CSC384: Introduction to Artificial Intelligence

Game Tree Search

- Chapter 5.1, 5.2, 5.3, 5.6 cover some of the material we cover here. Section 5.6 has an interesting overview of State-of-the-Art game playing programs.

- Section 5.5 extends the ideas to games with uncertainty (We won't cover that material but it makes for interesting reading).

Generalizing Search Problem

- So far: our search problems have assumed that the agent has complete control of environment
  - State does not change unless the agent changes it.
  - All we need to compute is a path to a goal state—there is no other agent nor environment that can move you off of that path.
- Assumption not always reasonable
  - Stochastic environment (e.g., the weather, traffic accidents).
  - Other agents whose interests conflict with yours (Focus of this module)
    - Search can find a path to a goal state, but the actions might not lead you to the goal since the **state can be changed by other agents**

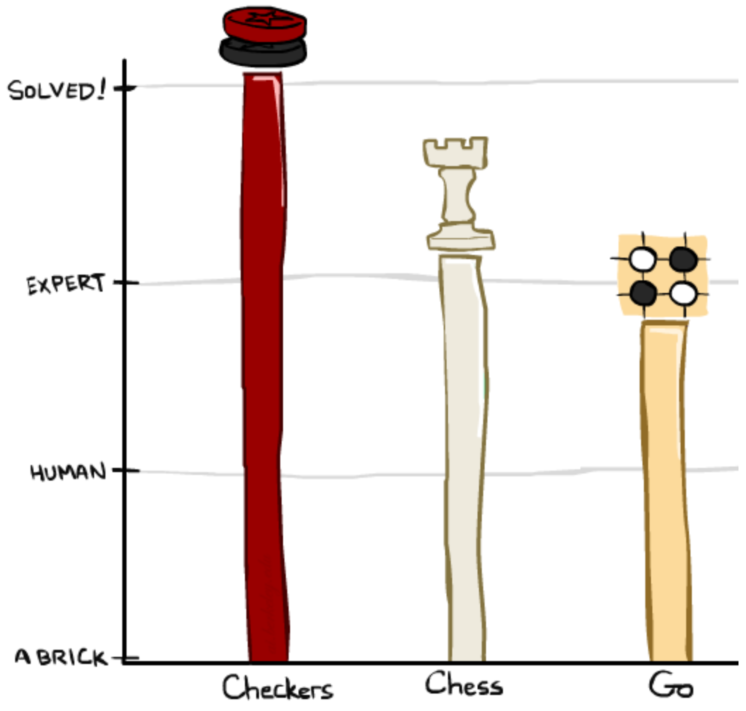
Generalizing Search for external changes

- We need to generalize our view of search to handle state changes that are not in the control of the agent.

- One generalization yields game tree search
  - 2 or more agents
  - All agents acting to maximize their own profits
  - This might not have a positive effect on **your** profits.
  - Can also generalize this approach to deal with the other agent being the “environment” that acts randomly.

Game Playing State-of-the-Art

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!
- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.
- **Go:** Best program AlphaGo has beaten best Go players. In Go, b > 300! Classic programs use pattern knowledge bases, but AlphaGo uses Monte Carlo (randomized) tree search methods, along with Neural Nets to compute heuristics.



[Slide created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley.

General Games

- What makes something a “game”?
  - There are two (or more) agents making changes to the world (the state)
  - Each agent has their own interests and goals
    - each agent has a different goal; or assigns different costs to different paths/states
  - Each agent independently tries to alter the world so as to best benefit itself.
  - Co-operation can occur—but only if it benefits both parties.

General Games

- What makes games hard?
  - How you should play depends on how you think the other person will play;
  - How the other person plays depends on how they think you will play.
  - This introduces a joint-dependency

Properties of Games

- **Two player**:
  - Algorithms presented can be extended to multi-player games, but multi-player games can also involve alliances where some players cooperate to defeat another player (see Chapter 5.2.2 of recommended text)
- **Finite**:
  - Finite number of states and moves from each state.
    - Techniques can be extended to deal with infinite games by applying heuristic cutoffs.
    - When the game is too large **Finite** becomes as bad as **infinite** and again heuristic cutoffs need to be used.

Properties of Games

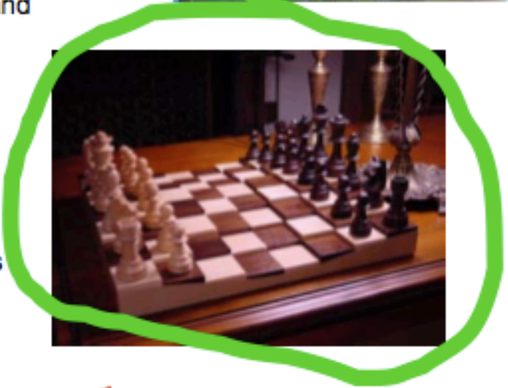
- **Zero-sum (constant sum) games (Fully competitive)**
  - Fully competitive: total payoff to all players is constant, so if one player gets a higher payoff the other player gets a lower payoff. e.g. Poker – you win what the other player lose
- **Non-zero sum**. Games can also be cooperative: some outcomes are preferred by both of us, or at least our values aren't diametrically opposed
- **Deterministic: no chance involved**
  - (no dice, or random deals of cards, or coin flips, etc.
- **Perfect information (all aspects of the state are fully observable,**
  - e.g., no hidden cards

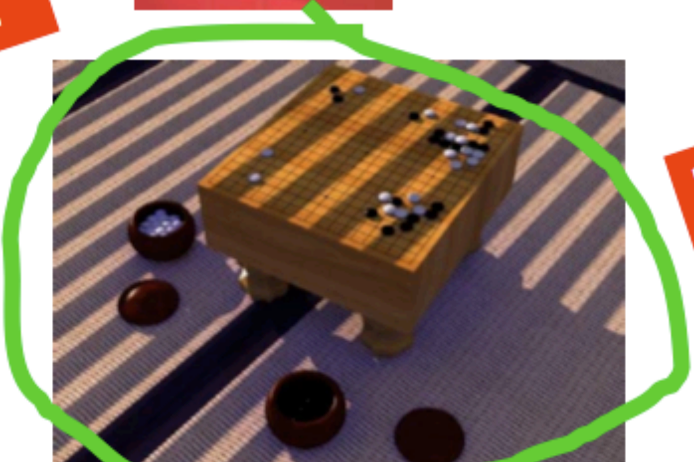
Which of these are: 2-player zero-sum discrete finite deterministic games of perfect information

- Two player: Duh!
- Zero-sum: In any outcome of any game, Player A's gains equal player B's losses.
- Discrete: All game states and decisions are discrete values.
- Finite: Only a finite number of states and decisions.
- Deterministic: No chance (no die rolls).
- Perfect information: Both players can see the state, and each decision is made sequentially (no simultaneous moves).

Which of these are: 2-player zero-sum discrete finite deterministic games of perfect information

- Two player: Duh!
- Zero-sum: In any outcome of any game, Player A's gains equal player B's losses.
- Discrete: All game states and decisions are discrete values.
- Finite: Only a finite number of states and decisions.
- Deterministic: No chance (no die rolls).
- Perfect information: Both players can see the state, and each decision is made sequentially (no simultaneous moves).





Involves Improbable Animal Behavior

Game 1: Rock, Paper, Scissors

- Scissors cut paper, paper covers rock, rock smashes scissors
- Represented as a matrix: Player I chooses a row, Player II chooses a column
- Payoff to each player in each cell (Pl.I / Pl.II)
- 1: win, 0: tie, -1: loss
  - so it's zero-sum

Player II

Player I

	R	P	S
R	0/0	-1/1	1/-1
P	1/-1	0/0	-1/1
S	-1/1	1/-1	0/0

Game 2: Prisoner's Dilemma

- Two prisoner's in separate cells, sheriff doesn't have enough evidence to convict them. They agree ahead of time to both deny the crime (they will cooperate).
- Payoff: 4 minus sentence—go to jail for 4 years zero payoff, go to jail for 0 years payoff = 4.
- If one defects (i.e., confesses) and the other doesn't
  - confessor goes free (payoff = 4-0 = 4)
  - other sentenced to 4 years (payoff = 4-4 = 0)
- If both defect (confess)
  - both sentenced to 3 years (payoff = 4 – 3 = 1)
- If both cooperate (neither confesses)
  - both sentenced to 1 year on minor charge (payoff = 4 - 1 = 3)

	Coop	Def
Coop	3/3	0/4
Def	4/0	1/1

- Not Zero Sum!

Extensive Form Two-Player Zero-Sum Games

- Key point of previous games: what you should do depends on what the other person does

- But previous games are simple “one shot” games
  - single move each
  - in game theory: strategic or normal form games

- Many games extend over multiple moves
  - turn-taking: players act alternatively
  - e.g., chess, checkers, etc.
  - in game theory: extensive form games

- We’ll focus on the extensive form
  - that’s where the computational questions emerge

Two-Player Zero-Sum Game – Definition

- Two players Max and Min
- Set of positions P (states of the game)
- A starting position p ∊ P (where game begins)
- Terminal positions $T \subseteq P$ (where game can end)
- Set of directed edges E_{Max} between some positions (Max's moves)
- set of directed edges E_{Min} between some positions (Min's moves)
- Utility or payoff function $U : T \rightarrow \mathbb{R}$ (how good is each terminal state for player Max)
    - Why don't we need a utility function for Min?

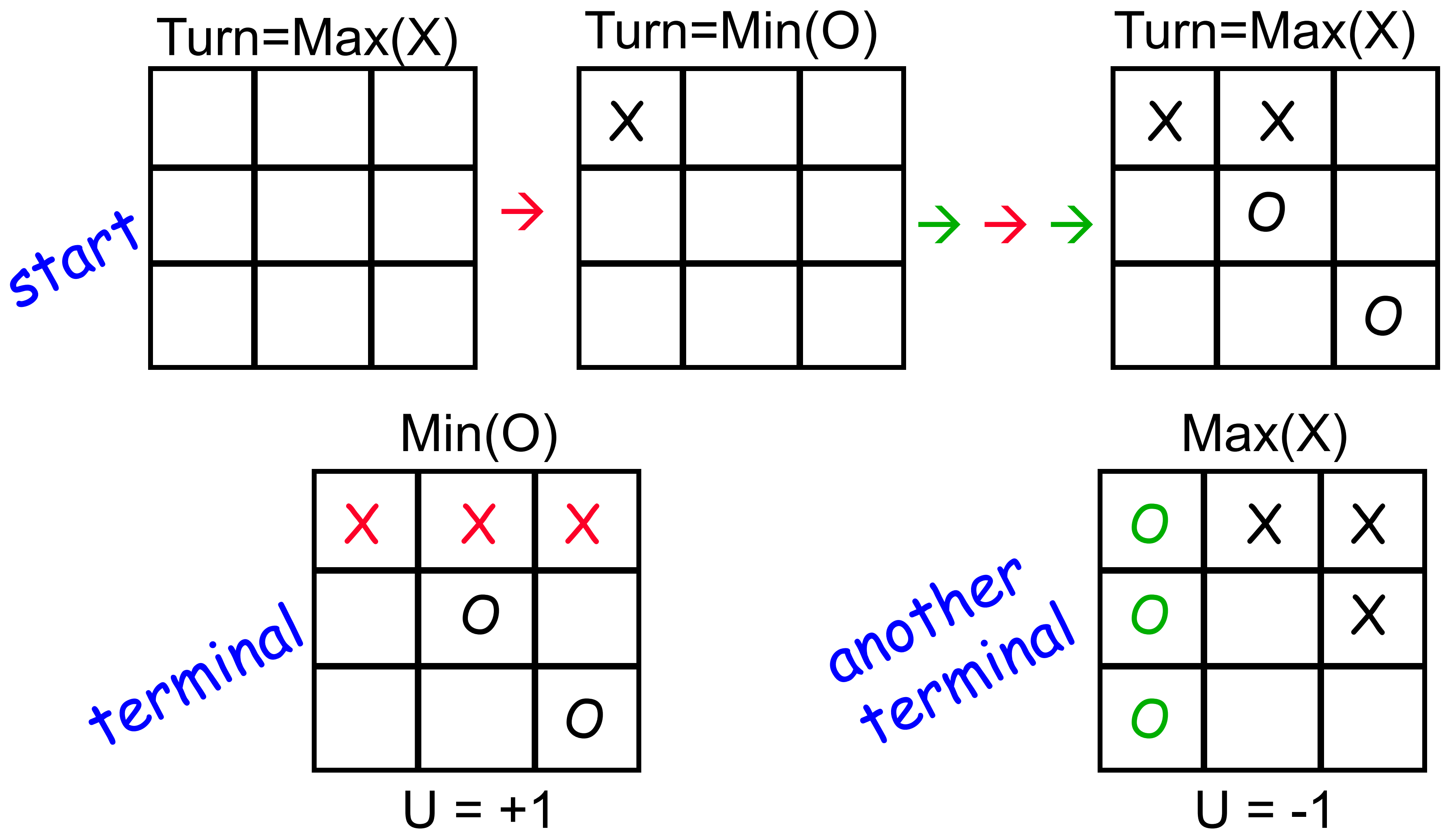
Two-Player Zero-Sum Game – Intuition

- Each position also specifies whose turn it is
  - Game ends when some terminal state ∊ T is reached

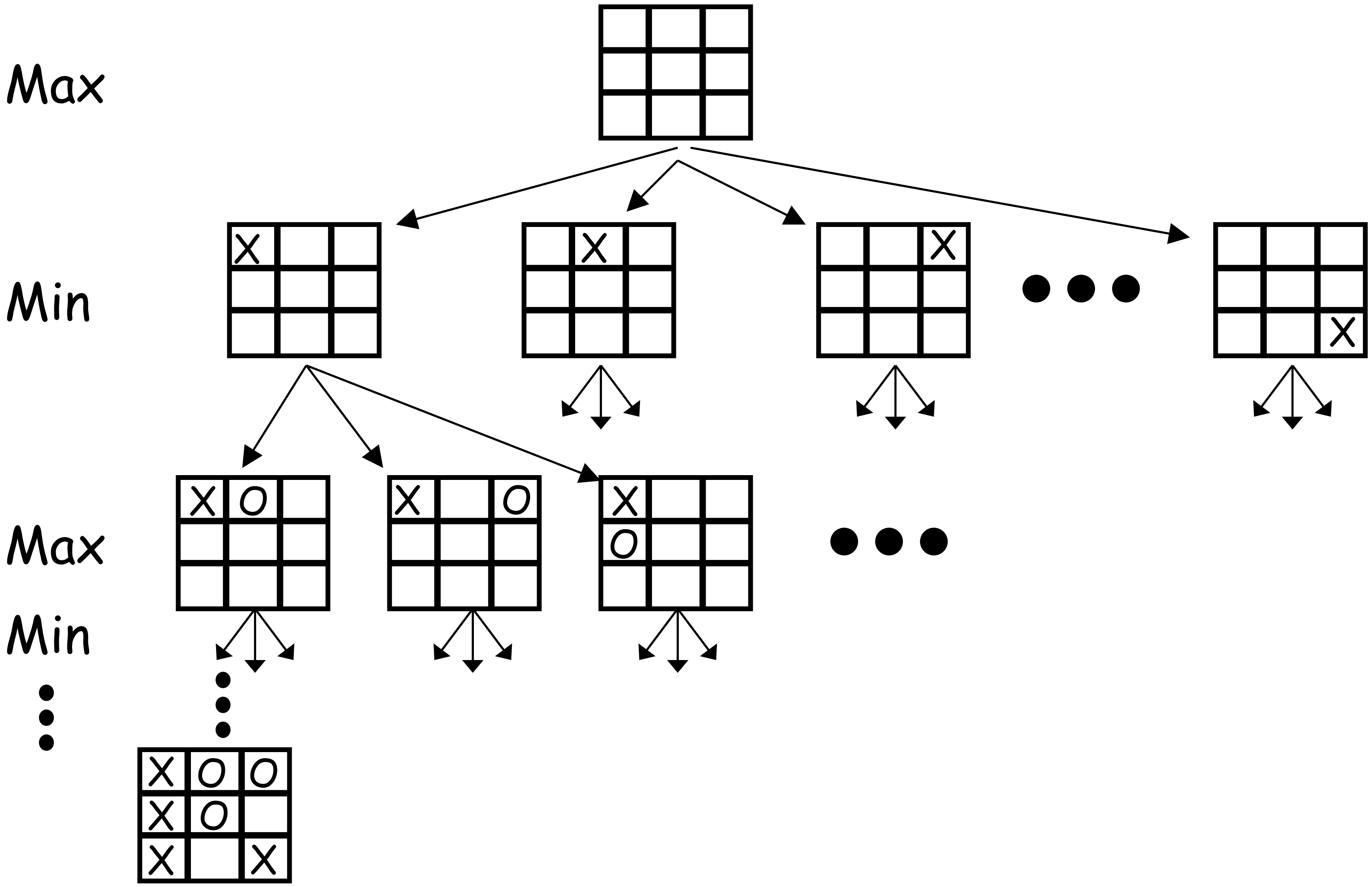
- Utility function and terminals replace goals
  - Max wants to maximize the terminal payoff
  - Min wants to minimize the terminal payoff

- Think of it as:
  - Max gets U(t), Min gets –U(t) for terminal node t
  - This is why it's called zero sum
    - If sum isn't zero but instead the constant C.
    - Max gets U(t), Min gets C – U(t) if there is a constant sum "C" (Min gets the remainder after Max gets their winnings).

Tic Tac Toe Positions



Tic Tac Toe Game Tree



Functions to support Game Tree Search

- The initial state START
- player(p): returns the player whose turn it is (in position p).
- actions(p): returns the set of legal moves for player(p) in position (p) (this function defines the directed edges E_{MAX} and E_{MIN})
- result(p,m): new position after making move m (note new position also specifies new player. In some games a player might get multiple moves in succession.
- terminal(p): returns True if the game is over in position p.
- utility(p): returns MAX's payoff in a terminal position p.

Game Tree

- Game tree has
  - Layers reflect alternating moves between Max and Min
  - Full Game Tree is large (10^{40} for chess). During game tree search we examine a subtree of the game tree, large enough to determine what move to make (but hopefully not too large).
- Player Max doesn't determine which terminal state is reached
  - After Max moves to a state, Min decides which subsequent state to move to
  - So the moves of Max and Min jointly determine the terminal state reached.
- Thus Max must have a strategy
  - Must know (or compute) what to do for each possible move of Min
  - One sequence of moves will not suffice: "What to do" will depend on how Min will play

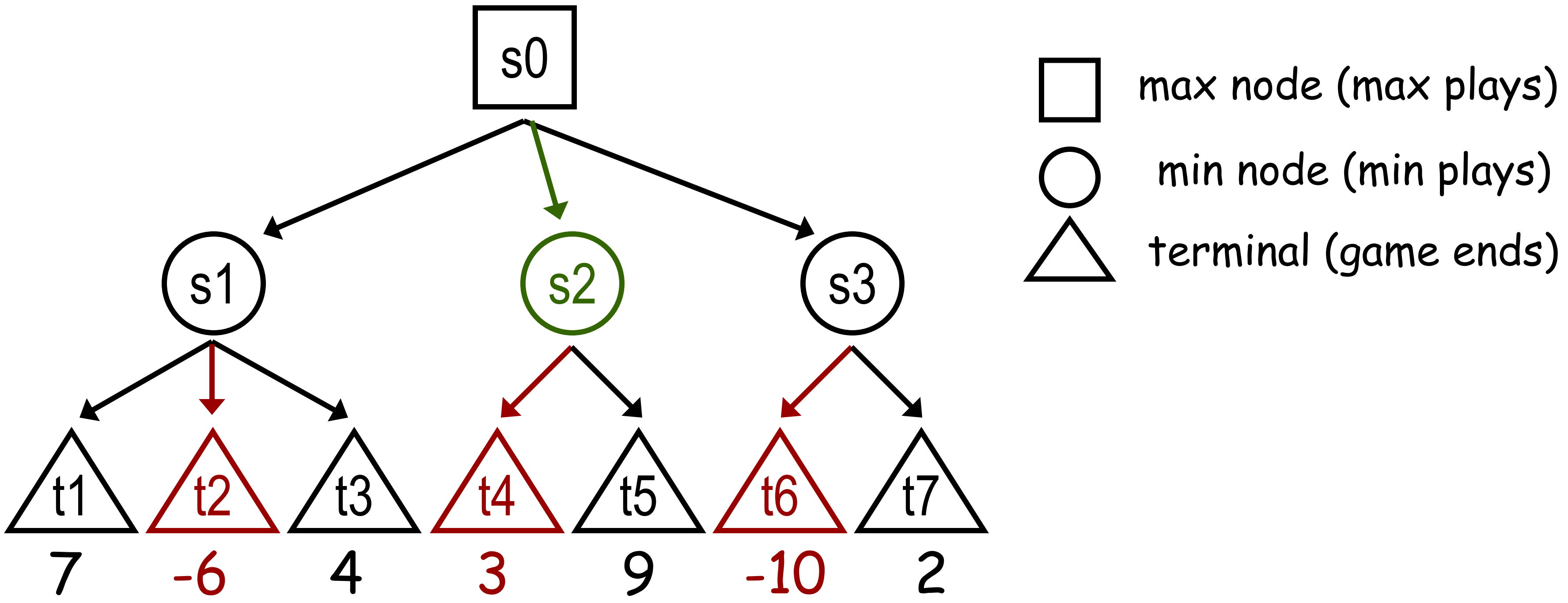
- What is a reasonable strategy?

Minimax Strategy

- Assume that the other player will always play their best move,
  - you always play a move that will minimize the payoff that could be gained by the other player.
  - By minimizing the other player's payoff you maximize yours.
- If you know that Min will play poorly in some circumstances, there might be a better strategy than MiniMax (i.e., a strategy that gives you a better payoff).

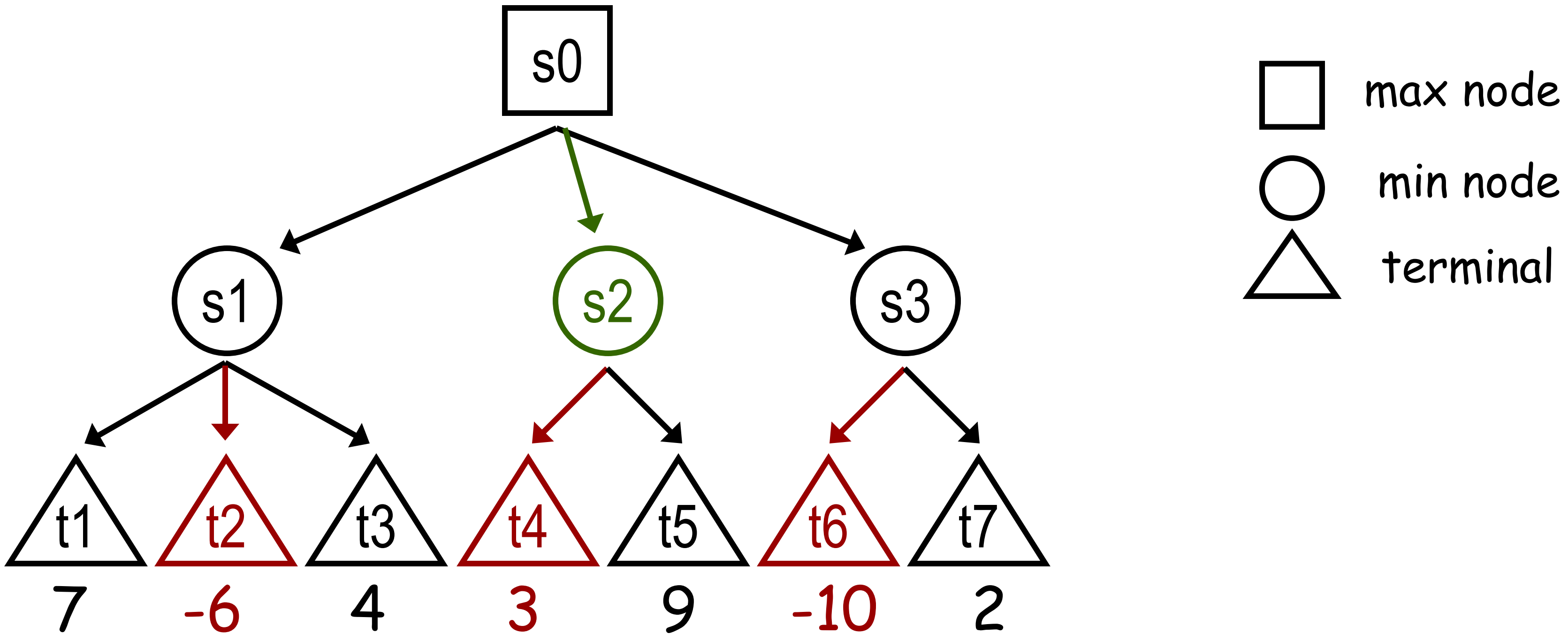
- But in the absence of that knowledge minimax "plays it safe". minimax gives Max the maximum minimum payoff over all possible strategies Min could play. (So if you know something about Min's strategy, e.g., they are not an optimal player, then a better strategy for Max might exist than the minimax strategy.

Minimax Strategy payoffs



The terminal nodes have utilities.
But we can compute a "utility" for the non-terminal states, by assuming both players always play their best move.

Minimax Strategy – Intuitions



If Max goes to s1, Min goes to t2, U(s1) = min{U(t1), U(t2), U(t3)} = -6
If Max goes to s2, Min goes to t4, U(s2) = min{U(t4), U(t5)} = 3
If Max goes to s3, Min goes to t6, U(s3) = min{U(t6), U(t7)} = -10

So Max goes to s2: so U(s0) = max{U(s1), U(s2), U(s3)} = 3

Minimax Strategy

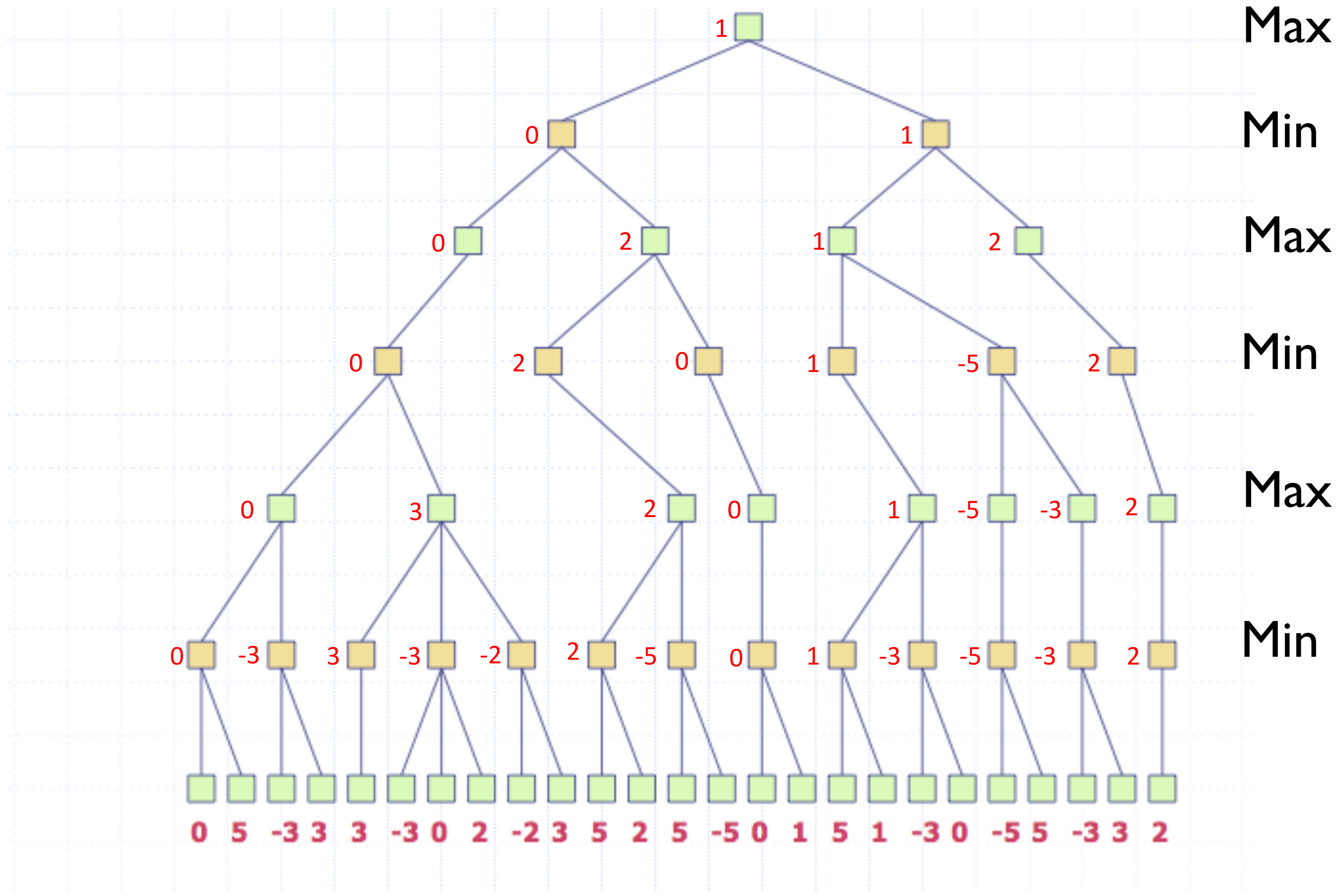
- Build full game tree (all leaves are terminals)
  - Root is start state, edges are possible moves, etc.
  - Label terminal nodes with utilities
  - (only feasible for smaller games!)

- Back values up the tree
  - U(t) is defined for all terminals (part of input)
  - U(n) = min {U(c) : c is a child of n} if n is a Min node
  - U(n) = max {U(c) : c is a child of n} if n is a Max node

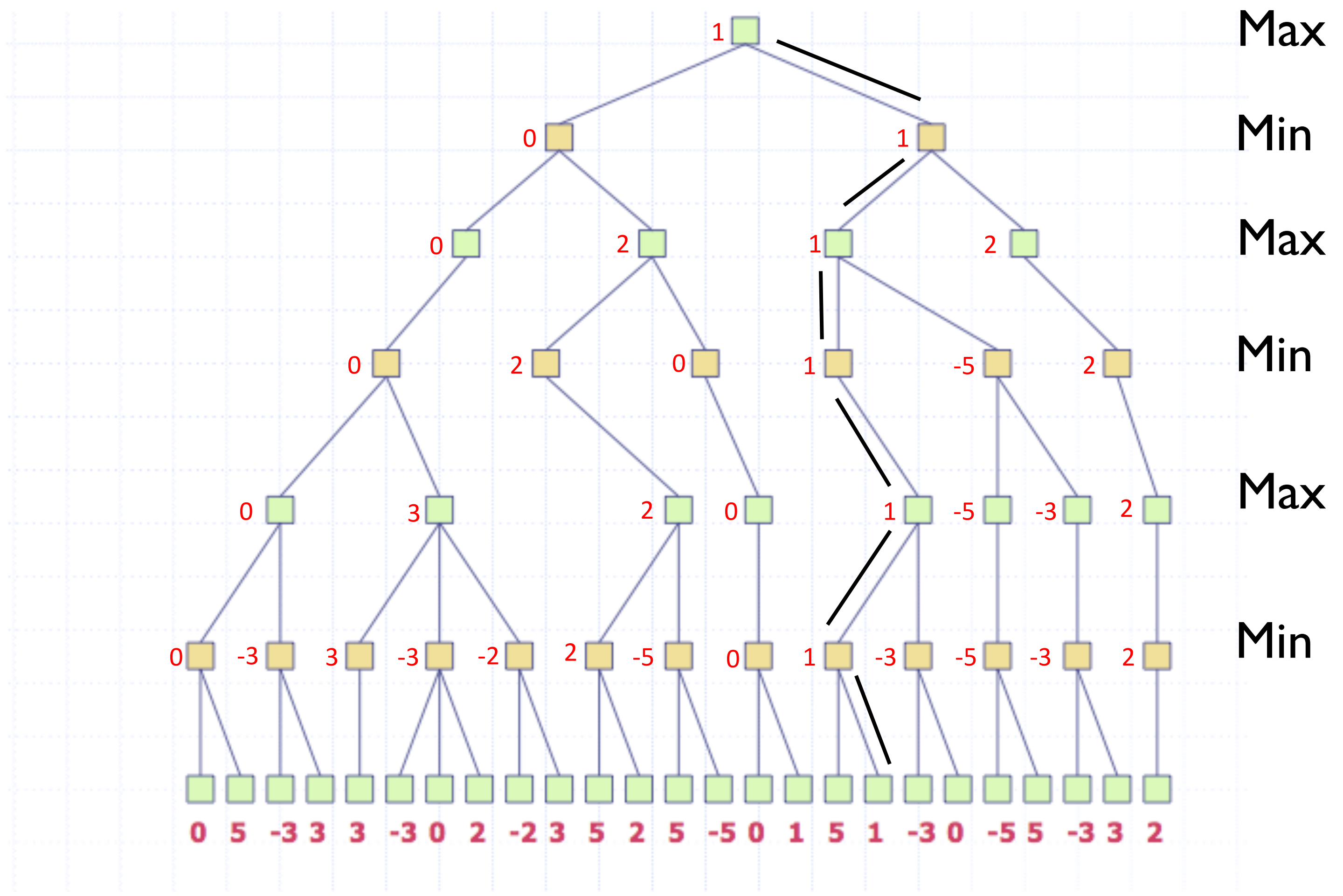
Minimax Strategy

- The values U(n) labeling each state n are the values that Max will achieve in that state if both Max and Min play their best moves.
- Minimax Strategy is:
  - Max always plays a move to change the state to the highest valued child.
  - Min always plays a move to change the state to the lowest valued child.
- If Min plays poorly (does not always move to lowest value child), Max could do better, but never worse.
  - If Max, however knows that Min will play poorly, there might be a better strategy of play for Max than Minimax.

Example


Max
Min
Max
Min
Max
Min
1
0 1
0 2 1 2
0 2 0 1 -5 2
0 3 2 0 1 -5 -3 2
0 -3 3 -3 -2 2 -5 0 1 -3 -5 -3 2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

Example



Max
Min
Max
Min
Max
Min
1
0
1
0
2
1
2
0
2
0
1
-5
2
0
3
2
0
1
-5
-3
2
0
-3
3
-3
-2
2
-5
0
1
-3
-5
-3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

Depth-First Implementation of Minimax

- Building the entire game tree and backing up values gives each player their strategy.
- However, the game tree is exponential in size and might be too large to store in memory.
- Furthermore, as we will see later it is not necessary to know all of the tree.

- We can save space (although still use exponential time) by computing the minimax values with a depth-first implementation of minimax.

  We run a depth-first search after each move to compute what is the next move for the MAX player. (We could do the same for the MIN player).

- This avoids explicitly representing the exponentially sized game tree: we just compute each move as it is needed. (Used space linear in the depth of the game tree)

Depth-First Implementation of Minimax

```
DFMiniMax(pos): #Return best move for player(pos)
                #and MAX's value for pos.
  best_move = None
  if terminal(pos):
    return best_move, utility(pos)
  if player(pos) == MAX: value = -infinity
  if player(pos) == MIN: value = infinity
  for move in actions(pos):
    nxt_pos = result(pos, move)
    nxt_val,nxt_move = DFMiniMax(nxt_pos)
    if player(pos) == MAX and value < nxt_val:
      value, best_move = nxt_val, move
    if player(pos) == MIN and value > nxt_value:
      value, best_move = nxt_val, move
  return best_move, value
```

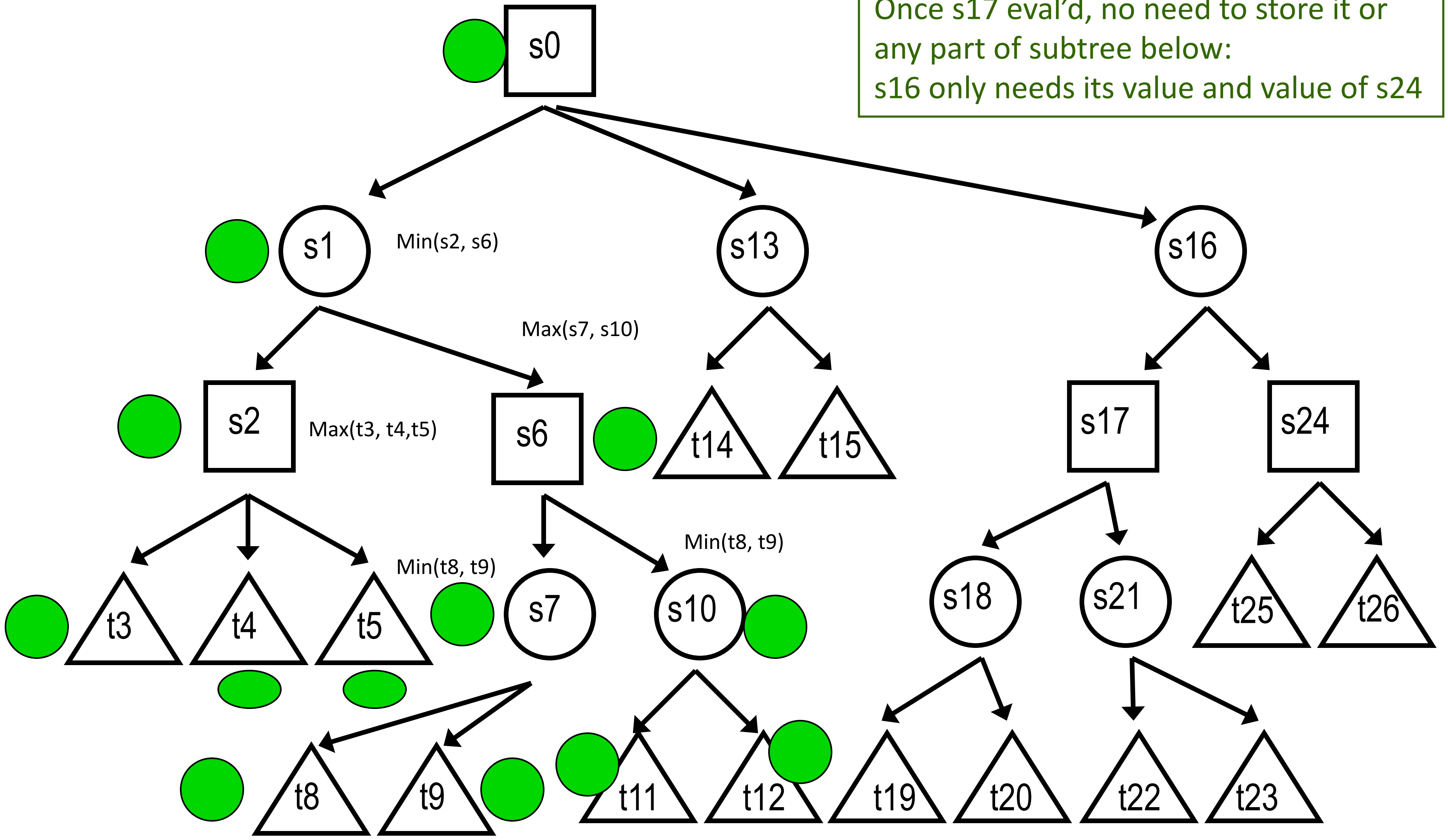
Depth-First Implementation of Minimax

- Notice that the game tree has to have finite depth for this to work (else the recursive calls will never stop)

- Advantage of DF implementation: space efficient

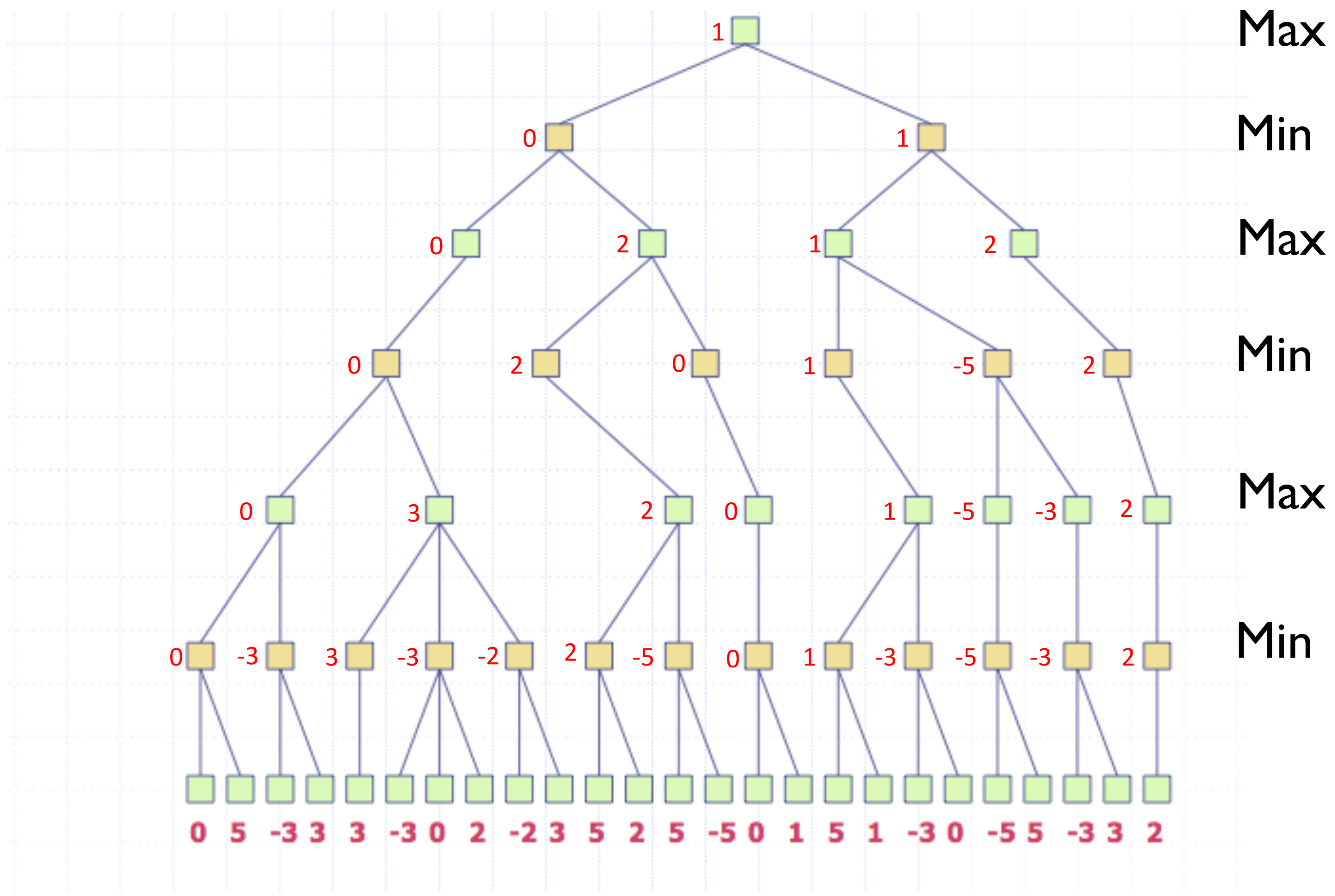
- Minimax will expand $O(b^d)$ states, which is both a BEST and WORSE case scenario.
  - We must traverse the entire search tree to evaluate all options
  - We can’t be lucky as in regular search and find a path to a goal before searching the entire tree.

Visualization of Depth-First Minimax



Once s17 eval'd, no need to store it or any part of subtree below: s16 only needs its value and value of s24

Example

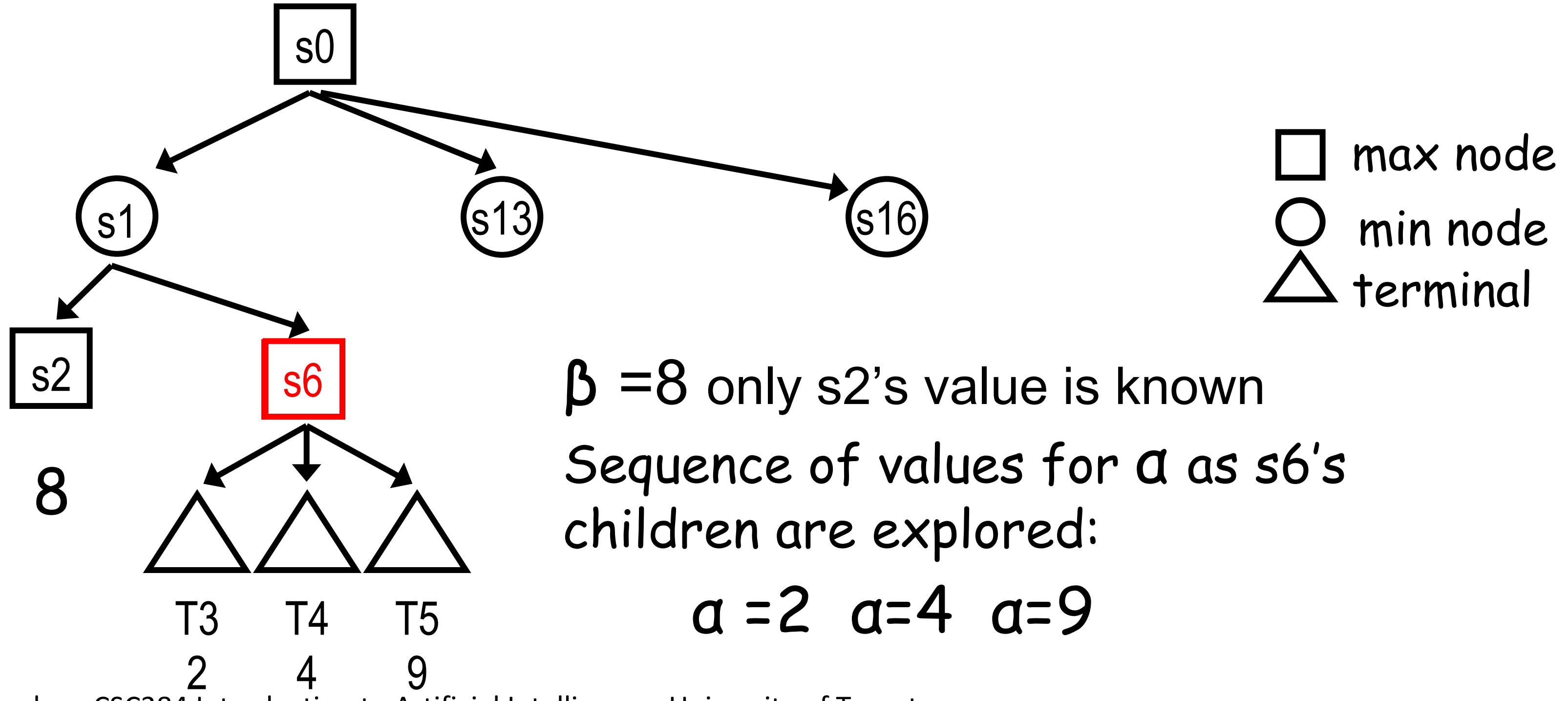


Alpha-Beta Pruning

- It is not necessary to examine entire tree to make correct Minimax decisions
- When using depth-first search of game tree
  - After generating value for only some of n's children we can prove that we'll never reach n in a Minimax strategy.
  - So we needn't generate or evaluate any further children of n
  - These other children can be pruned.
- Two types of pruning (cuts):
  - pruning of children of max nodes (***α***-cuts)
    α is the highest value found so far (MAX wants this)
  - pruning of children of min nodes (β-cuts)
    β is the lowest value found so far (MIN wants this)

Cutting Max Nodes (Alpha Cuts)

- At a max node n (s6):
  - β is the lowest value found so far (from nodes to the left of n that have already been searched) is fixed as we examine n
  - α, the highest value found amongst children of n–changes as children of n are examined



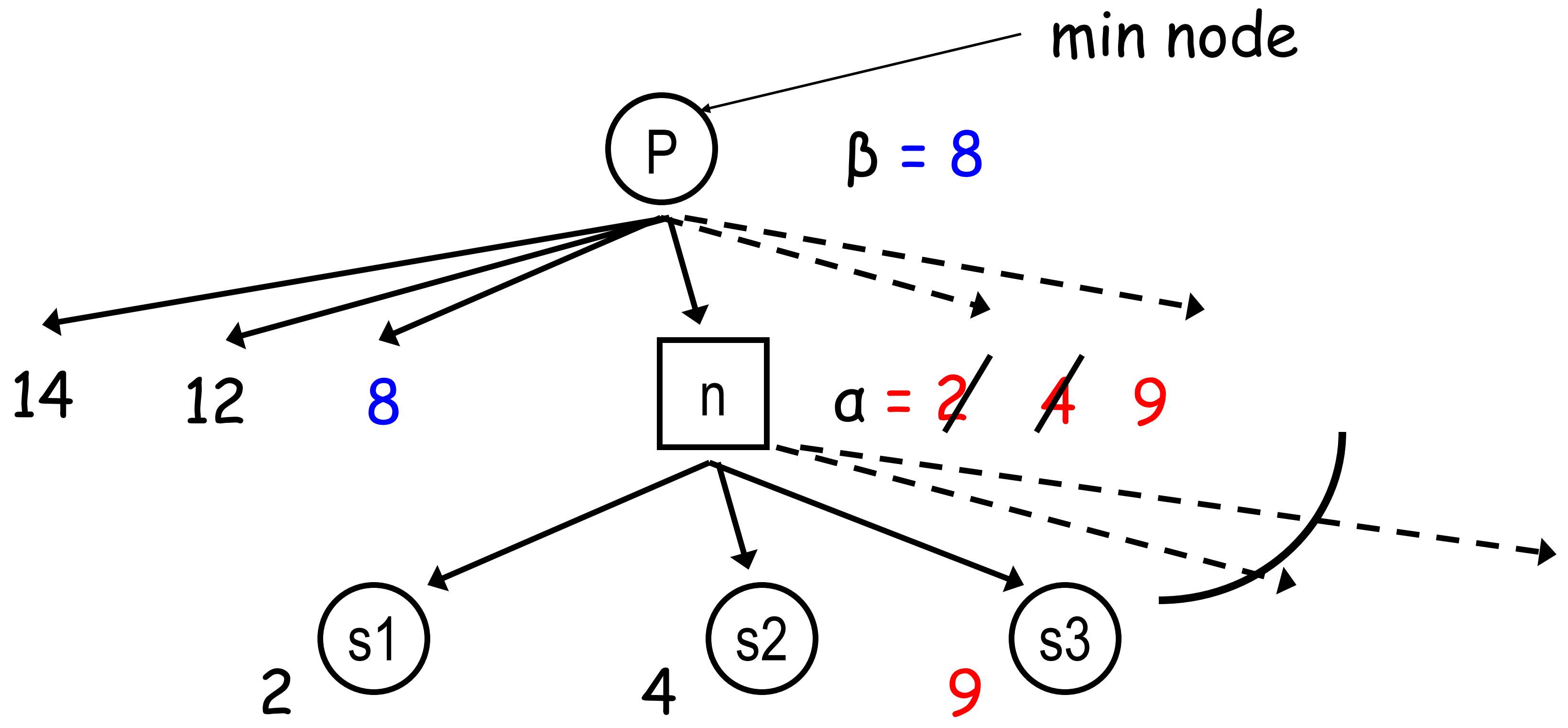
β =8 only s2's value is known

Sequence of values for α as s6's children are explored:

α =2 α=4 α=9

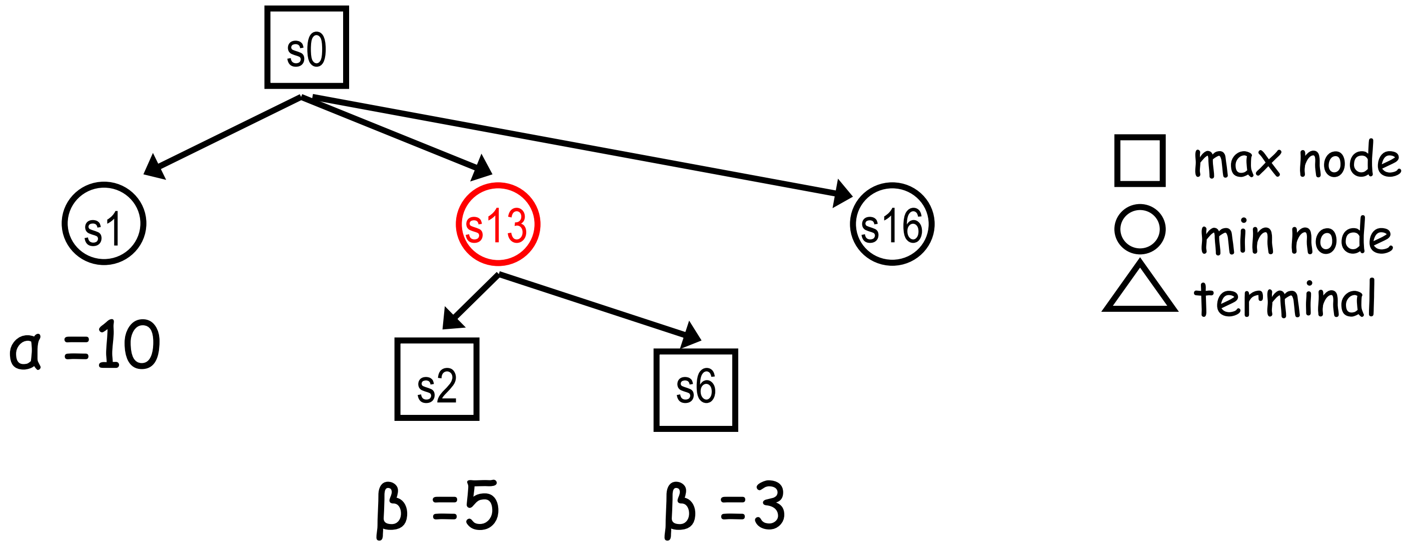
Cutting Max Nodes (Alpha Cuts)

- Allows children of a max node n to be pruned
- If α becomes ≥ β we can stop expanding the children of n
  - Min will never choose to move to n since it would choose one of n's lower valued siblings first.



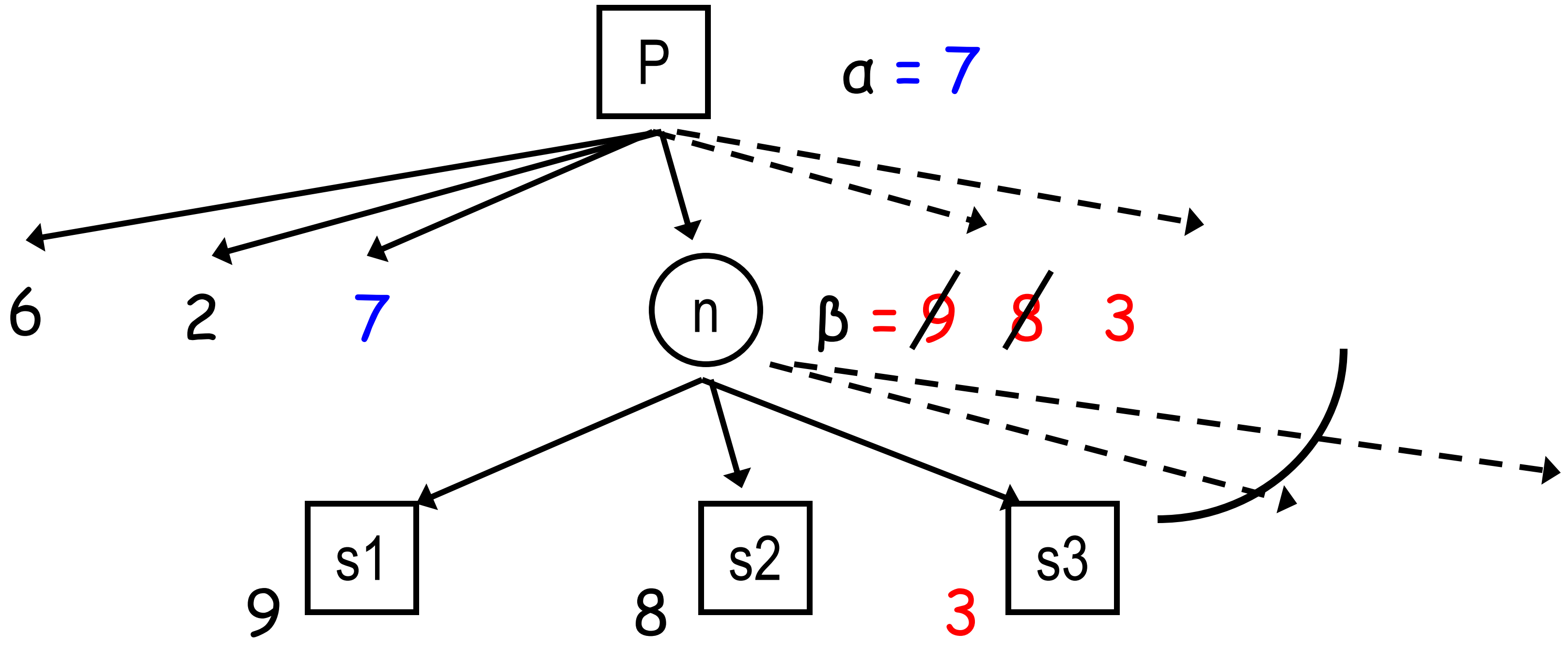
Cutting Min Nodes (Beta Cuts)

- At a Min node n:
  - α is the highest value found so far (from nodes to the left of n that have already been searched) is fixed as we examine n
  - β, the lowest value found amongst children of n–changes as children of n are examined



Cutting Min Nodes (Beta Cuts)

- If β becomes ≤ α we can stop expanding the children of n.
  - Max will never choose to move from n's parent to n since it would choose one of n's higher value siblings first.



Alpha-Beta Pruning Implemenation

```
AlphaBeta(pos,alpha,beta): #return best move for player(pos)
                           #and MAX's value for pos
  best_move = None
  if terminal(pos):
    return best_move, utility(pos)
  if player(pos) == MAX: value = -infinity
  if player(pos) == MIN: value = infinity
  for move in actions(pos):
    nxt_pos = result(pos, move)
    nxt_val,nxt_move = AlphaBeta(nxt_pos, alpha, beta)
    if player(pos) == MAX:
      if value < nxt_val: value, best_move = nxt_val, move
      if value >= beta: return best_move, value
      alpha = max(alpha, value)
    if player(pos) == MIN:
      if value > nxt_value: value, best_move = nxt_val, move
      if value <= alpha: return best_move, value
      beta = min(beta, value)
  return best_move, value
```

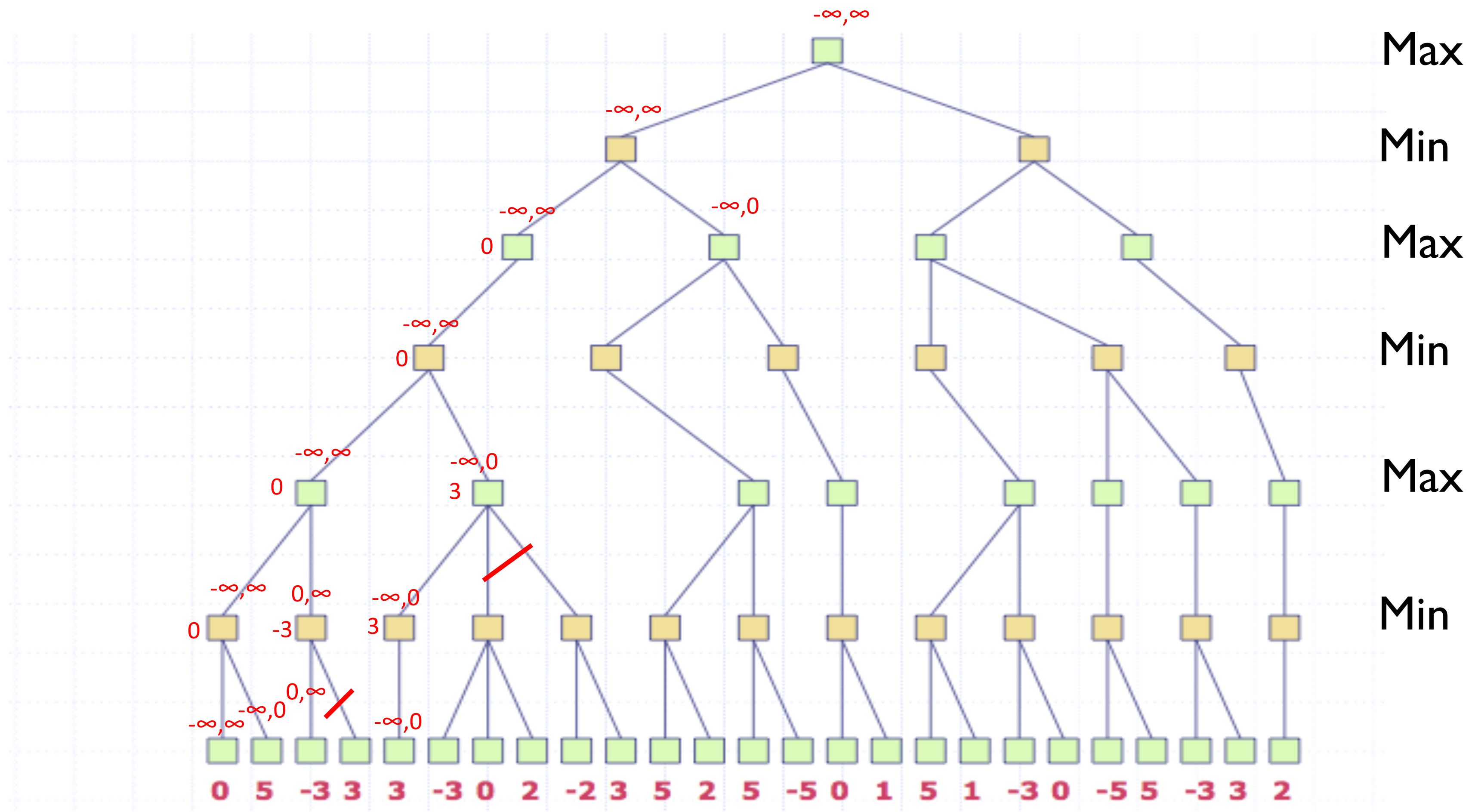
Implementing Alpha-Beta Pruning

```
Initial call

AlphaBeta(START,-infinity,+infinity)
```

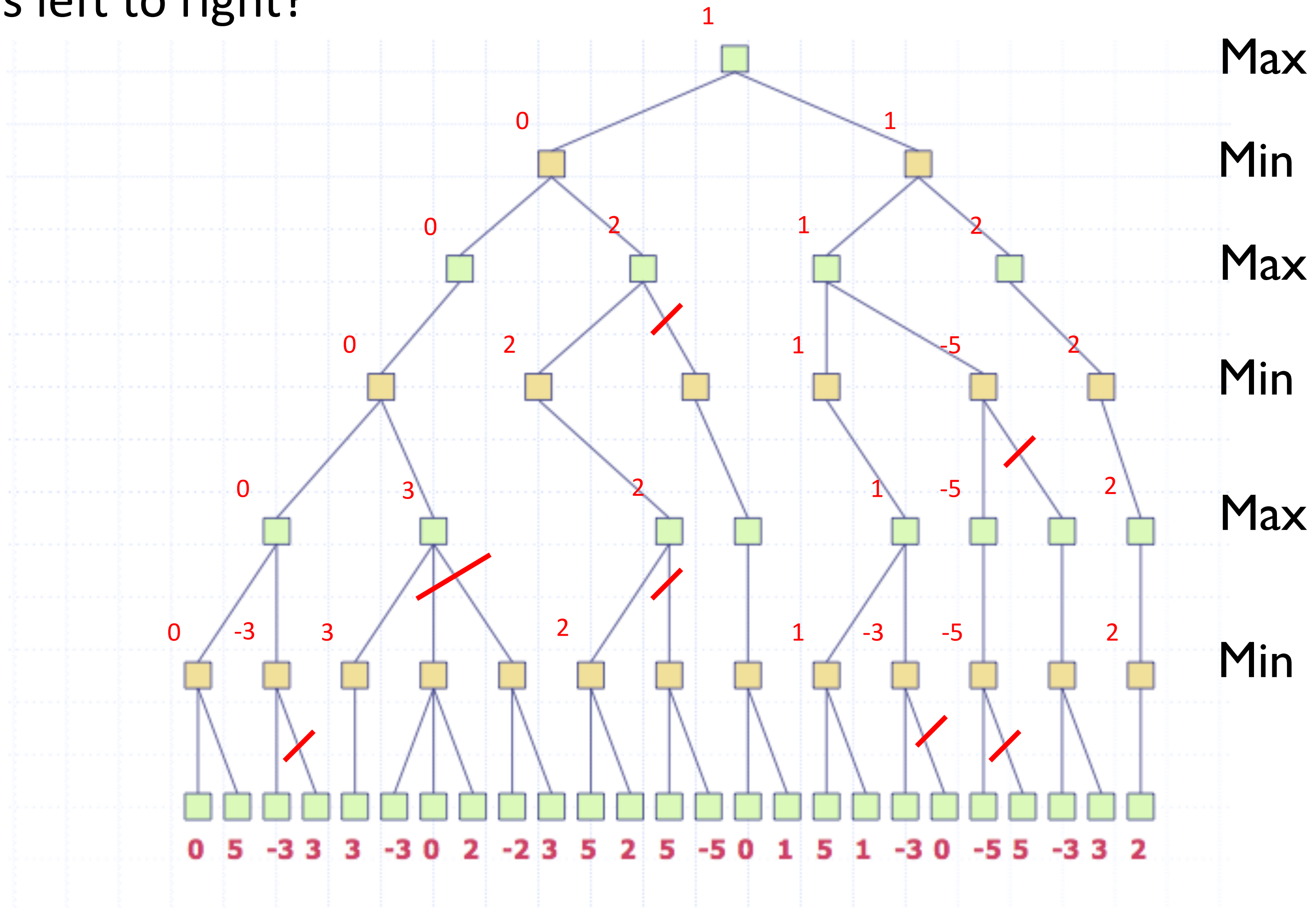
Example

Which computations could we have avoided here? Assuming we expand nodes left to right?

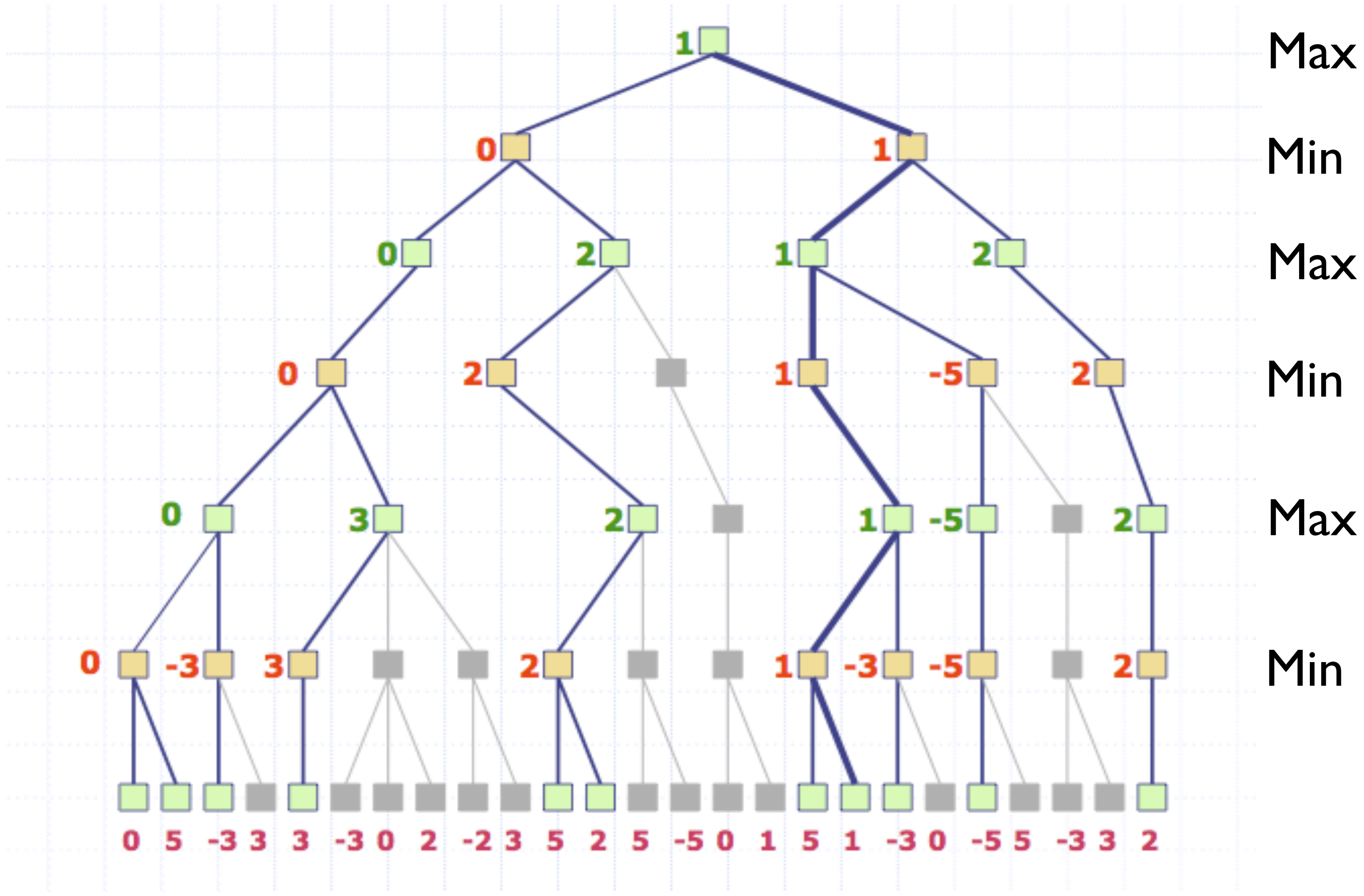


Example

Which computations could we have avoided here? Assuming we expand nodes left to right?



Example



Max
Min
Max
Min
Max
Min
1
0
1
0
2
1
2
0
2
1
-5
2
0
3
2
1
-5
2
0
-3
3
2
1
-3
-5
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

Ordering Moves

- For MIN nodes the best pruning occurs if the best move for MIN (child yielding lowest value) is explored first. (Triggers value <= alpha return early)
- For MAX nodes the best pruning occurs if the best move for MIN (child yielding highest value) is explored first. (Triggers value >= beta return early).
- We don't know with certainty which child has highest or lowest value without doing all of the work!
- But we can use heuristics to estimate the value, and then choose the child with highest (lowest) heuristic value.
- This can make a tremendous difference in practice.

Effectiveness of Alpha-Beta Pruning

- With no pruning, you have to explore $O(b^d)$ nodes, which makes the run time of a search with pruning the same as plain Minimax.
- If, however, the move ordering for the search is optimal (meaning the best moves are searched first), the number of nodes we need to search using alpha beta pruning is $O(b^{d/2})$. That means you can, in theory, search twice as deep!
- In Deep Blue, they found that alpha beta pruning cause the search tree to have an average branching factor of about 6, while without alpha-beta pruning the average branching factor was about 35.

Rational Opponents

- May want to compute your full strategy ahead of time.
  - you must store "decisions" for each node you can reach by playing optimally and assuming your opponent also plays optimally.
  - if your opponent has unique optimal choices, this is a single branch through game tree
  - if there are "ties", opponent could choose any one of the "tied" moves: must store strategy for each subtree
  - In general space is an issue.
  - Alternatively you can recompute your next move at each stage—this is the typical procedure (also useful if you limit the depth of the search)

Practical Matters

- All “real” games are too large to enumerate tree
  - e.g., chess branching factor is roughly 35
  - Depth 10 tree: 2,700,000,000,000,000 nodes
  - Even alpha-beta pruning won’t help here!
- We must limit depth of search tree
  - Can’t expand all the way to terminal nodes
  - We must make heuristic estimates about the values of the non-terminal positions where we terminate the search.
  - These heuristics are often called evaluation function
  - evaluation functions are often learned

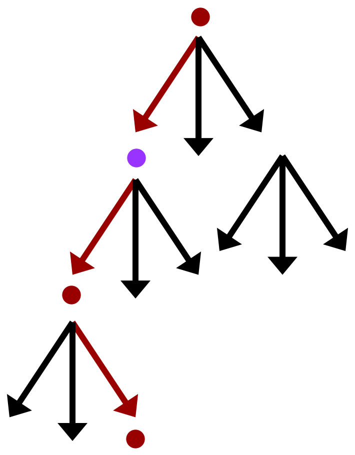
Heuristics in Games

- Example for tic tac toe: h(n) = [# of length 3 runs that are left open for player A] - [# of length 3 runs that are left open for player B].
- Alan Turing's function for chess: h(n) = A(n)/B(n) where A(n) is the sum of the point value for player A's pieces and B(n) is the sum for player B.
- Many evaluation functions can be specified as a weighted sum of features:
  $h(n) = w_1{*}feature_1(n) + w_2{*}feature_2(n) + \ldots\ w_i{*}feature_i(n)$.
  - The weights can be learnt (Samuel's checker player 1949)
- Deep Blue used about 6000 features in its evaluation function.
- AlphaGo used neural networks trained on many examples to provide a estimate of the value of each state. It also used a randomized search that did not explore all children but instead randomly sampled which children to explore. Running the randomized search many times gives an estimate of which move has the best value.

Large Search Problems

- Similar ideas can be used in heuristic search.
- Often we can't expect A* to reach a goal by during search.
- So often as with games, we search to some limited depth (or limited number of nodes on OPEN) and then select the first move on the best path we have seen so far.
- Sometimes called online or realtime search
- In general, guarantees of optimality are lost, but we reduce computational/memory expense dramatically

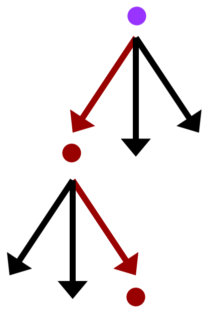
Realtime Search Graphically



1. We run A* (or our favorite search algorithm) until we are forced to make a move or run out of memory. Note: no leaves are goal states yet.

2. We use f(n) to decide which path looks best (let’s say it is the red one).

3. We take the first step along the best path (red), by actually making that move.



4. We restart search at the node we reach by making that move. (We may actually cache the results of the relevant part of first search tree if it’s hanging around, as it would with A*).

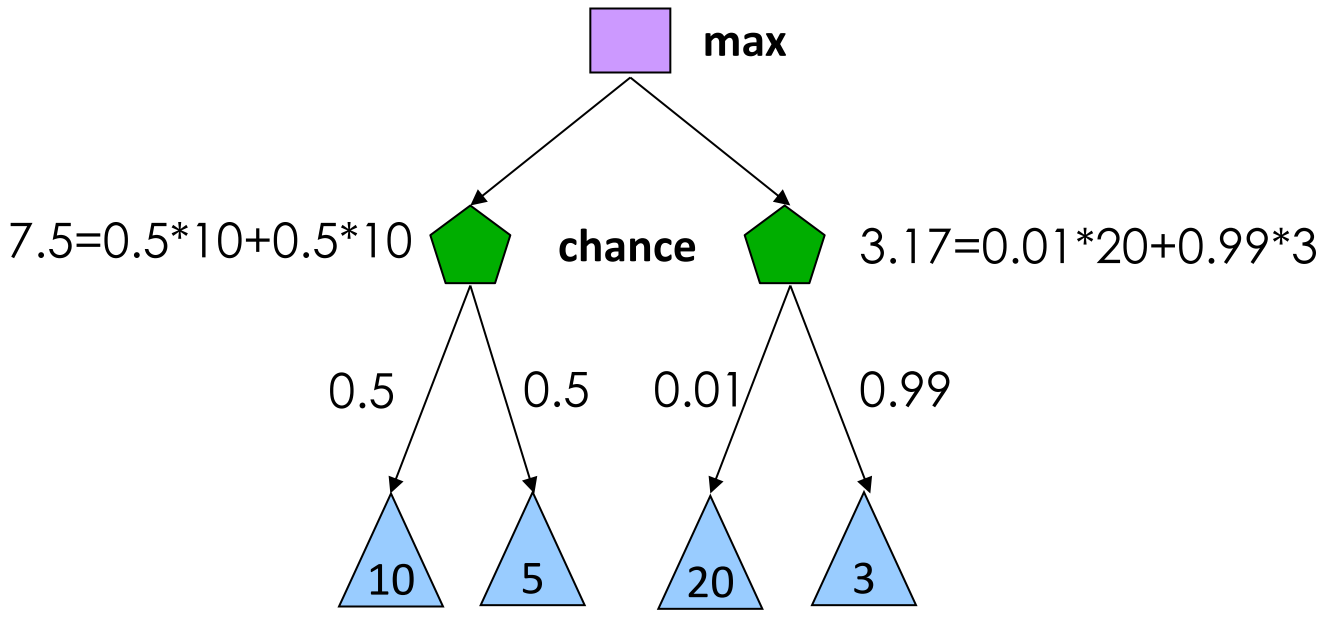
Expectimax Search

- If you don’t know what kind of strategy your opponent uses, then Minimax might playing it too safe.
  - Ensures that in the worst case (very smart opponent) you do best
  - But by playing it safe it might lead to much worse outcomes.
- One important generalization is to consider probabilistic opponents, where your opponent chooses its moves by chance.
  - Perhaps it is more likely to pick the best action, but occasionally picks the worst action.
- Also useful when your opponent is “nature”
- Or when there are chance moves in the game, like throwing of dice.

- Now MAX wants to pick a node that maximizes the expected value.

- Expectimax search: compute the average score
  - Max nodes as in minimax search
  - Chance nodes are like min nodes but which move will be picked is uncertain
  - At chance nodes we calculate expected value

Expectimax Search



Max should pick the child with greatest expected value

Expectimax Search

- We can develop a simple algorithm for dealing with MAX and chance nodes.
- Each to extend to a combination of MAX/MIN/Chance nodes!

Expectimax Search

```
Expectimax(pos): #Return best move for player(pos)
                 #and MAX's value for pos.
  best_move = None
  if terminal(pos):
    return best_move, utility(pos)
  if player(pos) == MAX:     value = -infinity
  if player(pos) == CHANCE: value = 0
  for move in actions(pos):
    nxt_pos = result(pos, move)
    nxt_val,nxt_move = Expectimax (nxt_pos)
    if player == MAX and value < nxt_val:
      value, best_move = nxt_val, move
    if player == CHANCE:
      value = value + prob(move) * nxt_val
  return best_move, value
  #no best_move for CHANCE player
```