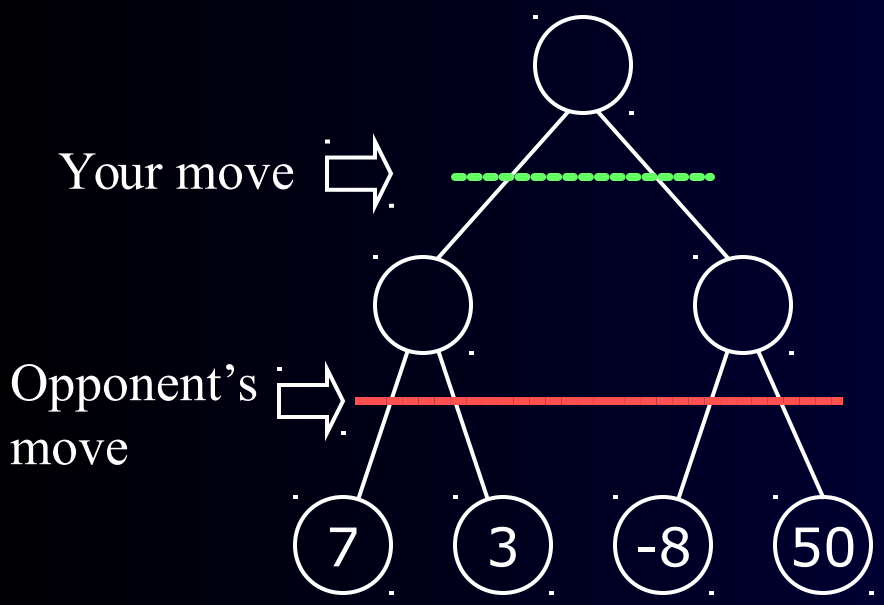
Payoffs

- Each game outcome has a *payoff,* which we can represent as a number
- By convention, we prefer positive numbers
- In some games, the outcome is either a simple win (+1) or a simple loss (-1)
- In some games, you might also *tie,* or *draw* (0)
- In other games, outcomes may be other numbers (say, the amount of money you win at Poker)

Zero-sum games

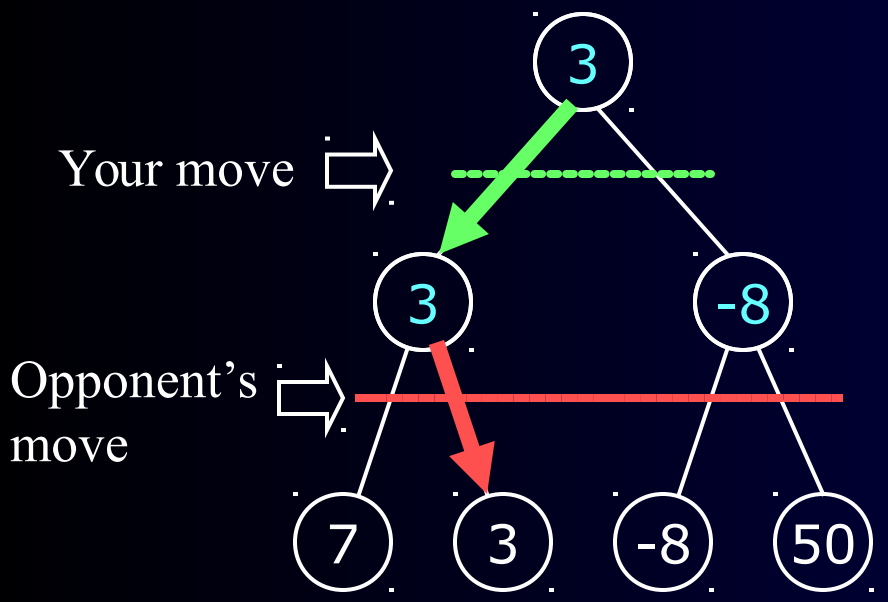
- Most common games are zero-sum: What I win ($12), plus what you win (-$12), equals zero
- Not all games are zero-sum games
- For simplicity, we consider only zero-sum games
- From our point of view, positive numbers are good, negative numbers are bad
- From our opponents point of view, positive numbers are bad, negative numbers are good

A trivial "game"



- Wouldn't you like to win 50?
- Do you think you will?
- Where should you move?

Minimaxing



- Your opponent will choose smaller numbers
- If you move left, your opponent will choose 3
- If you move right, your opponent will choose -8
- Therefore, your choices are really 3 or -8
- You should move left, and play will be as shown

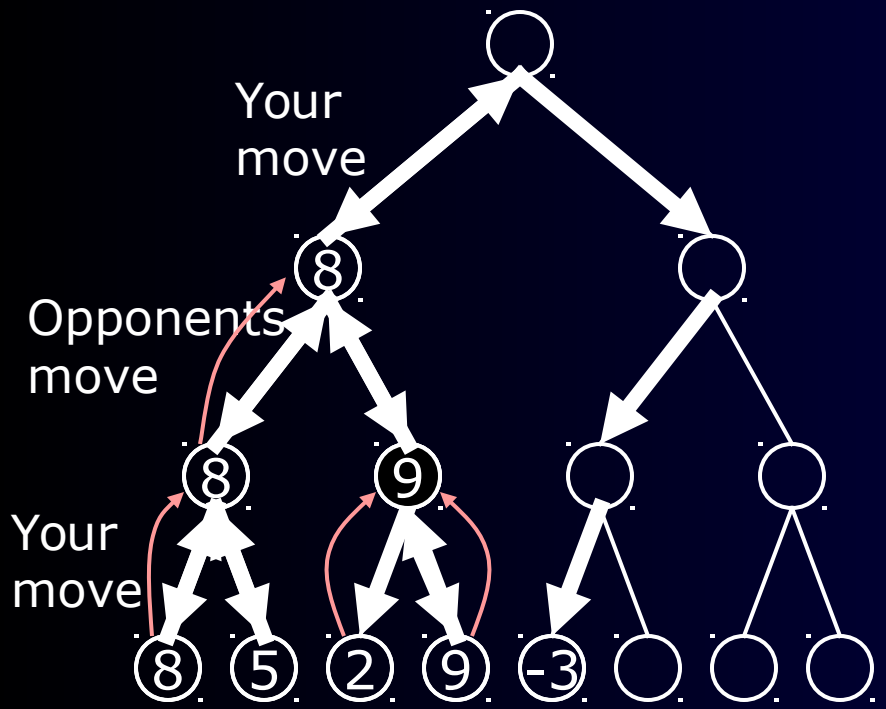
Heuristics

- In a large game, you don't really know the payoffs
- A heuristic function computes, for a given node, your best guess as to what the payoff will be
- The heuristic function uses whatever knowledge you can build into the program
- We make two key assumptions:
  - Your opponent uses the same heuristic function as you do
  - The more moves ahead you look, the better your heuristic function will work

PBVs

- A PBV is a preliminary backed-up value
  - Explore down to a given level using depth-first search
  - As you reach each lowest-level node, evaluate it using your heuristic function
  - Back up values to the next higher node, according to the following rules:
    - If it's your move, bring up the largest value, possibly replacing a smaller value
    - If it's your opponent's move, bring up the smallest value, possible replacing a larger value

Using PBVs (animated)

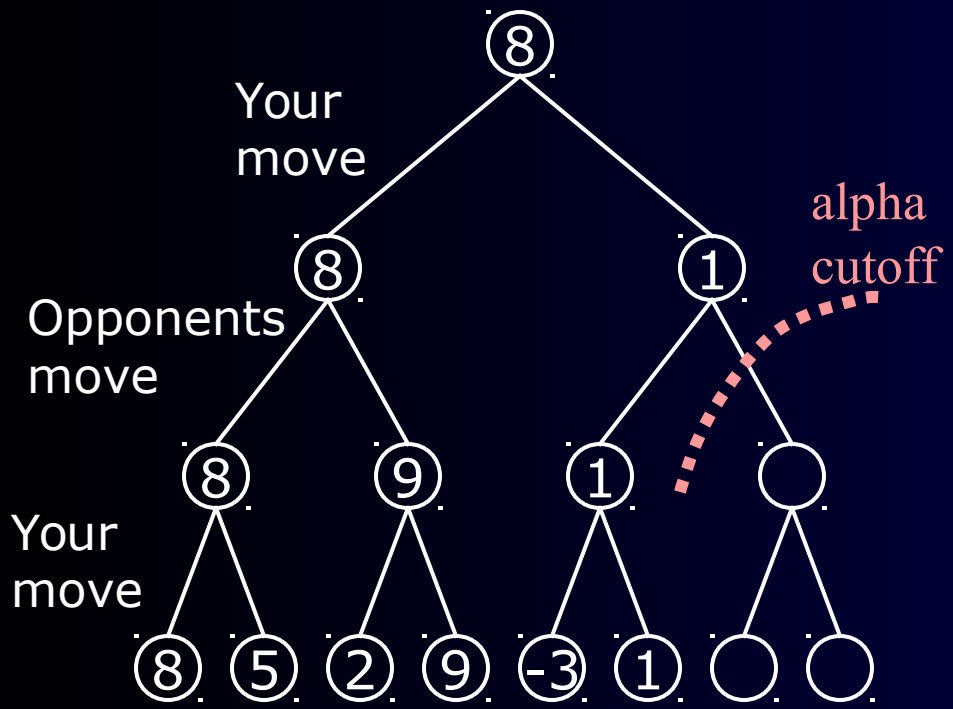


- Do a DFS; find an 8 and bring it up
- Explore 5; smaller than 8, so ignore it
- Backtrack; bring 8 up another level
- Explore 2; bring it up
- Explore 9; better than 2, so bring it up, replacing 2
- 9 is not better than 8 (for your opponent), so ignore it
- Explore –3, bring it up
- Etc.

Bringing up values

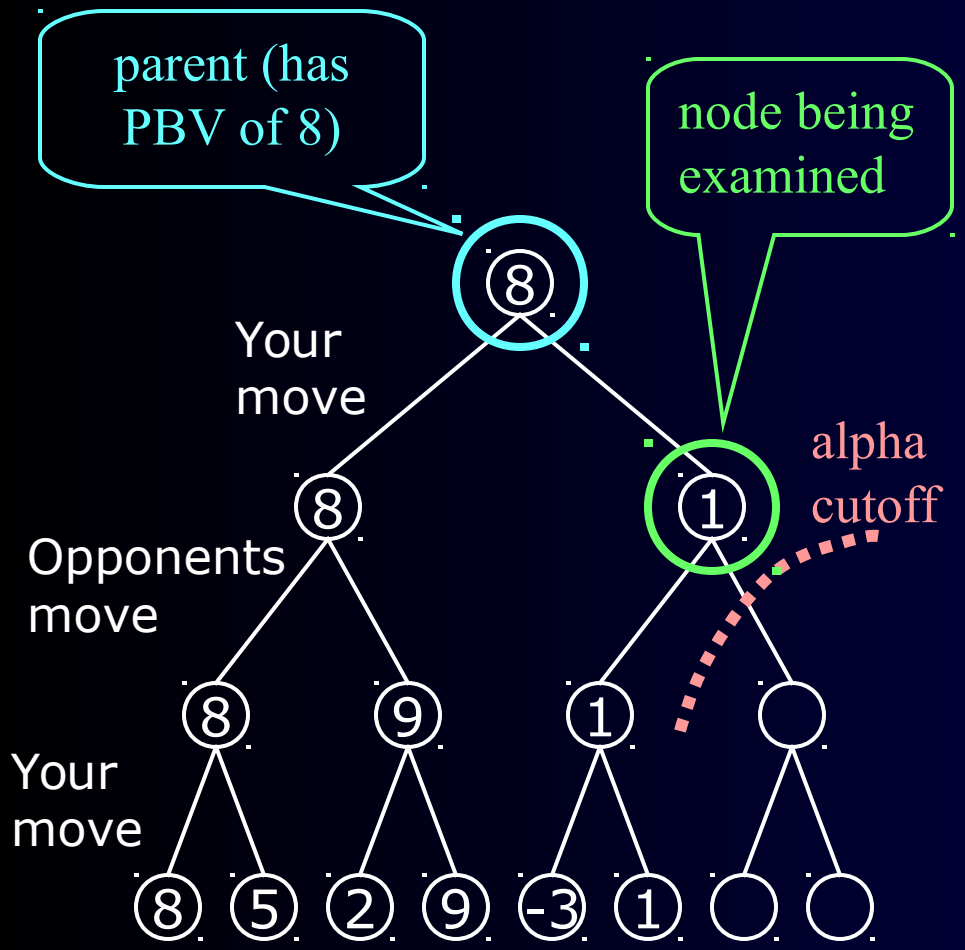
- If it's your move, and the next child of this node has a larger value than this node, replace this value
- If it's your opponent's move, and the next child of this node has a smaller value than this node, replace this value
- At your move, never reduce a value
- At your opponent's move, never increase a value

Alpha cutoffs



- The value at your move is 8 (so far)
- If you move right, the value there is 1 (so far)
- Your opponent will *never increase* the value at this node; it will always be less than 8
- You can *ignore* the remaining nodes

Alpha cutoffs, in more detail



- You have an alpha cutoff when:
  - You are examining a node at which it is your opponent's move, *and*
  - You have a PBV for the node's parent, *and*
  - You have brought up a PBV that is less than the PBV of the node's parent, *and*
  - The node has other children (which we can now "prune")

Beta cutoffs

- An alpha cutoff occurs where
  - It is *your opponent's turn* to move
  - You have computed a PBV for this node's parent
  - The node's parent has a *higher PBV* than this node
  - This node has other children you haven't yet considered
- A beta cutoff occurs where
  - It is *your turn* to move
  - You have computed a PBV for this node's parent
  - The node's parent has a *lower PBV* than this node
  - This node has other children you haven't yet considered

Using beta cutoffs

- Beta cutoffs are harder to understand, because you have to see things from your opponent's point of view
- Your opponent's alpha cutoff is your beta cutoff
- We assume your opponent is rational, and is using a heuristic function similar to yours
- Even if this assumption is incorrect, it's still the best we can do

The importance of cutoffs

- If you can search to the end of the game, you know exactly how to play
- The further ahead you can search, the better
- If you can *prune* (ignore) large parts of the tree, you can search deeper on the other parts
- Since the number of nodes at each level grows exponentially, the higher you can prune, the better
- You can save exponential time

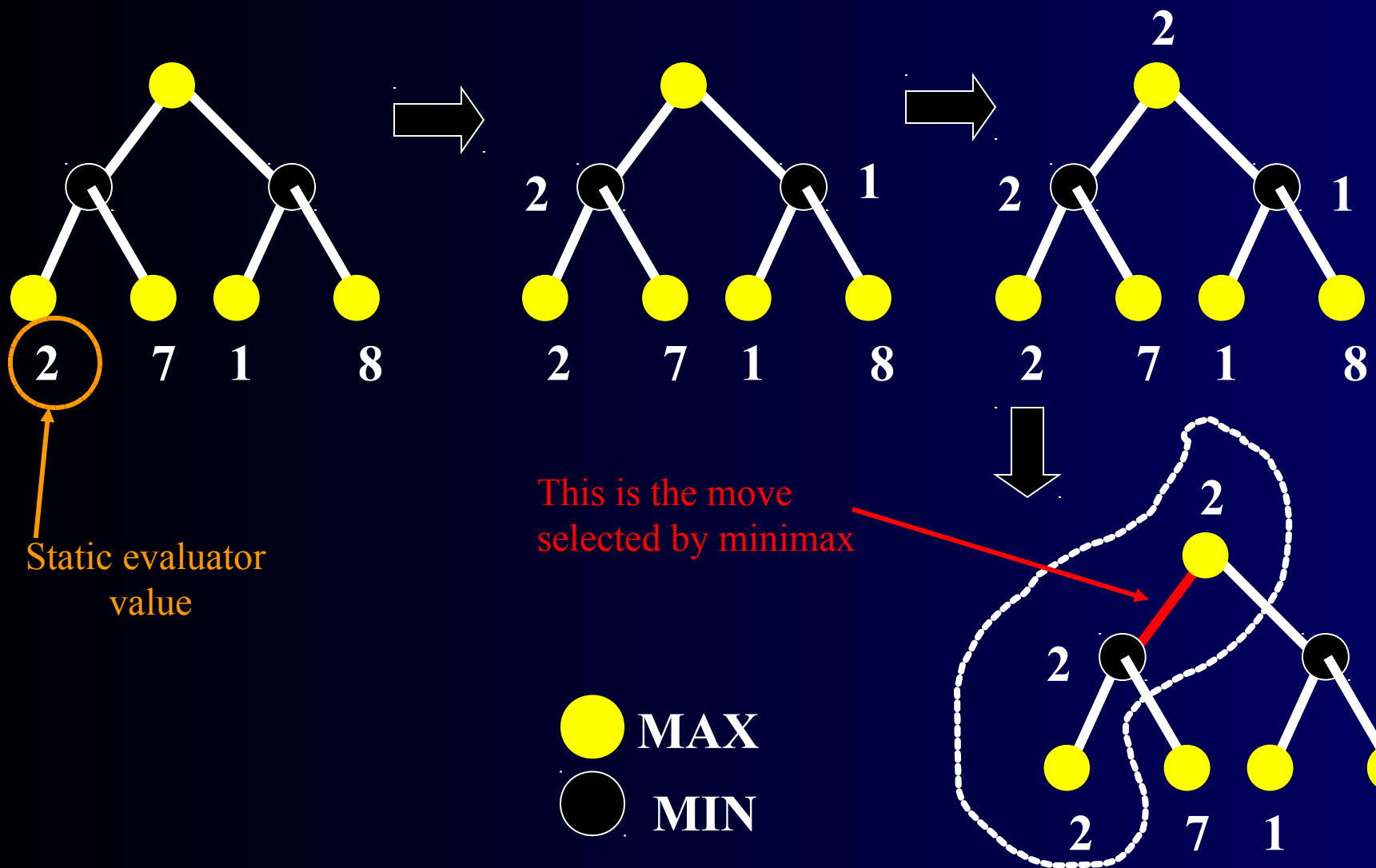
Heuristic alpha-beta searching

- The higher in the search tree you can find a cutoff, the better (because of exponential growth)
- To maximize the number of cutoffs you can make:
  - Apply the heuristic function at each node you come to, not just at the lowest level
  - Explore the "best" moves first
  - "Best" means best for the player *whose move it is at that node*

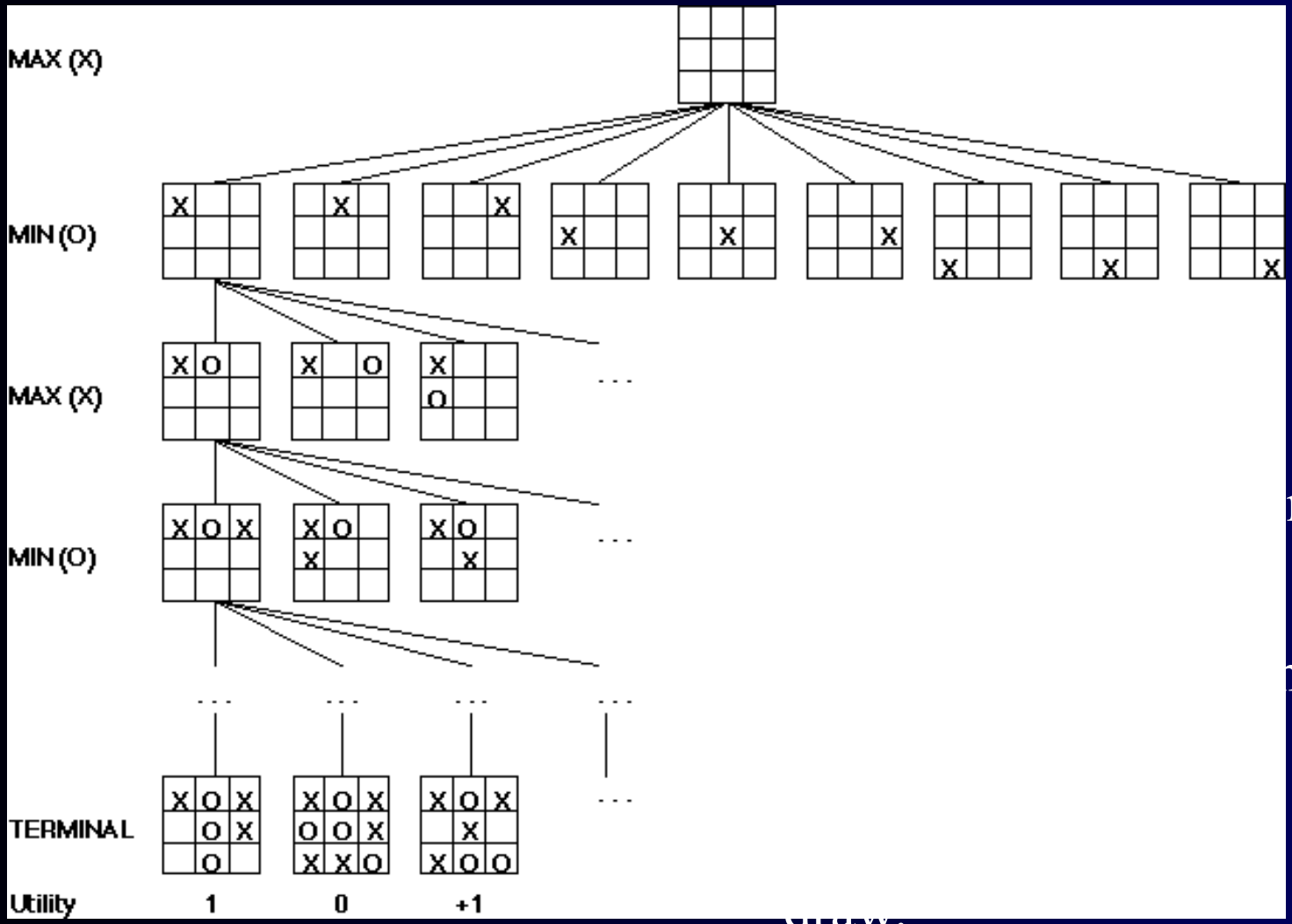
Best game playing strategies

- For any game much more complicated than tic-tac-toe, you have a time limit
- Searching takes time; you need to use heuristics to minimize the number of nodes you search
- But complex heuristics take time, reducing the number of nodes you can search
- Seek a balance between simple (but fast) heuristics, and slow (but good) heuristics

Minimax Algorithm



Partial Game Tree for Tic-Tac-Toe



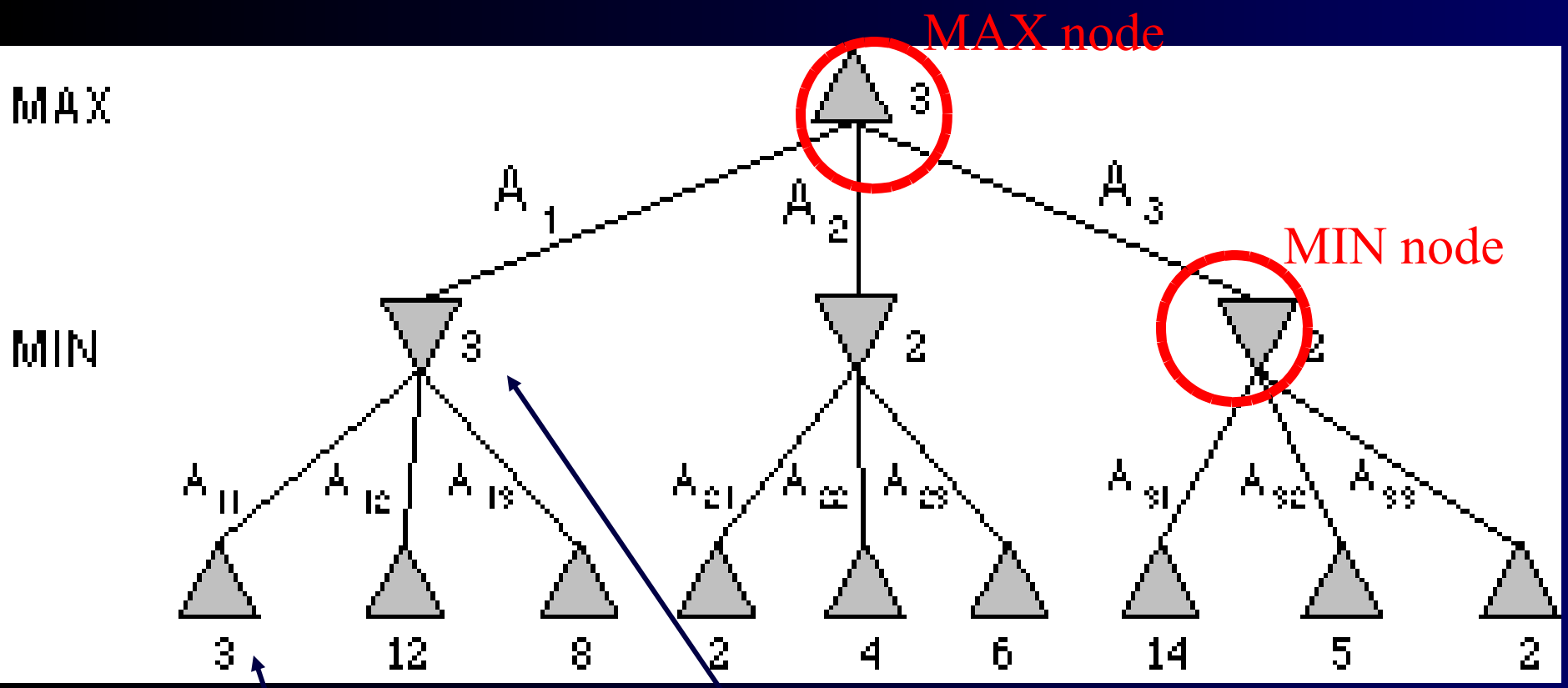
n is a

n is a

is a

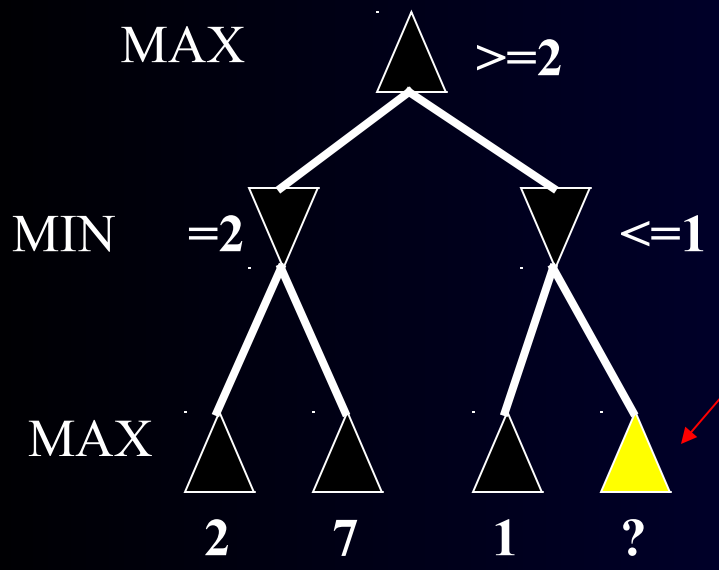
draw.

Minimax Tree



Alpha-beta pruning

- We can improve on the performance of the minimax algorithm through
- Basic idea: *"If you have an idea that is surely bad, don't take the time to see how truly awful it is."* -- Pat Winston

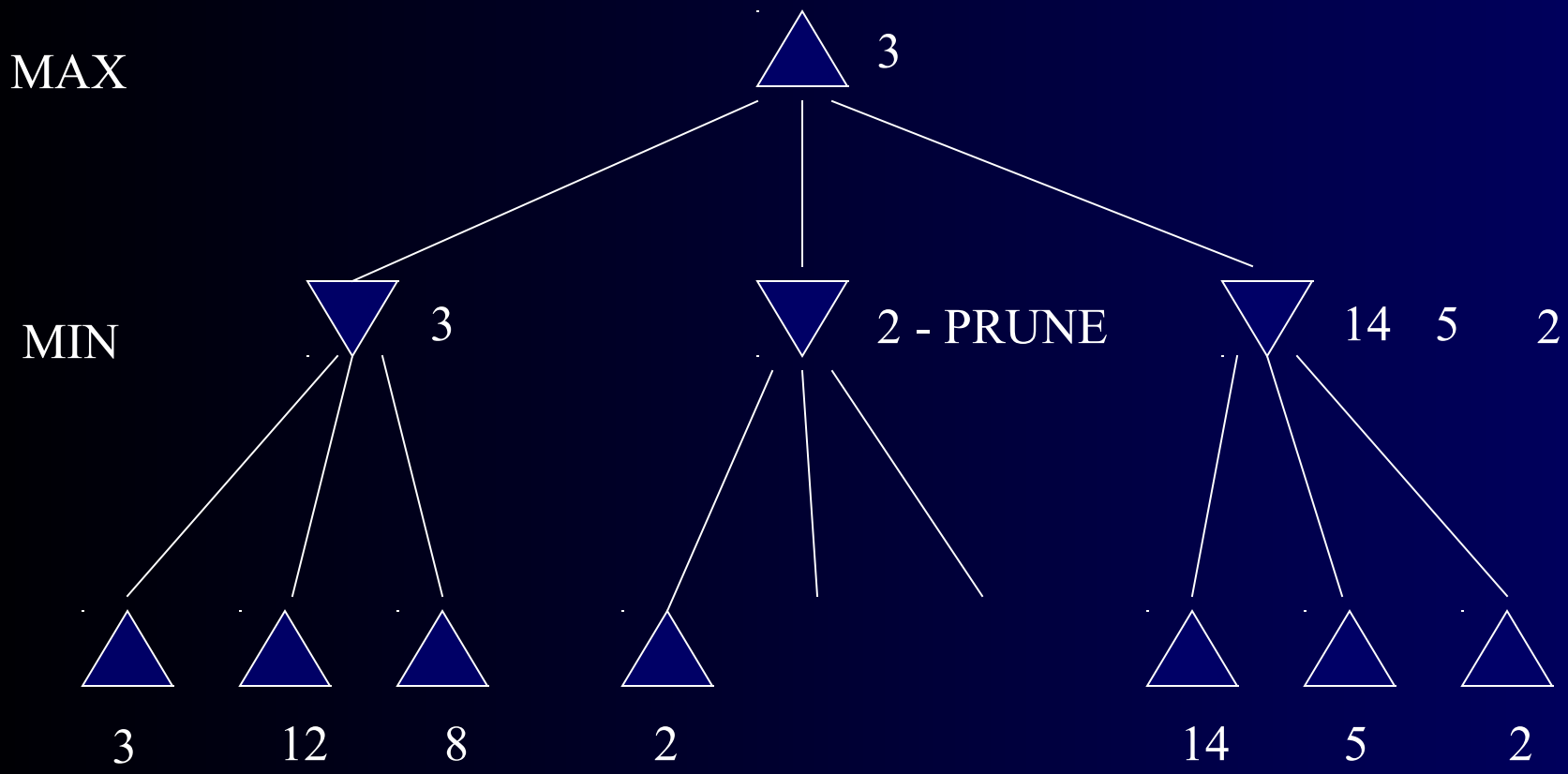


- We don't need to compute the value at this node.
- No matter what it is, it can't affect the value of the root node.

Alpha-beta pruning

- Traverse the search tree in depth-first order
- At each **MAX** node n, **alpha(n)** = maximum value found so far
- At each **MIN** node n, **beta(n)** = minimum value found so far
  - Note: The alpha values start at -infinity and only increase, while beta values start at +infinity and only decrease.
- **Beta cutoff**: Given a MAX node n, cut off the search below n (i.e., don't generate or examine any more of n's children) if alpha(n) >= beta(i) for some MIN node ancestor i of n.
- **Alpha cutoff:** stop searching below MIN node n if beta(n) <= alpha(i) for some MAX node ancestor i of n.

Alpha-beta example



Alpha-beta algorithm

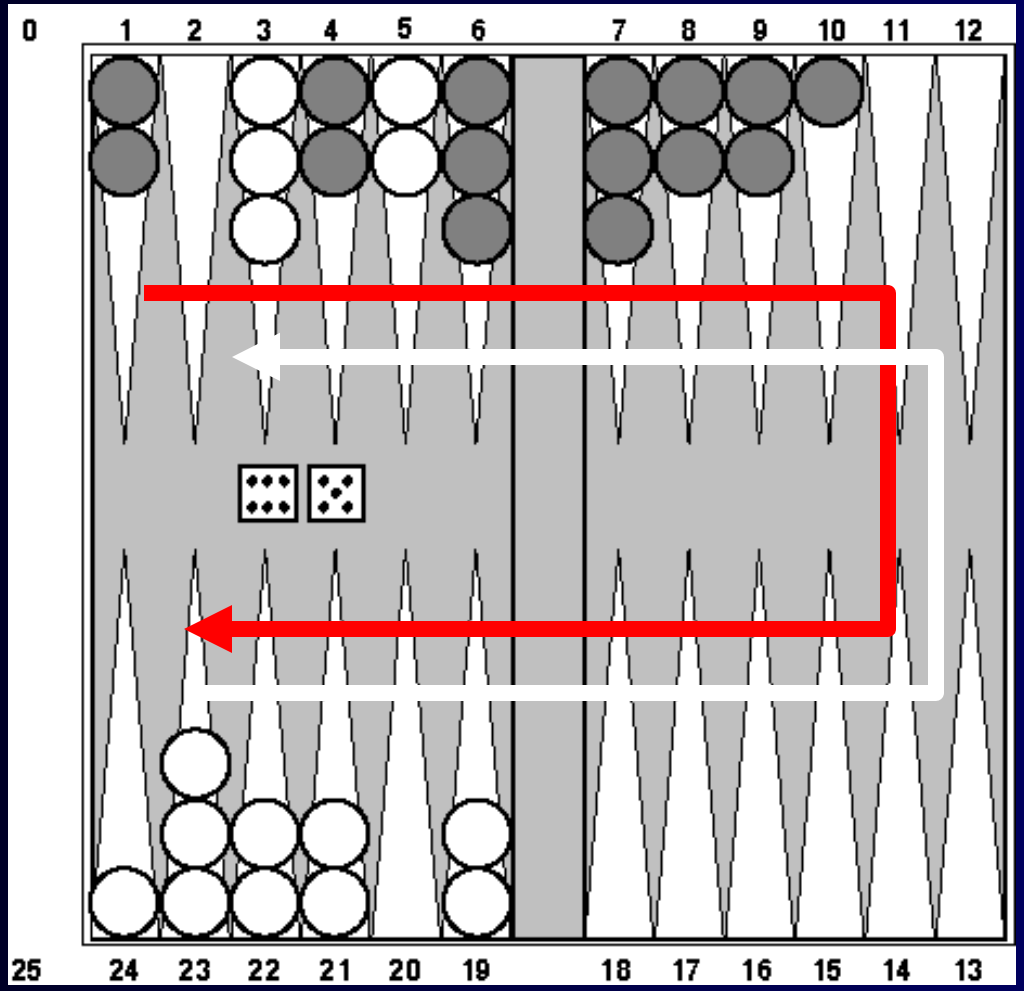
```
function MAX-VALUE (state, game, alpha, beta)
    ;; alpha = best MAX so far; beta = best MIN
if CUTOFF-TEST (state) then return EVAL (state)
for each s in SUCCESSORS (state) do
    alpha := MAX (alpha, MIN-VALUE (state, game,
                  alpha, beta))
    if alpha >= beta then return beta
end
return alpha

function MIN-VALUE (state, game, alpha, beta)
if CUTOFF-TEST (state) then return EVAL (state)
for each s in SUCCESSORS (state) do
    beta := MIN (beta, MAX-VALUE (s, game, alpha, beta))
    if beta <= alpha then return alpha
end
return beta
```

Effectiveness of alpha-beta

- Alpha-beta is guaranteed to compute the same value for the root node as computed by minimax, with less or equal computation
- **Worst case:** no pruning, examining b^d leaf nodes, where each node has b children and a d-ply search is performed
- **Best case:** examine only (2b)^(d/2) leaf nodes.
  - Result is you can search twice as deep as minimax!
- **Best case** is when each player's best move is the first alternative generated
- In Deep Blue, they found empirically that alpha-beta pruning meant that the average branching factor at each node was about 6 instead of about 35!

Games of chance

- Backgammon is a two-player game with **uncertainty**.
- Players roll dice to determine what moves to make.
- White has just rolled *5 and 6* and has four legal moves:
  - 5-10, 5-11
  - 5-11, 19-24
  - 5-10, 10-16
  - 5-11, 11-16
- Such games are good for exploring decision making in adversarial problems involving skill and luck.



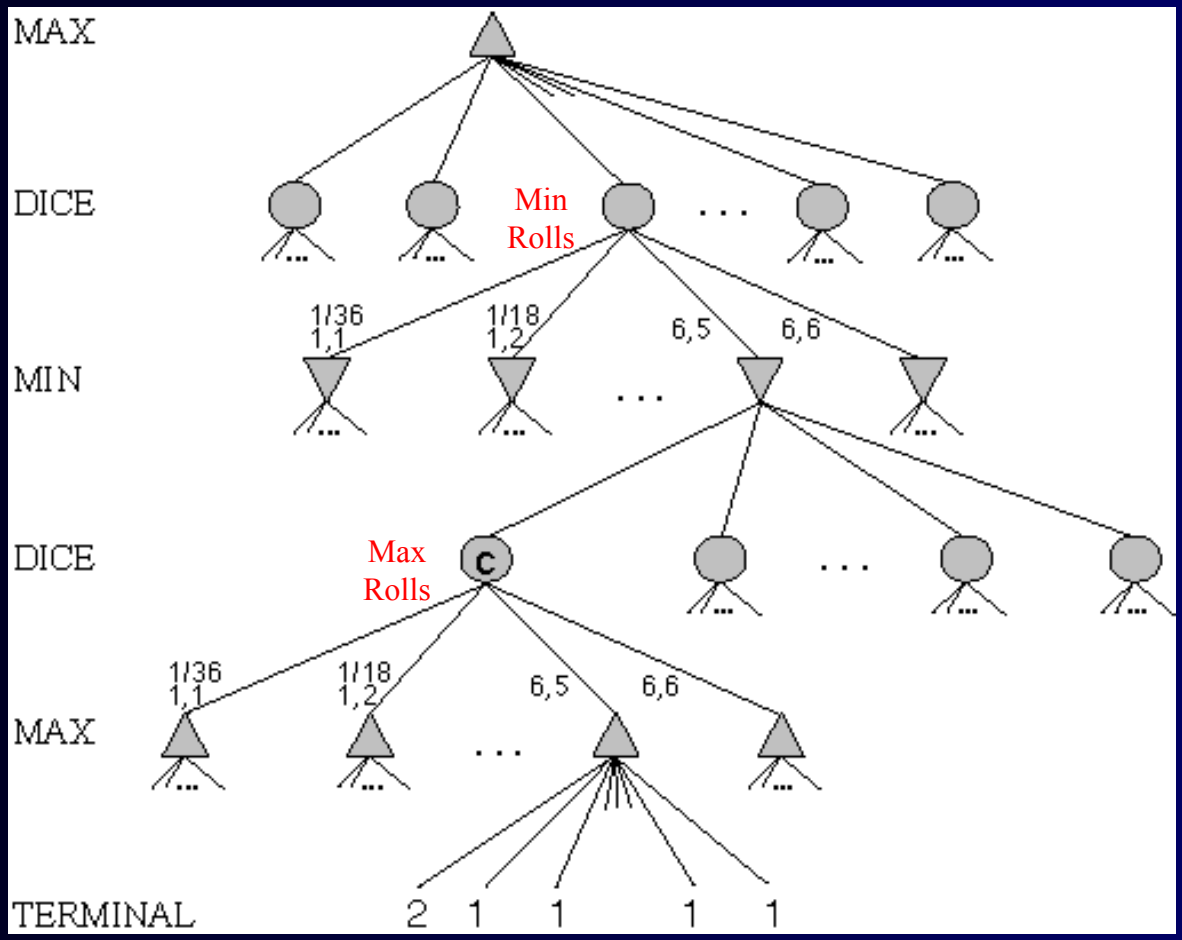
Game Trees with Chance Nodes

- **Chance nodes** (shown as circles) represent random events
- For a random event with N outcomes, each chance node has N distinct children; a probability is associated with each
- (For 2 dice, there are 21 distinct outcomes)
- Use minimax to compute values for MAX and MIN nodes
- Use **expected values** for chance nodes
- For chance nodes over a max node, as in C:

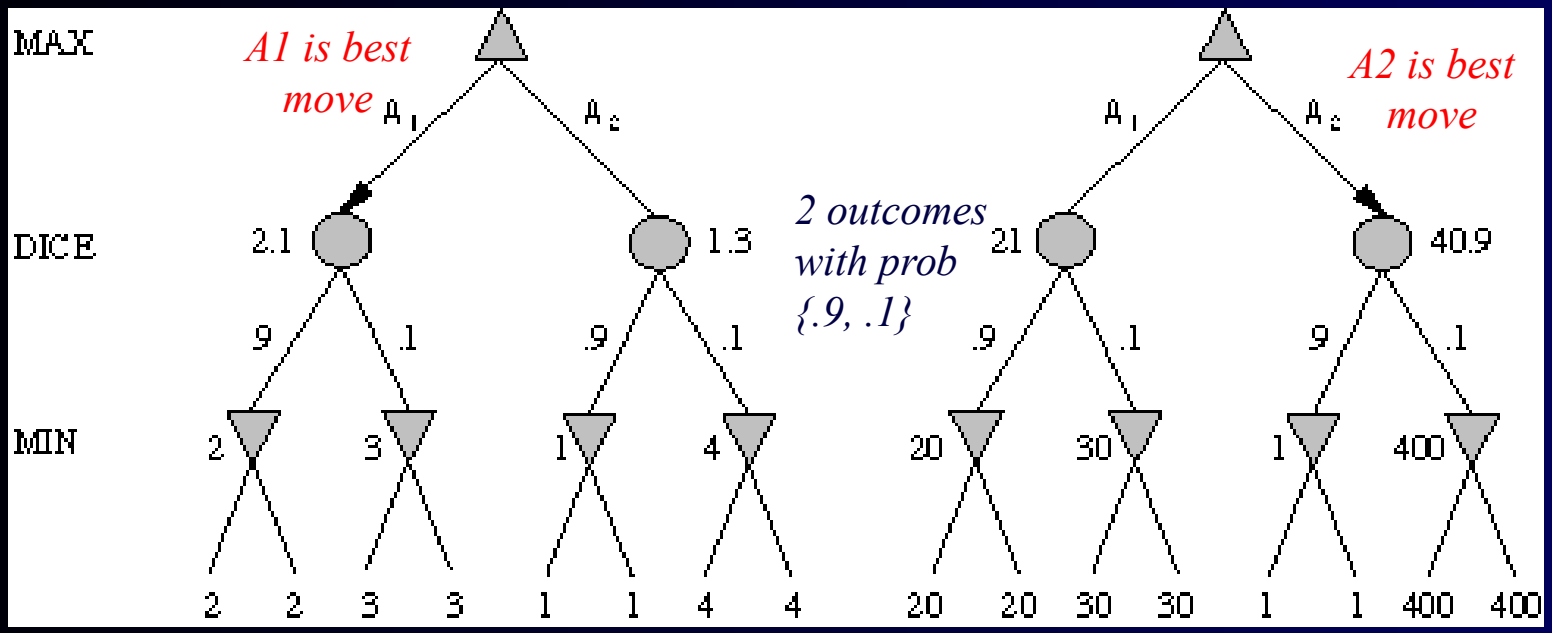
expectimax(C) = $\text{Sum}_i(P(d_i) * \text{maxvalue}(i))$

- For chance nodes over a min node:

expectimin(C) = $\text{Sum}_i(P(d_i) * \text{minvalue}(i))$



Meaning of the evaluation function



- Dealing with probabilities and expected values means we have to be careful about the "meaning" of values returned by the static evaluator.
- Note that a "relative-order preserving" change of the values would not change the decision of minimax, but could change the decision with chance nodes.
- Linear transformations are ok