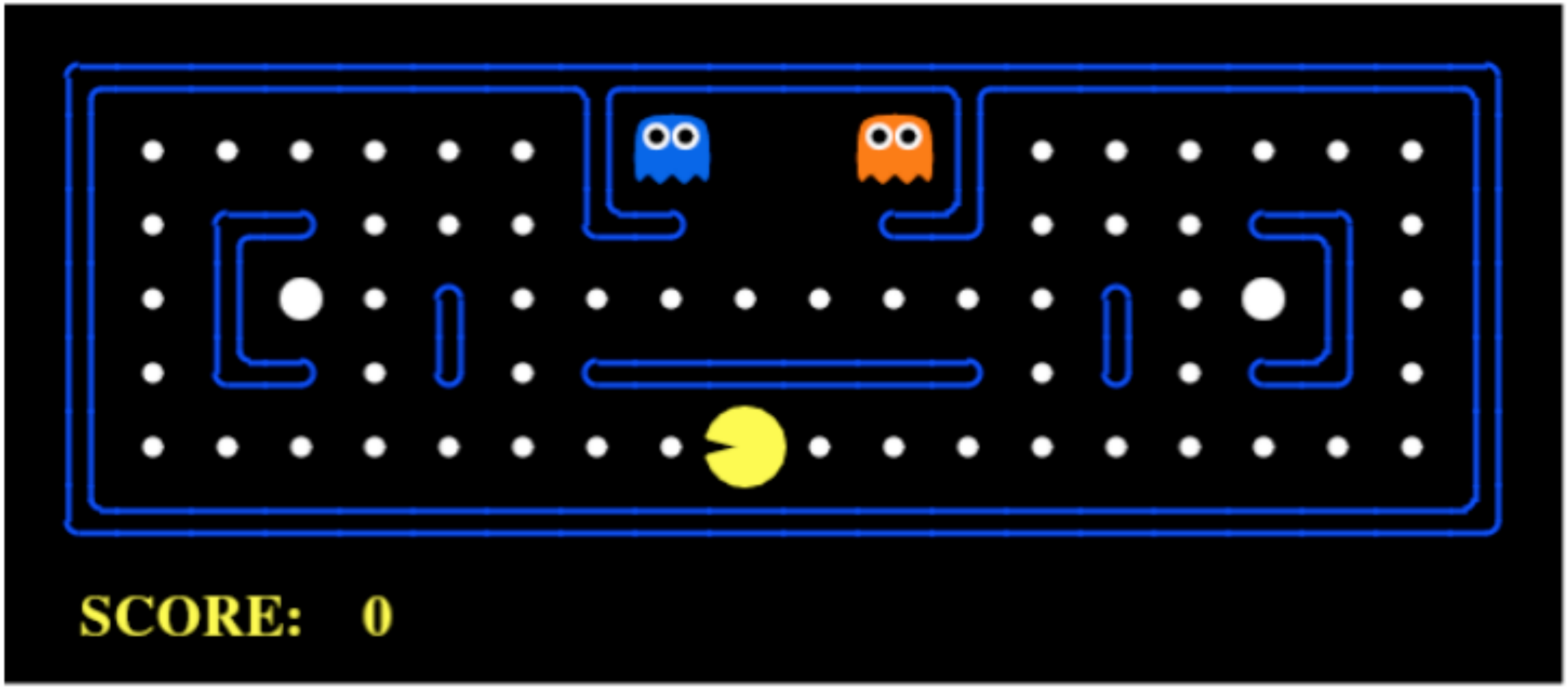
Adversarial Search and Game Playing

Where choosing actions means respecting your opponent

R&N: Chap. 6

With some slides from Dan Klein, Luke Zettlemoyer, Percy Liang, Stuart Russell

Adversarial Search

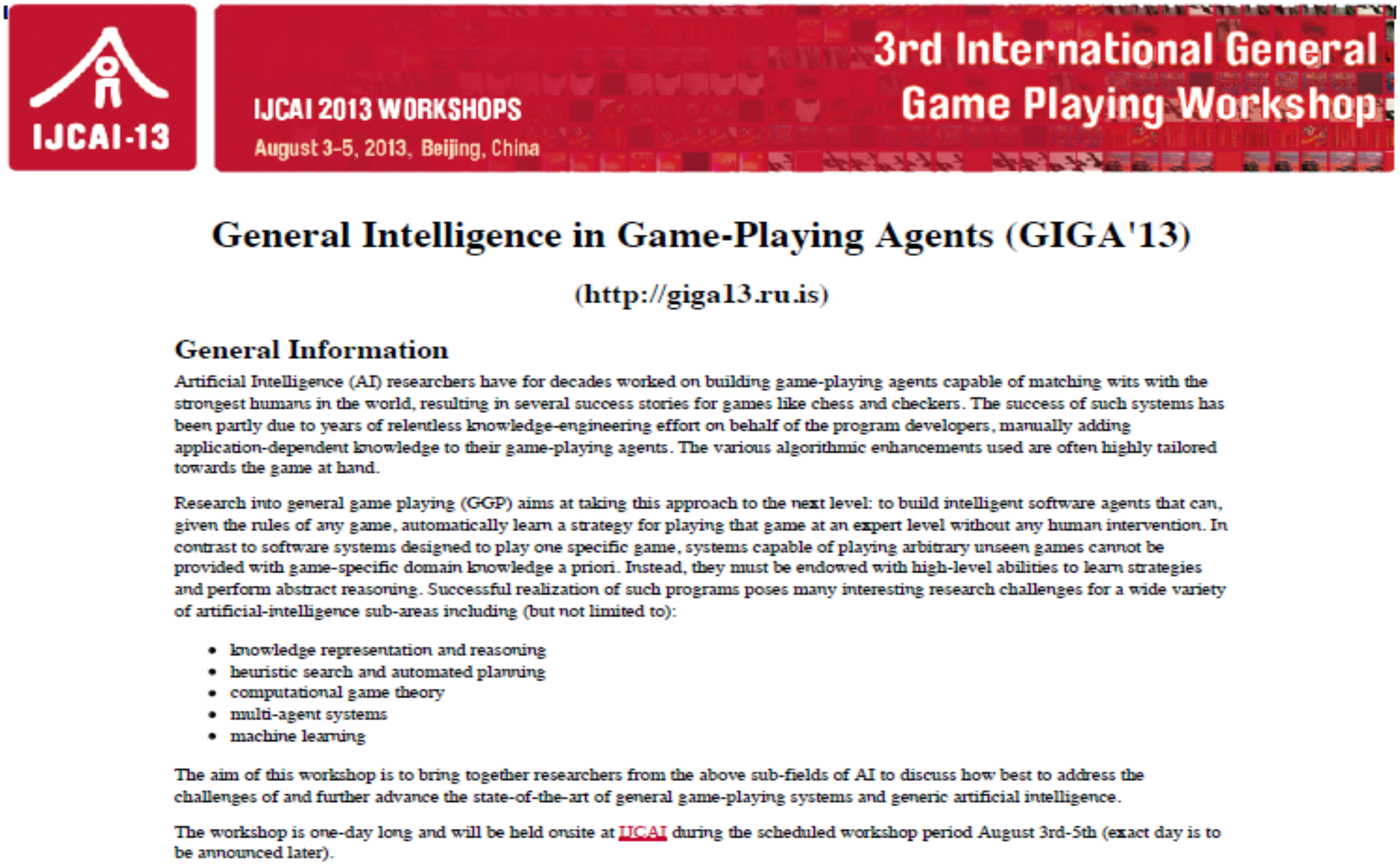


Video of Demo Mystery Pacman

Why Study Games?

- Games mimic the natural world
- For centuries humans have used them for fun, education, and measuring their intelligence
- Making rules (and winning) explicit makes real-world problems possible to analyze

General Game Playing



IJCAI-13

IJCAI 2013 WORKSHOPS

August 3-5, 2013, Beijing, China

3rd International General Game Playing Workshop

General Intelligence in Game-Playing Agents (GIGA'13)

(http://giga13.ru.is)

General Information

Artificial Intelligence (AI) researchers have for decades worked on building game-playing agents capable of matching wits with the strongest humans in the world, resulting in several success stories for games like chess and checkers. The success of such systems has been partly due to years of relentless knowledge-engineering effort on behalf of the program developers, manually adding application-dependent knowledge to their game-playing agents. The various algorithmic enhancements used are often highly tailored towards the game at hand.

Research into general game playing (GGP) aims at taking this approach to the next level: to build intelligent software agents that can, given the rules of any game, automatically learn a strategy for playing that game at an expert level without any human intervention. In contrast to software systems designed to play one specific game, systems capable of playing arbitrary unseen games cannot be provided with game-specific domain knowledge a priori. Instead, they must be endowed with high-level abilities to learn strategies and perform abstract reasoning. Successful realization of such programs poses many interesting research challenges for a wide variety of artificial-intelligence sub-areas including (but not limited to):

- knowledge representation and reasoning
- heuristic search and automated planning
- computational game theory
- multi-agent systems
- machine learning

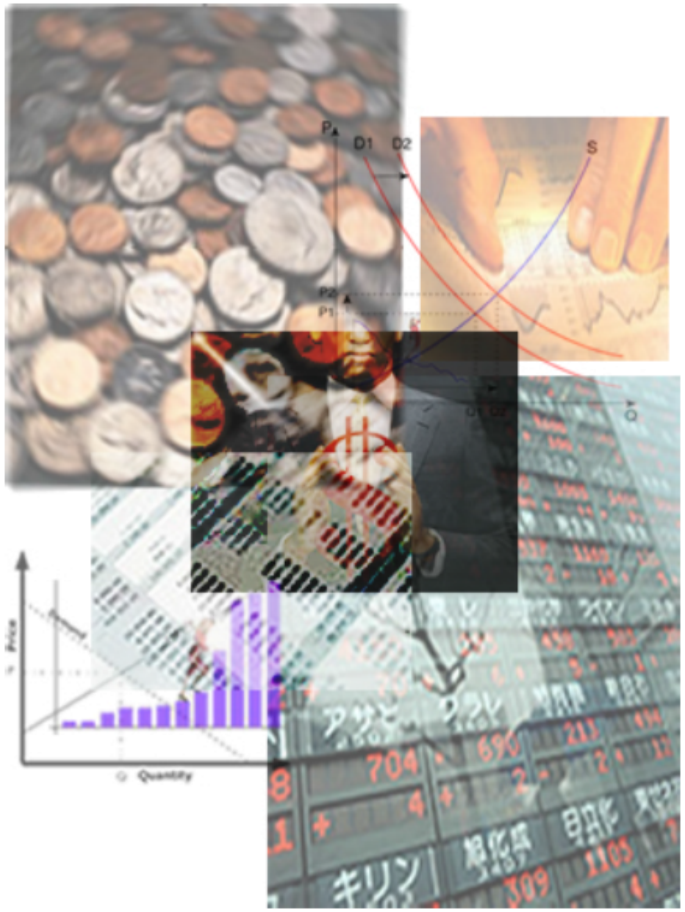
The aim of this workshop is to bring together researchers from the above sub-fields of AI to discuss how best to address the challenges of and further advance the state-of-the-art of general game-playing systems and generic artificial intelligence.

The workshop is one-day long and will be held onsite at IJCAI during the scheduled workshop period August 3rd-5th (exact day is to be announced later).

Real World: Serious Games

Some Fields where Game Theory is Used

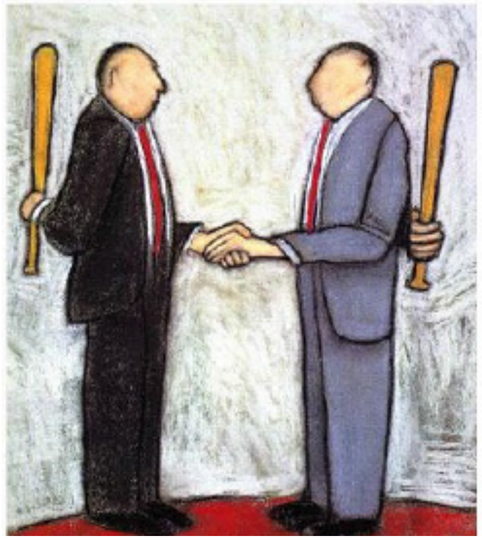
- Economics
  - Auctions
  - Markets
  - Bargaining
  - Fair division
  - Social networks
  - …

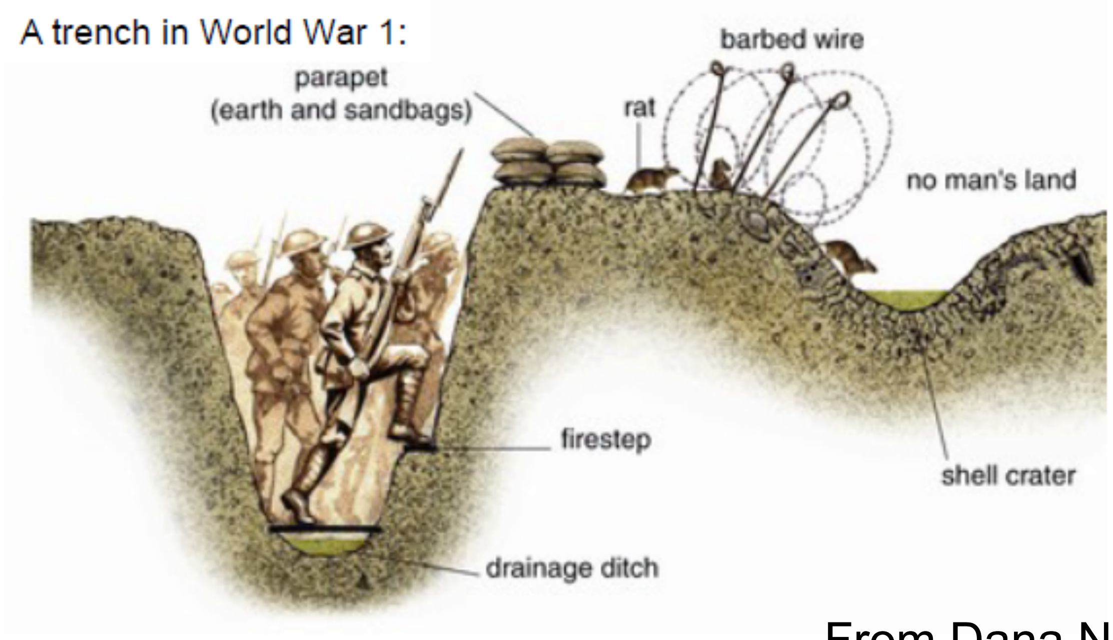


From Dana Nau

Some Fields where Game Theory is Used

- Government and Politics
  - Voting systems
  - Negotiations
  - International relations
  - War
  - Human rights



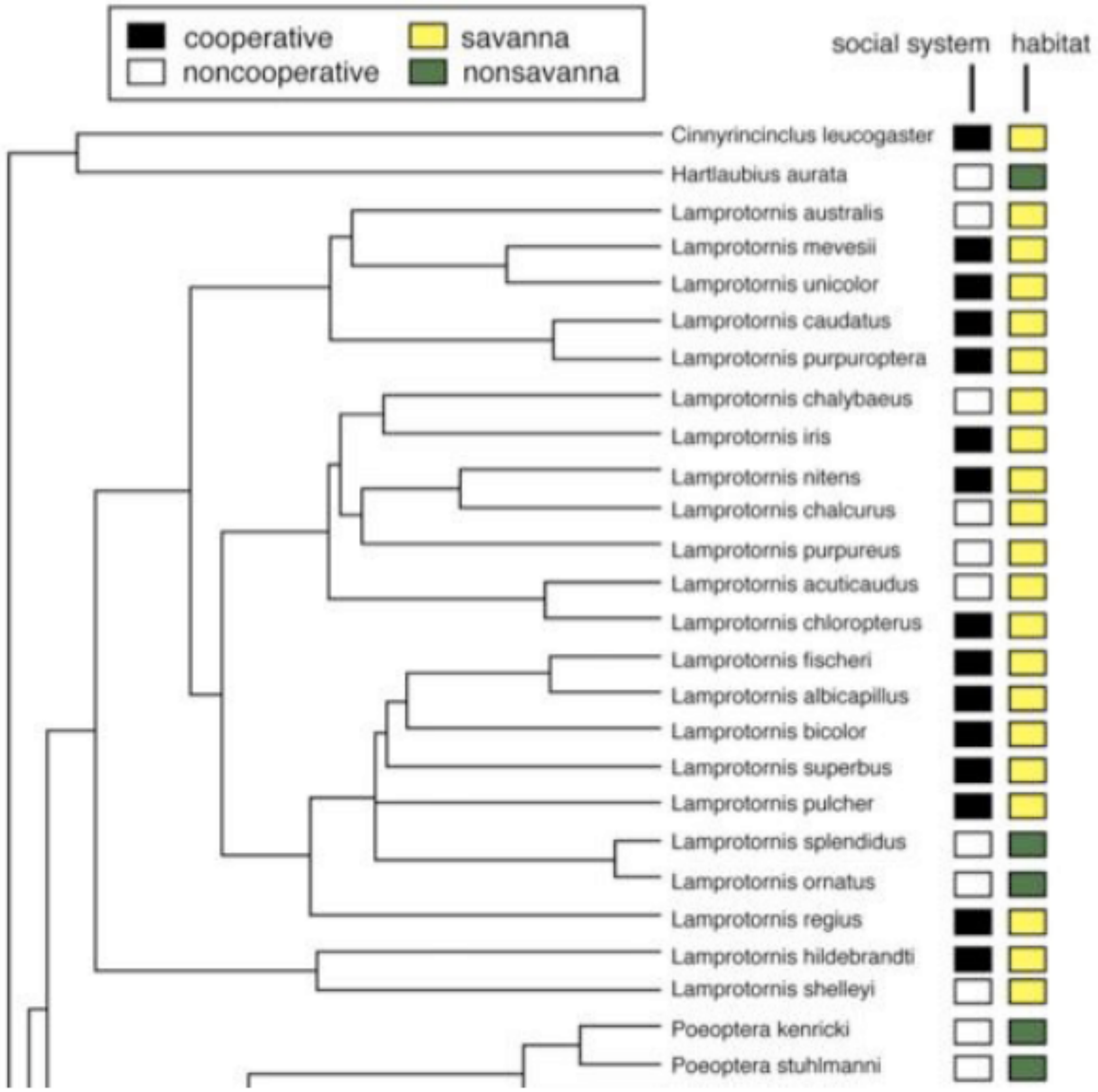


From Dana Nau

Some Fields where Game Theory is Used

- Evolutionary Biology
  - Communication
  - Population ratios
  - Territoriality
  - Altruism
  - Parasitism, symbiosis
  - Social behavior

From Dana Nau



TCP/IP Traffic

From Dana Nau, UMD

- Internet traffic is governed by the TCP protocol
- TCP's *backoff* mechanism
  - If the rate at which you're sending packets causes congestion, reduce the rate until congestion subsides
- Suppose that
  - You're trying to finish an important project
    - It's extremely important for you to have a fast connection
  - Only one other person is using the Internet
    - That person wants a fast connection just as much as you do
- You each have 2 possible actions:
  - C (use a *correct* implementation)
  - D (use a *defective* implementation that won't back off)

TCP/IP: Analysis

From Dana Nau, UMD

- An **action profile** is a choice of action for each agent
  - You both use C => average packet delay is 1 ms
  - You both use D => average delay is 3 ms (router overhead)
  - One of you uses D, the other uses C:
    - D user's delay is 0
    - C user's delay is 4 ms
- **Payoff matrix**:
  - Your options are the rows
  - The other agent's options are the columns
  - Each cell = an action profile
    - 1st number in the cell is your **payoff** or **utility** (I'll use those terms synonymously)
      - In this case, the negative of your delay
    - 2nd number in each cell is the other agent's payoff

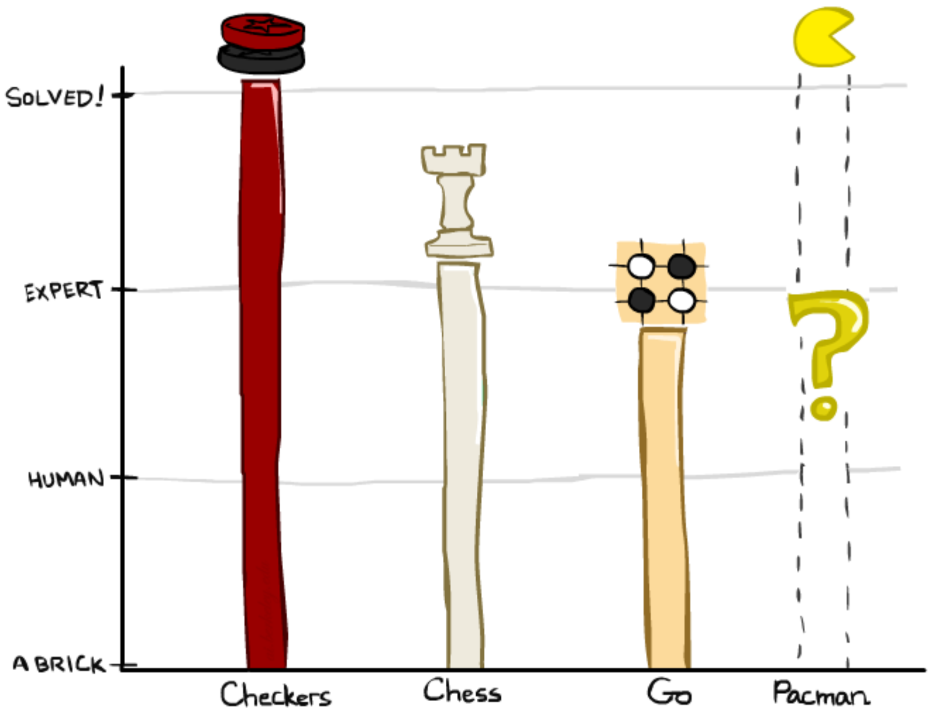
$$
\begin{array}{c|c|c|}
 & C & D \\
\hline
C & -1, -1 & -4, 0 \\
\hline
D & 0, -4 & -3, -3 \\
\hline
\end{array}
$$

General Idea: Rational Agents

- To model games, assume agents are **rational**
- Rational = ?
- What is the rational thing to do in TCP/IP example?
- Rational agent will take a sequence of actions to maximize utility
  - $$
  - Winning a game
  - Minimize delay

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:** Human champions are now starting to be challenged by machines, though the best humans still beat the best machines. In go, b > 300! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.
- **Pacman**



Checkers: Tinsley vs. Chinook

Name:	Marion Tinsley
Profession:	Teaches Math
Hobby:	Checkers
Record:	Over 42 years loses only 3 games of checkers

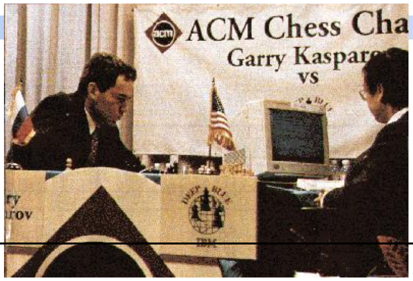
World champion for over 40 years

Mr. Tinsley suffered his 4th and 5th losses against Chinook

Chinook: World Checkers Champion

First computer to become official world champion of Checkers!

Chess: Kasparov vs. Deep Blue



Kasparov		Deep Blue
5’10”	**Height**	6’ 5”
176 lbs	**Weight**	2,400 lbs
34 years	**Age**	4 years
50 billion neurons	**Computers**	32 RISC processors + 256 VLSI chess engines
2 pos/sec	**Speed**	200,000,000 pos/sec
Extensive	**Knowledge**	Primitive
Electrical/chemical	**Power Source**	Electrical
Enormous	**Ego**	None

1997: Deep Blue wins by 3 wins, 1 loss, and 2 draws

Chess: Kasparov vs. Deep Junior

Deep Junior

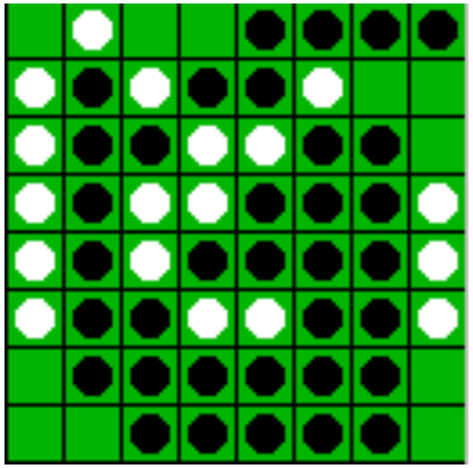
8 CPU, 8 GB RAM, Win 2000

2,000,000 pos/sec

Available at $100

August 2, 2003: Match ends in a 3/3 tie!

Othello: Murakami vs. Logistello



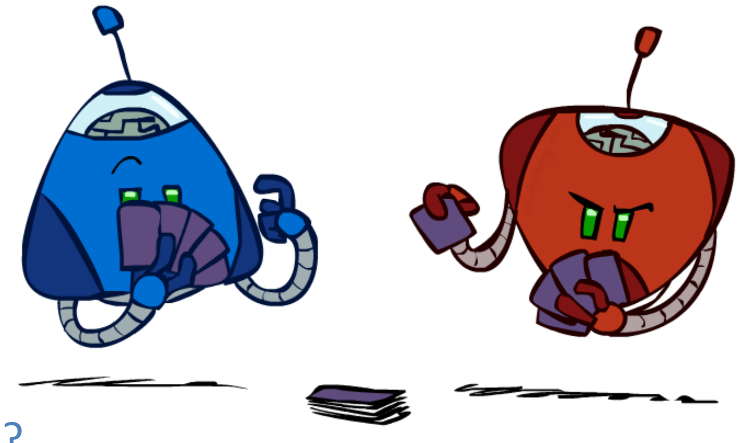
Takeshi Murakami
World Othello Champion

1997: The Logistello software crushed Murakami by 6 games to 0

Easy game? Try it: http://www.othelloonline.org/

Types of Games

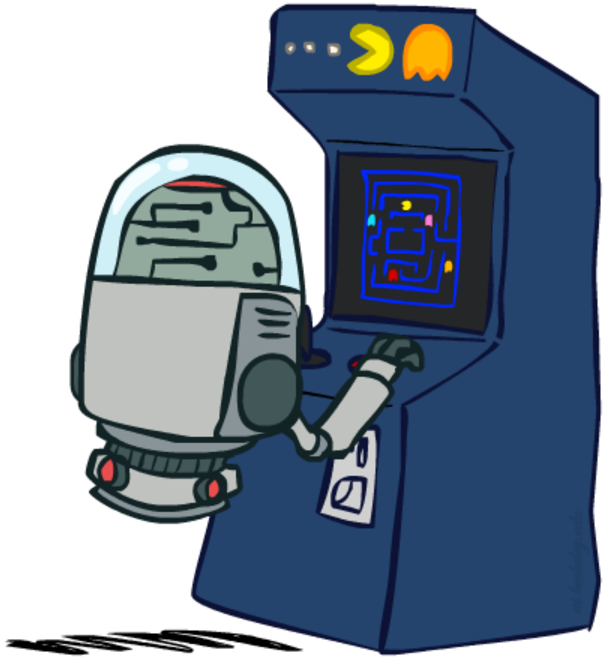
- Many different kinds of games!
- Dimensions:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?
- Want algorithms for calculating a strategy (policy) which recommends a move from each state



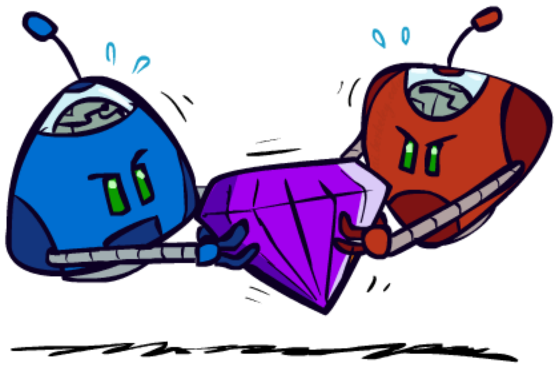
Deterministic Games

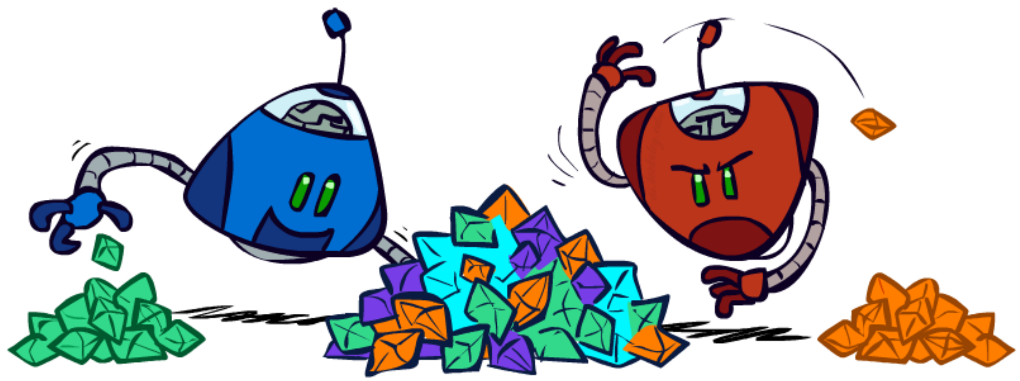
- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: $S \times A \rightarrow S$
  - Terminal Test: $S \rightarrow \{t,f\}$
  - Terminal Utilities: $S \times P \rightarrow R$

- Solution for a player is a policy: $S \rightarrow A$



Zero-Sum Games

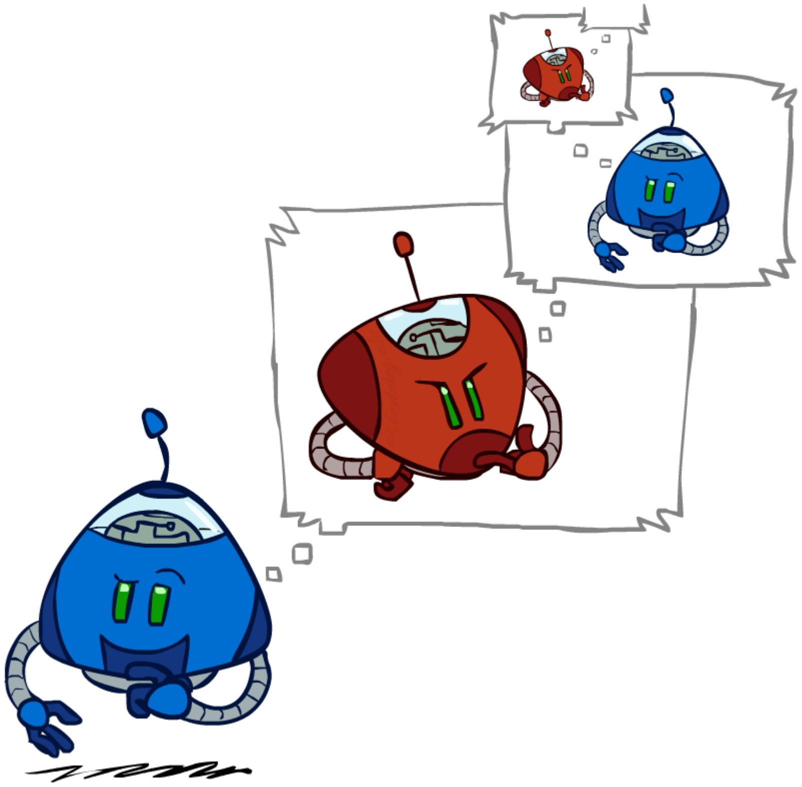




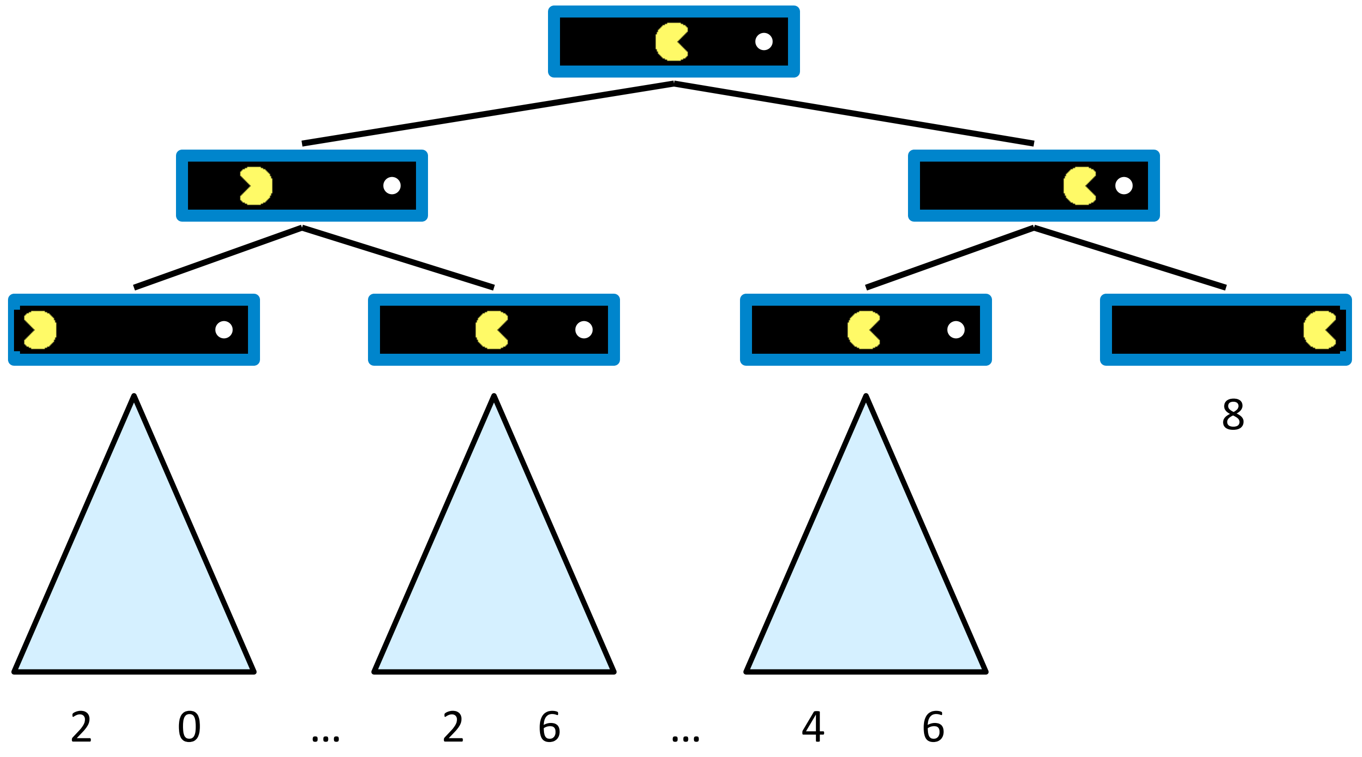
- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition

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

Adversarial Search



Single-Agent Trees

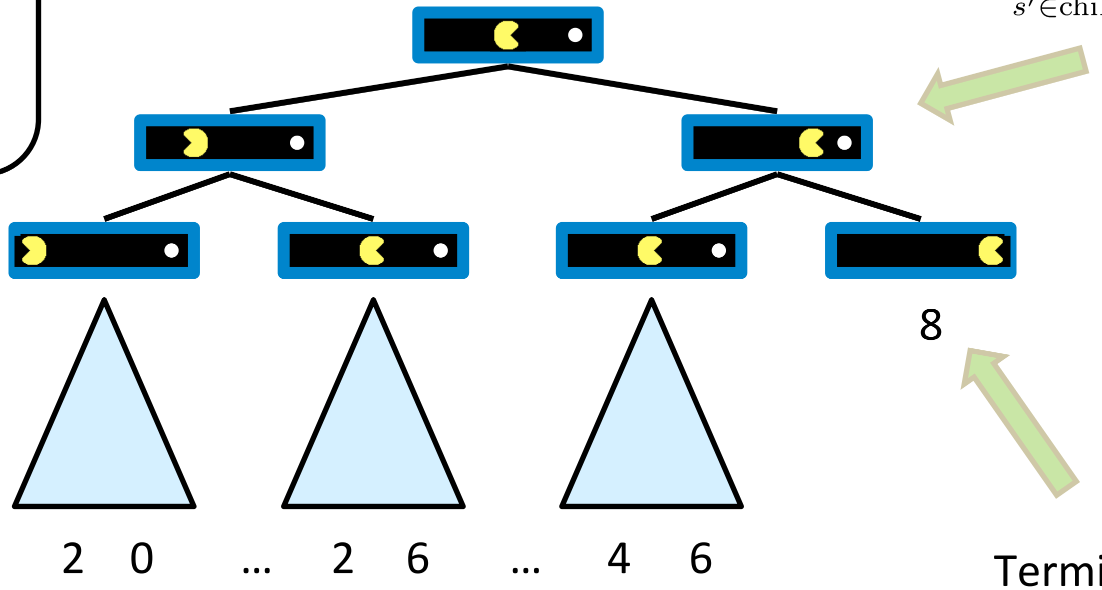


Value of a State

Value of a state: The best achievable outcome (utility) from that state

Non-Terminal

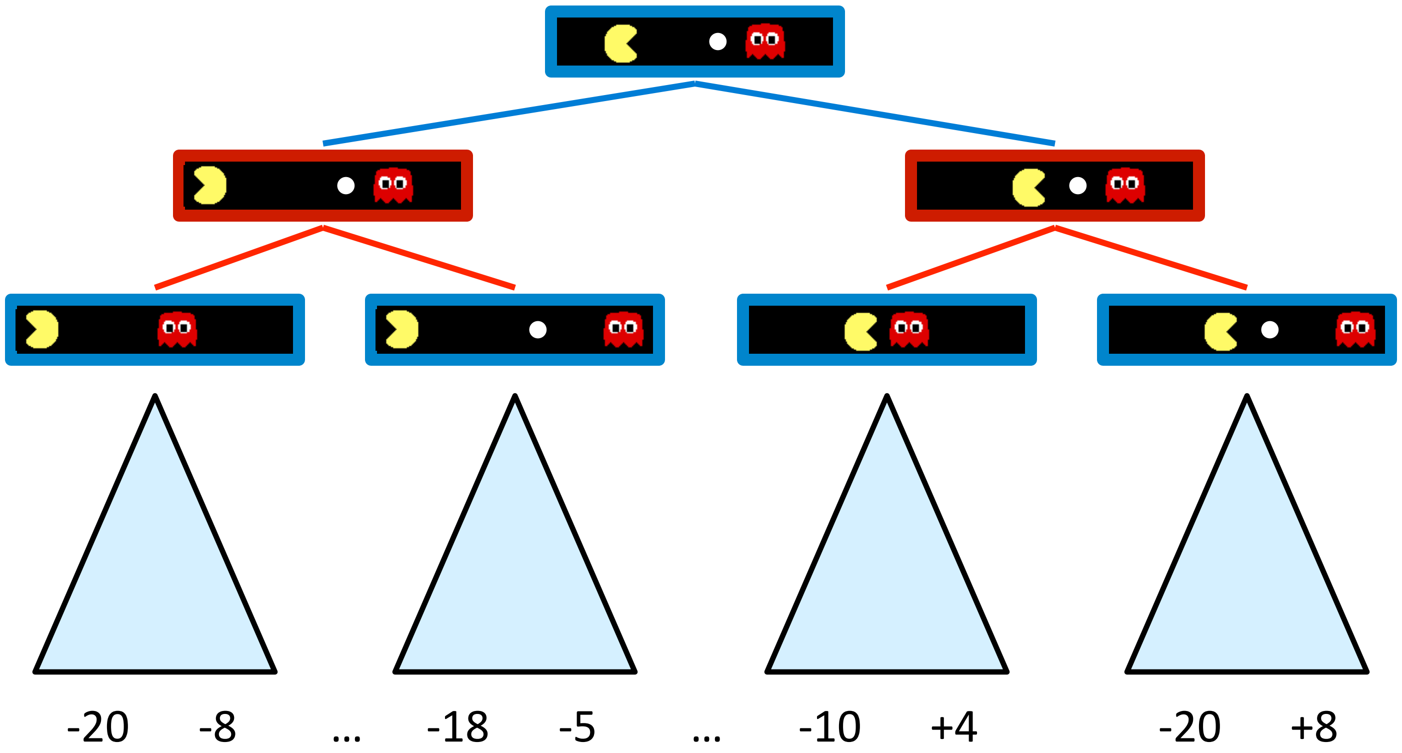
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Terminal States:

$$V(s) = \text{known}$$

Adversarial Game Trees

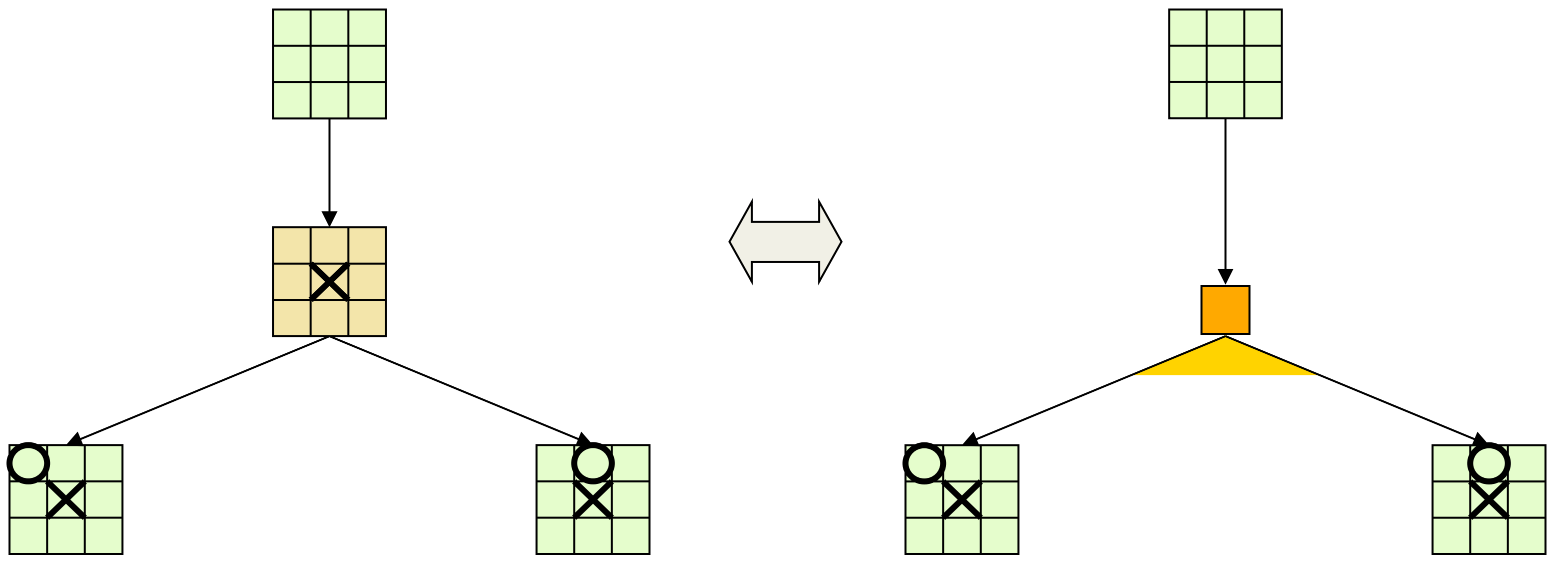


Setting: Two-player, turn-taking, deterministic, fully observable, zero-sum, time-constrained game

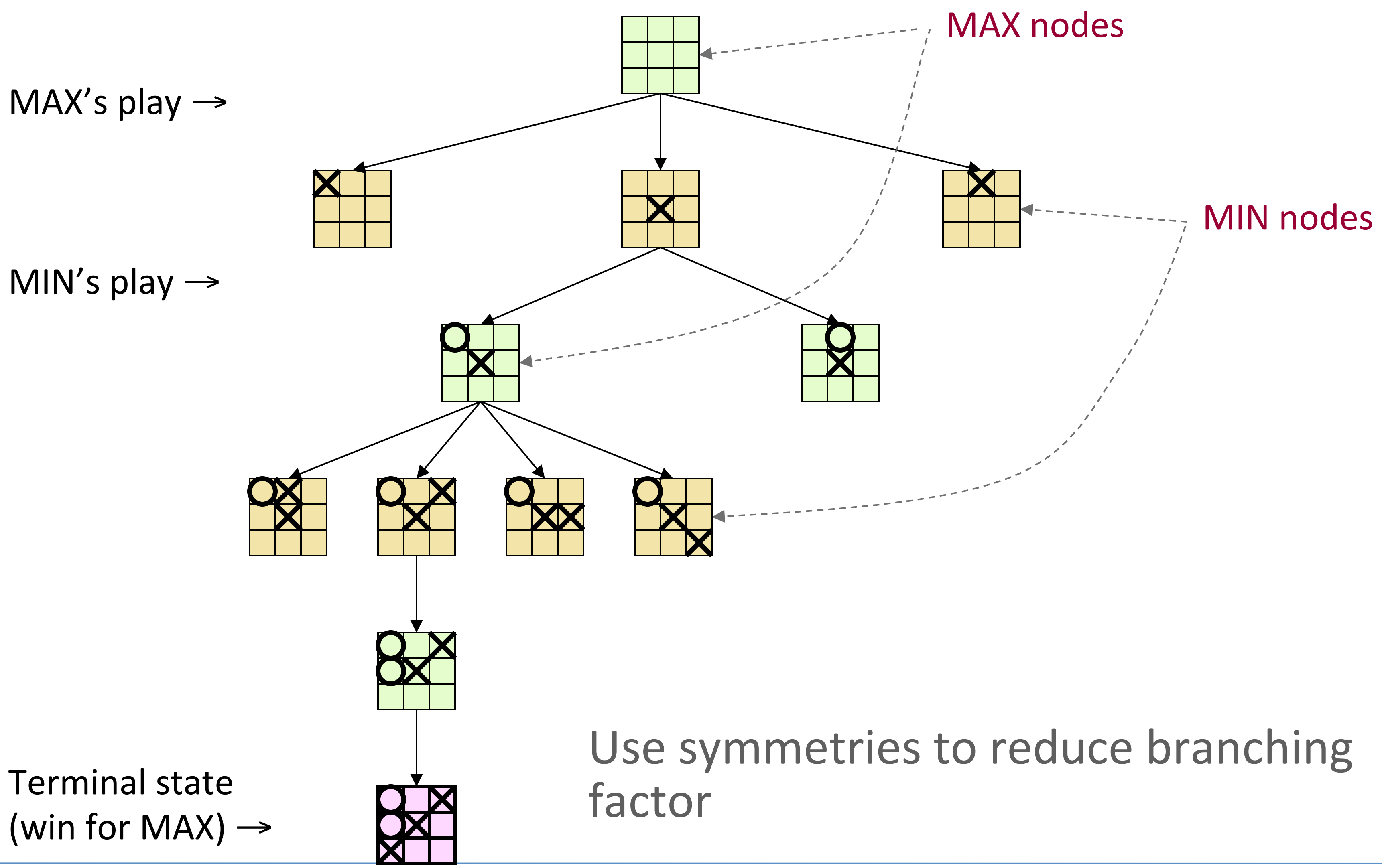
- State space
- Initial state
- Successor function: it tells which actions can be executed in each state and gives the successor state for each action
- MAX's and MIN's actions alternate, with MAX playing first in the initial state
- Terminal test: it tells if a state is terminal and, if yes, if it's a win or a loss for MAX, or a draw
- All states are fully observable

Why This Search is Harder?

- **Uncertainty** caused by the actions of another agent (MIN), who competes with our agent (MAX)

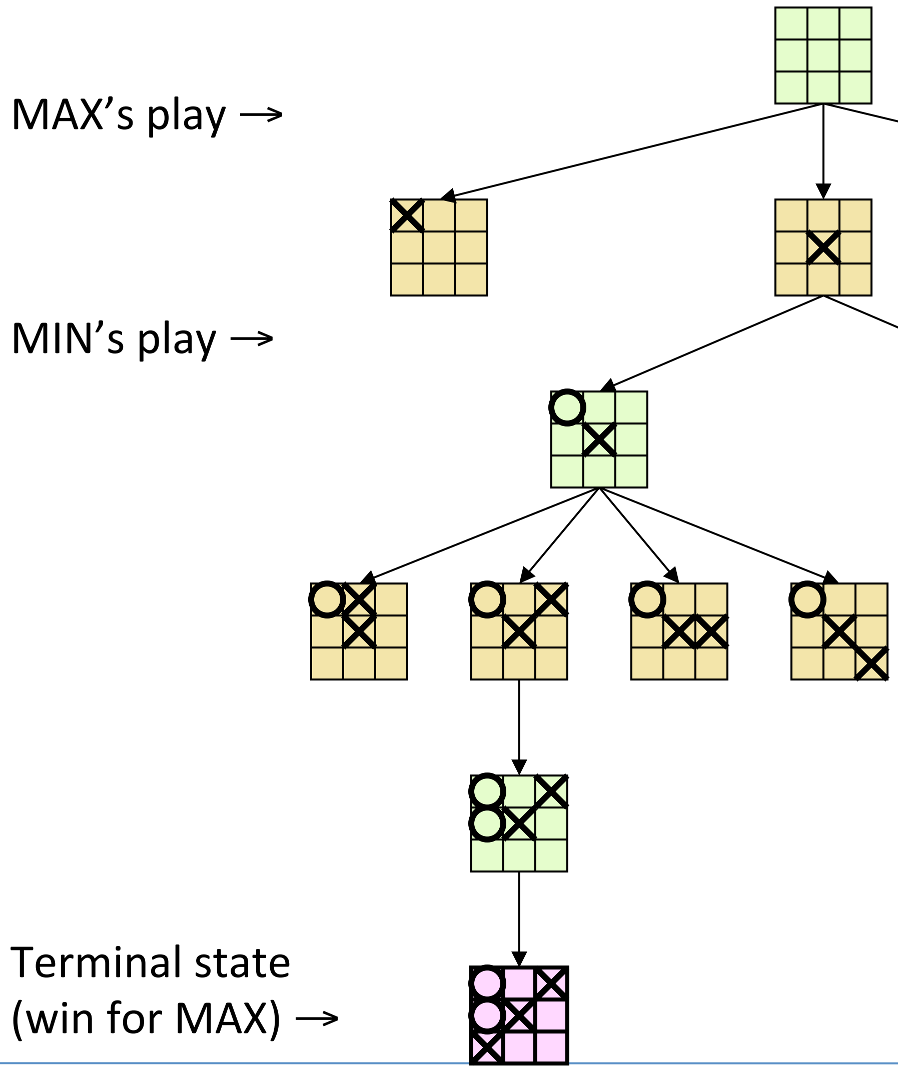


Game Tree



Use symmetries to reduce branching factor

Game Tree



In general, the branching factor and the depth of terminal states are large

Chess:

- Number of states: ~10^{40}
- Branching factor: ~35
- Number of total moves in a game: ~100

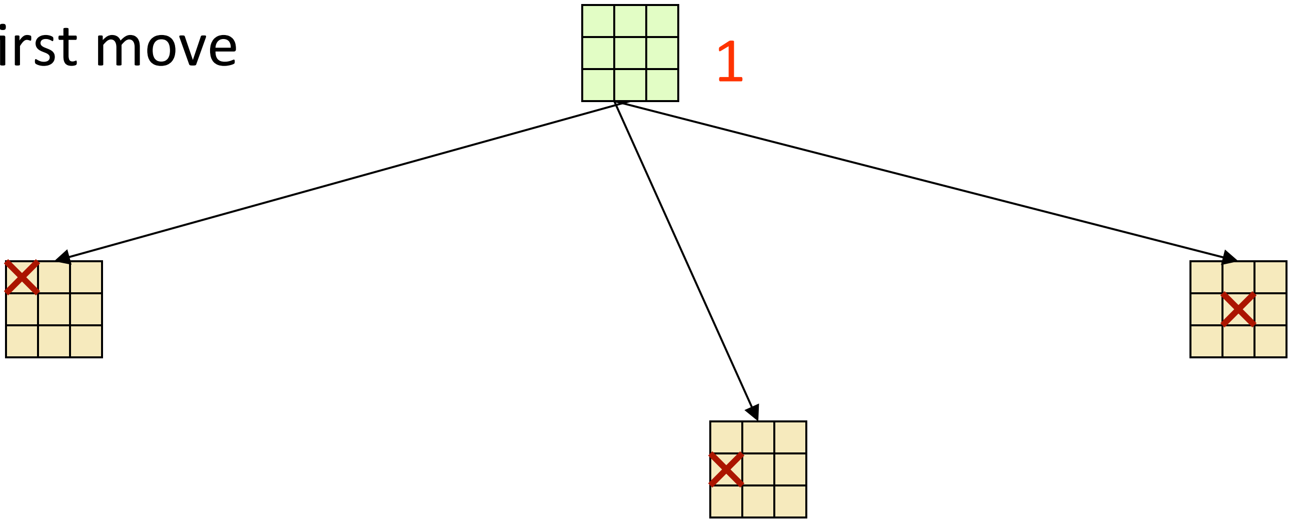
Choosing an Action: Basic Idea

1) Using the current state as the initial state, build the game tree uniformly to the maximal depth h (called horizon) feasible within the time limit
2) Evaluate the states of the leaf nodes
3) Back up the results from the leaves to the root and pick the best action assuming the worst (for us) from MIN

→ Minimax algorithm

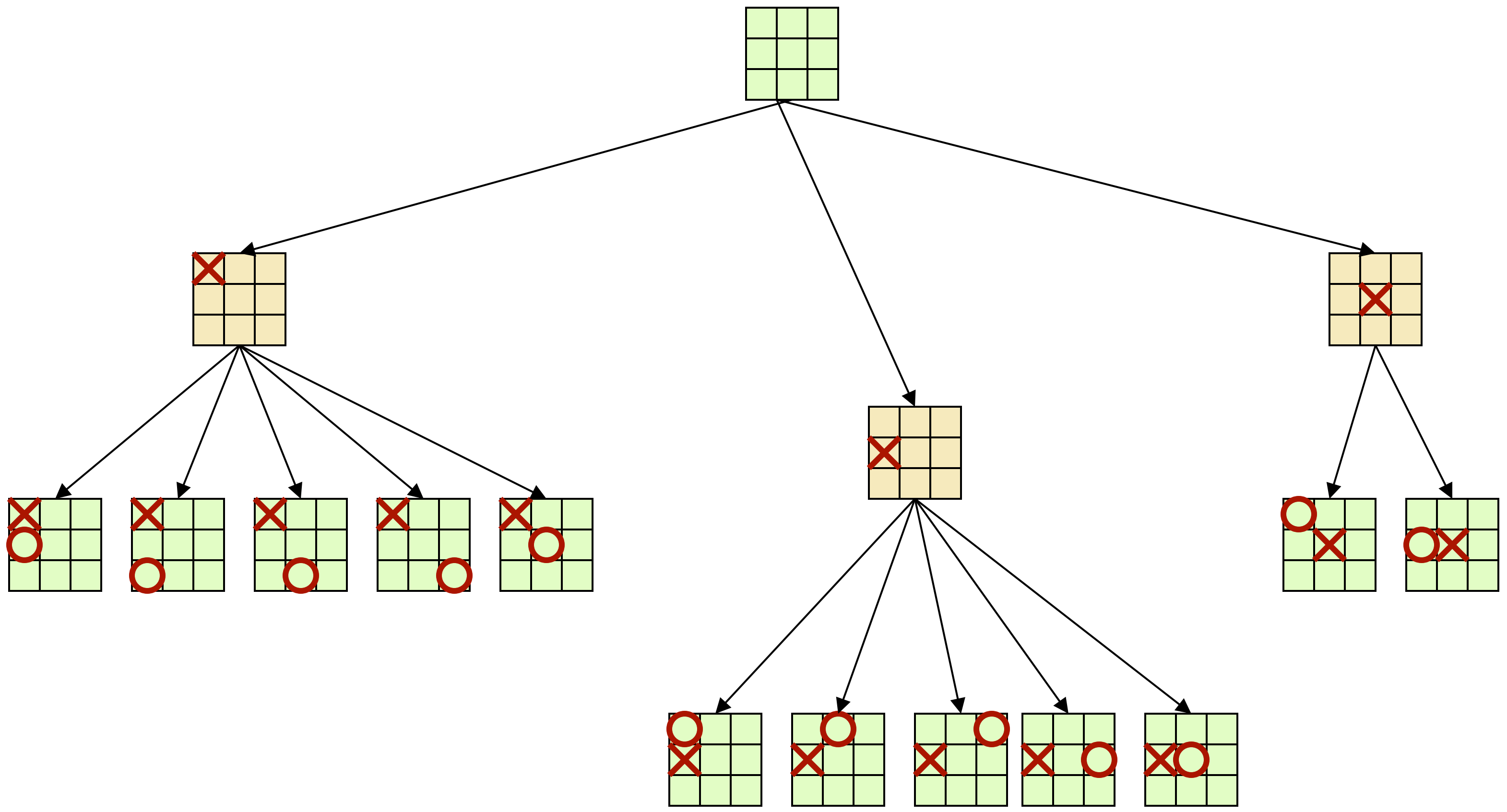
Propagating e-values to root (first move)

MAX first move



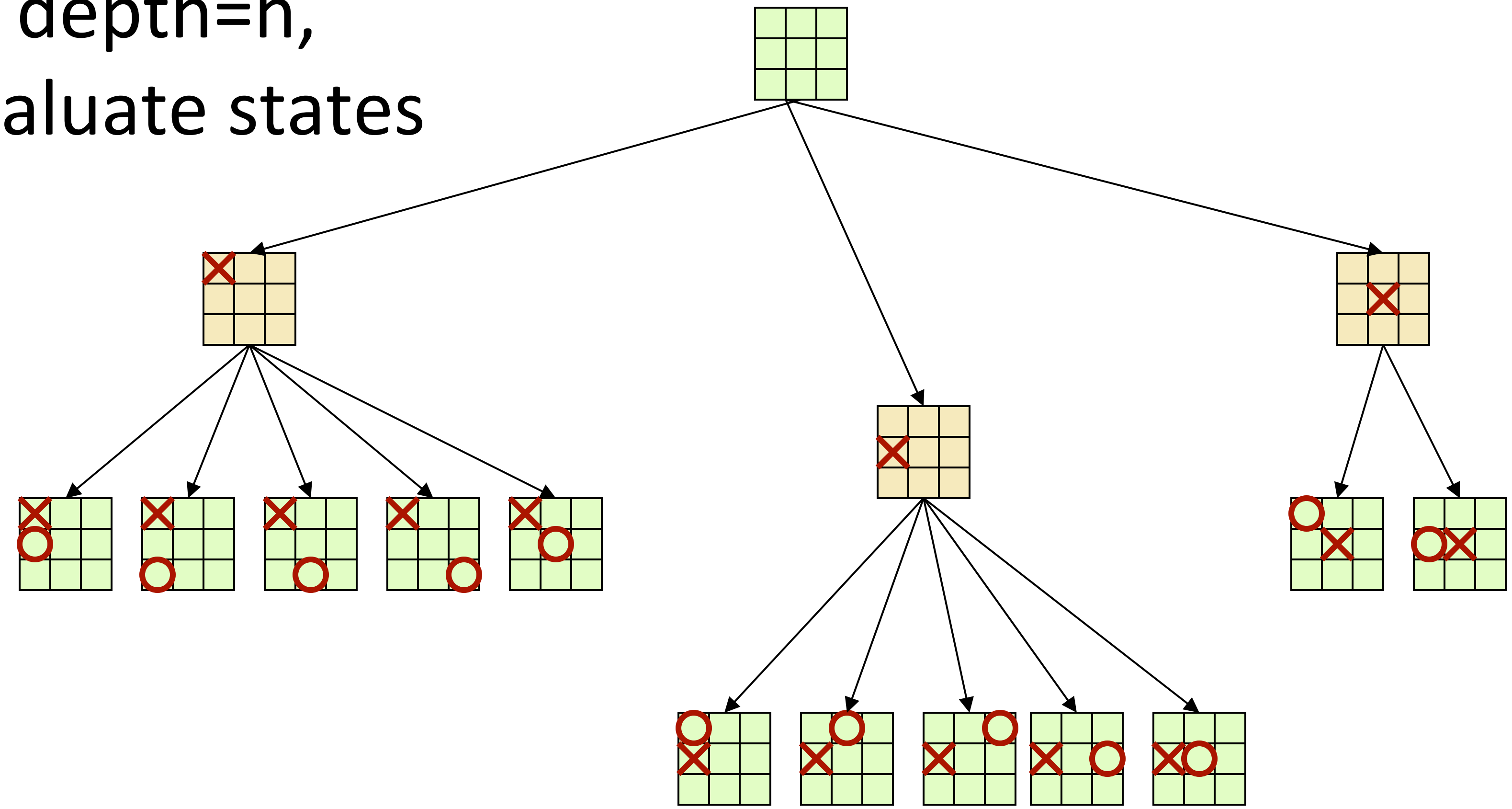
e(s) = (#rows + #columns + #diagonals for MAX)
- (#rows+#columns+#diagonals for MIN)

Propagating e-values to root (first move)



Propagating e-values to root (first move)

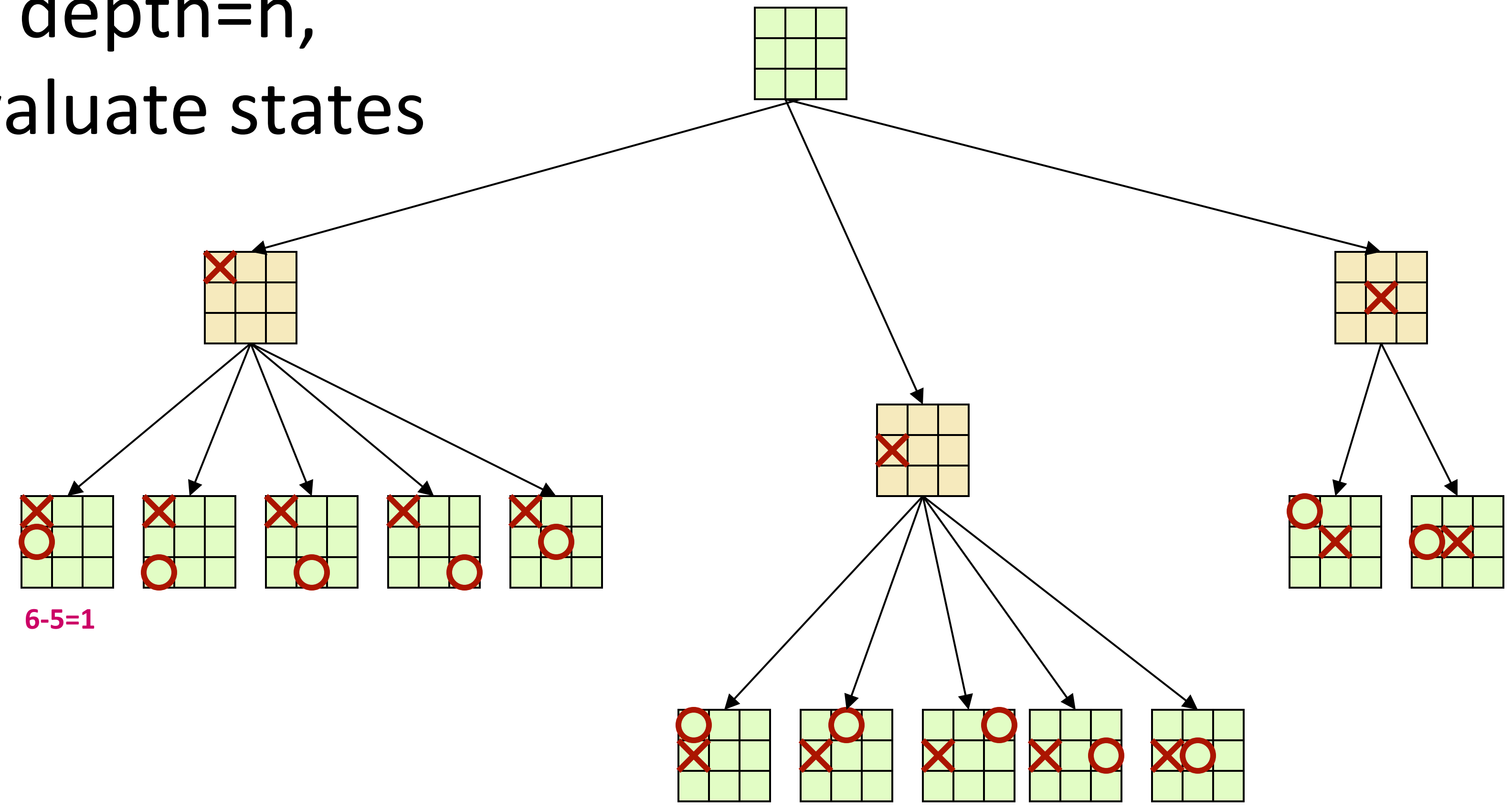
- At depth=h, evaluate states



e(s) = (#row + #col + #diag for MAX)- (#row+#col+#diag for MIN)

Propagating e-values to root (first move)

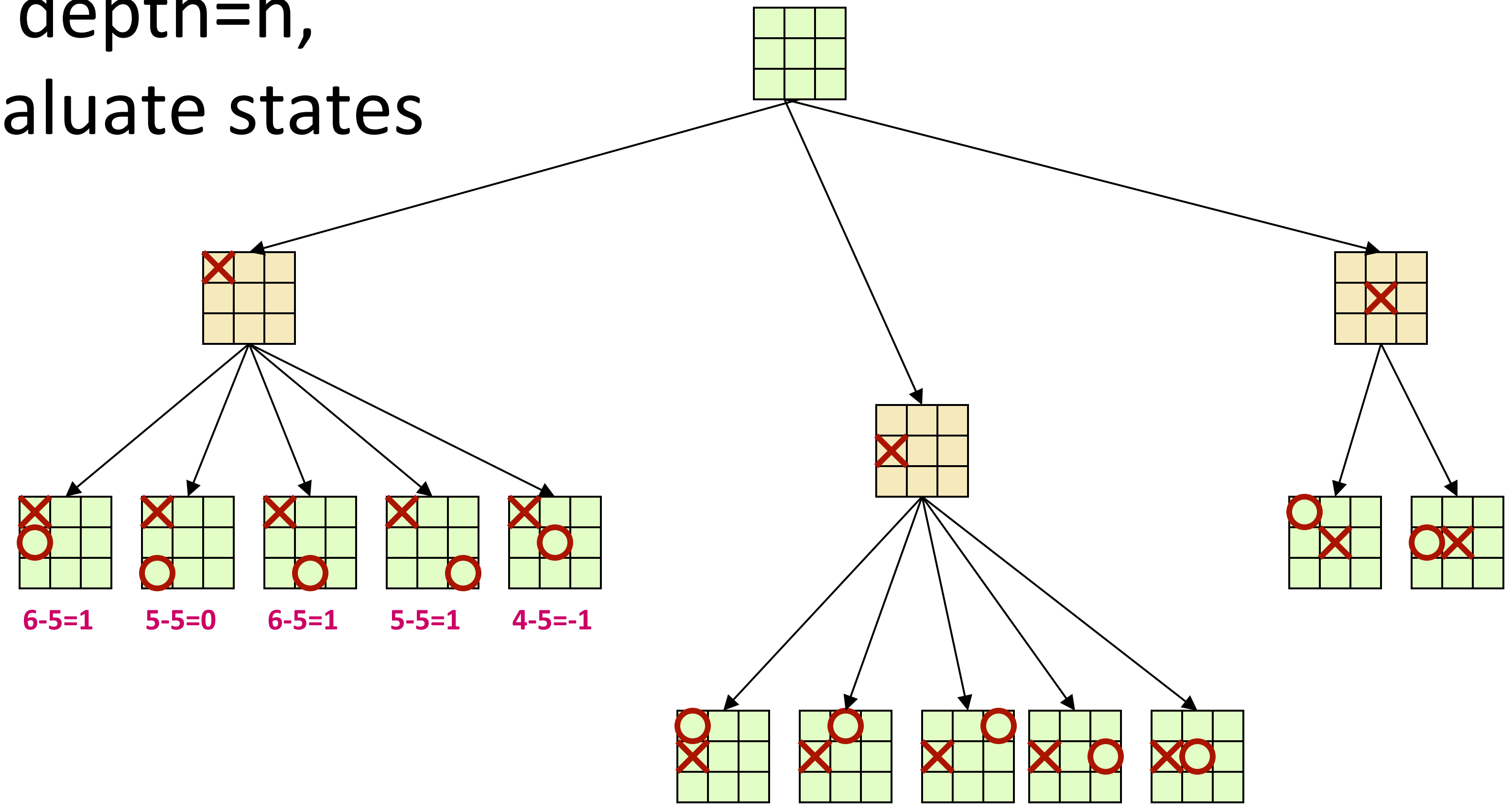
- At depth=h, evaluate states



e(s) = (#row + #col + #diag for MAX)- (#row+#col+#diag for MIN)

Propagating e-values to root (first move)

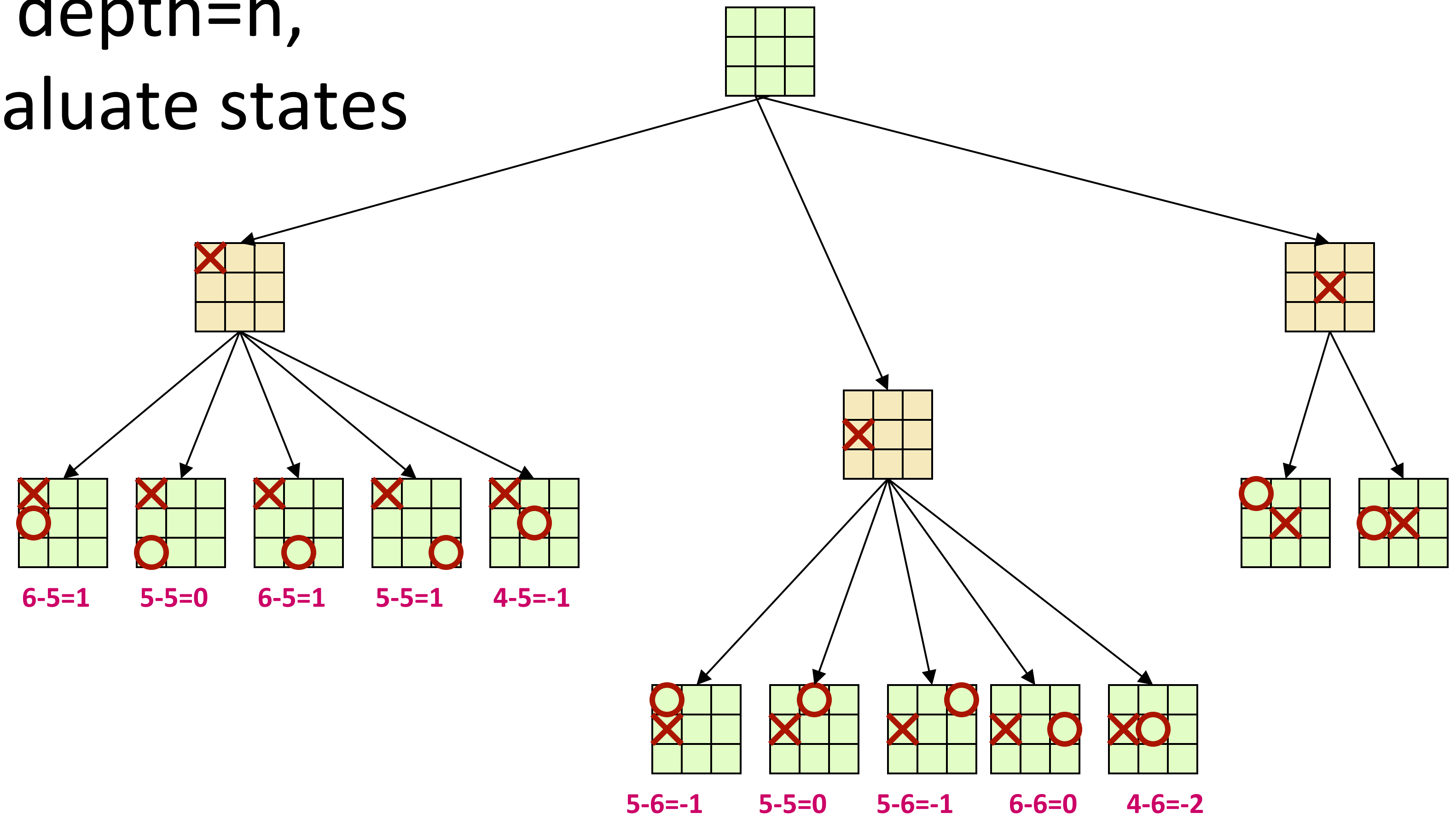
- At depth=h, evaluate states



e(s) = (#row + #col + #diag for MAX)- (#row+#col+#diag for MIN)

Propagating e-values to root (first move)

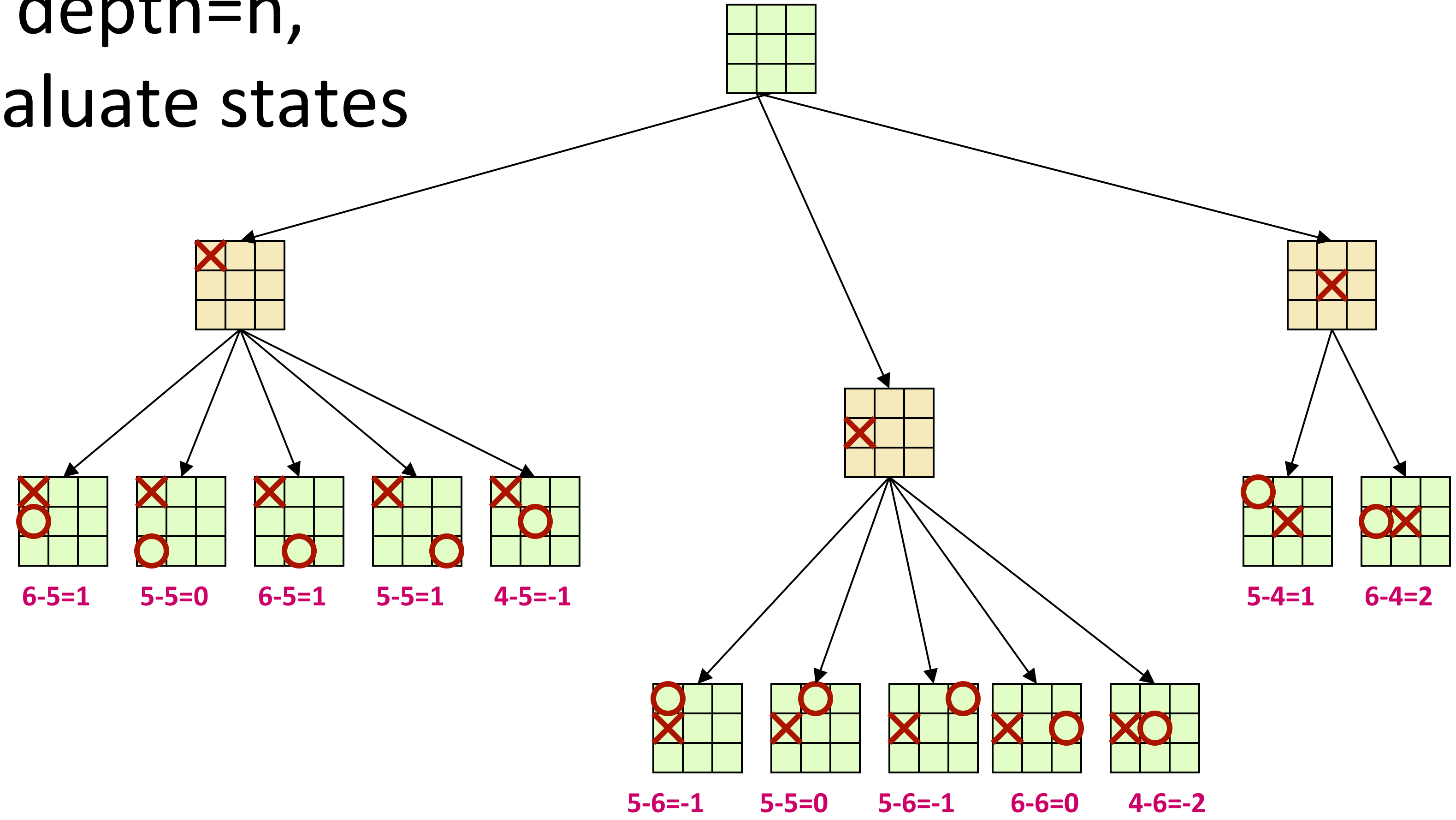
- At depth=h, evaluate states



e(s) = (#row + #col + #diag for MAX)- (#row+#col+#diag for MIN)

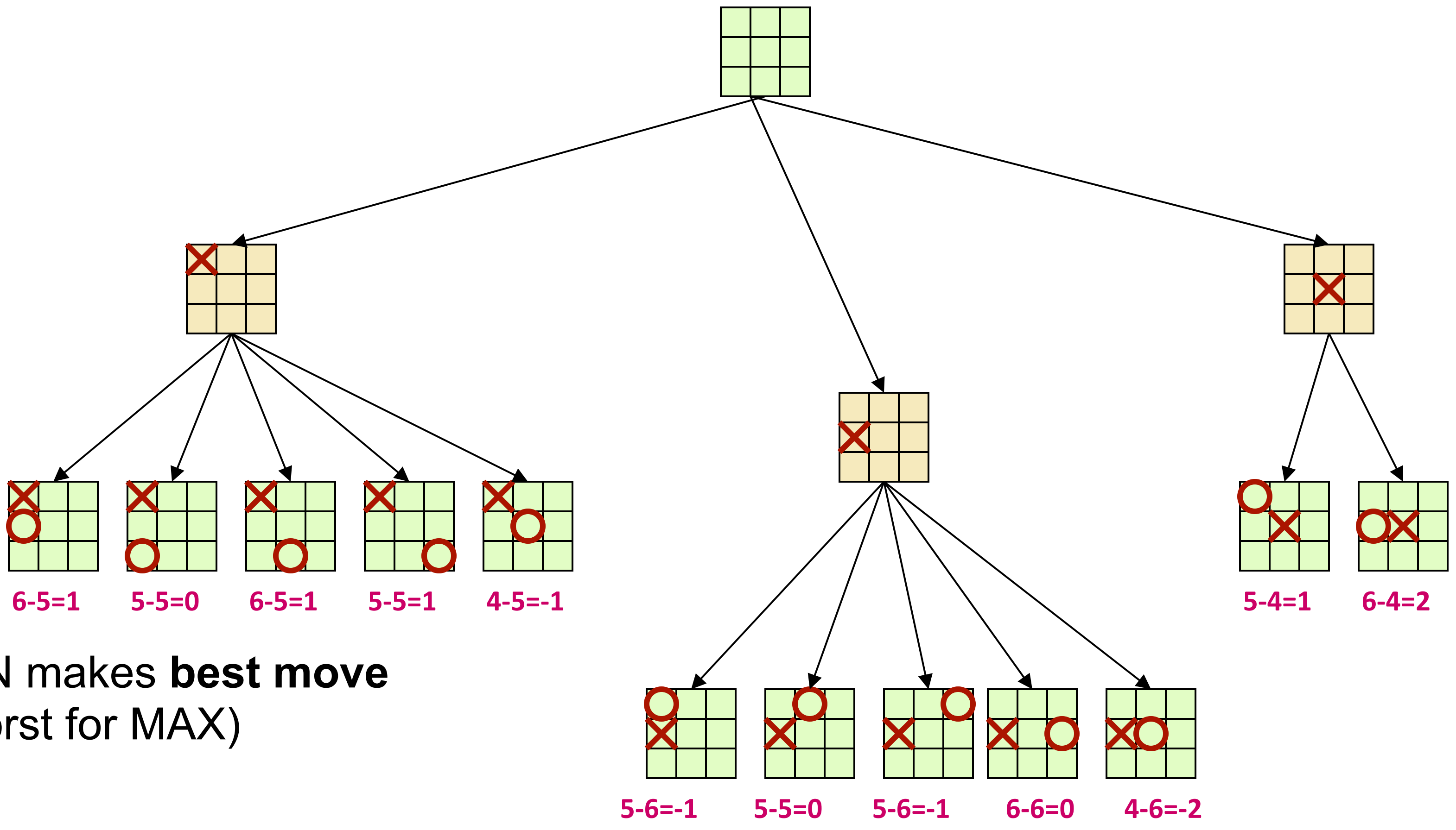
Propagating e-values to root (first move)

- At depth=h, evaluate states



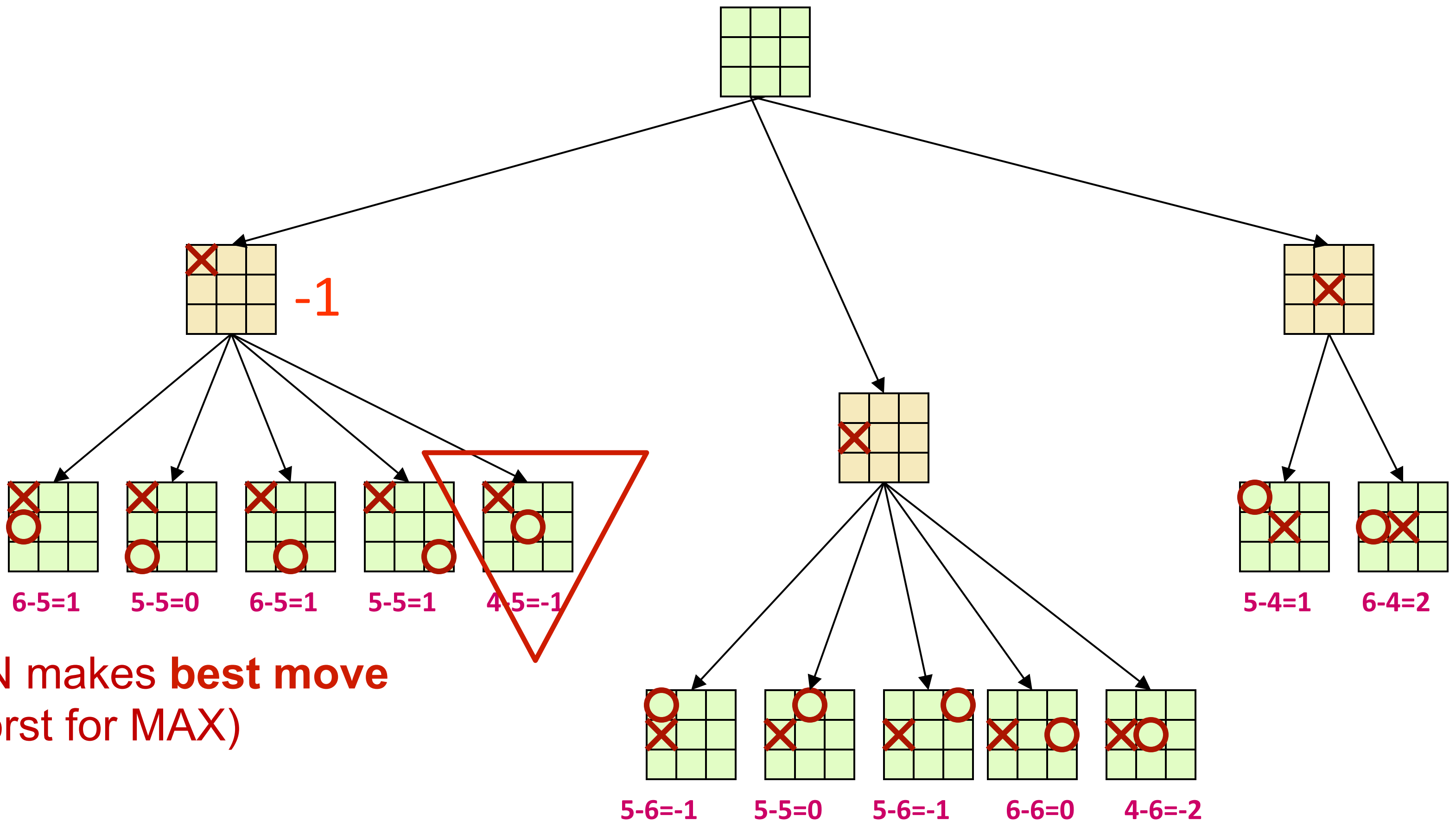
e(s) = (#row + #col + #diag for MAX)- (#row+#col+#diag for MIN)

Propagating e-values to root (first move)



MIN makes **best move**
(worst for MAX)

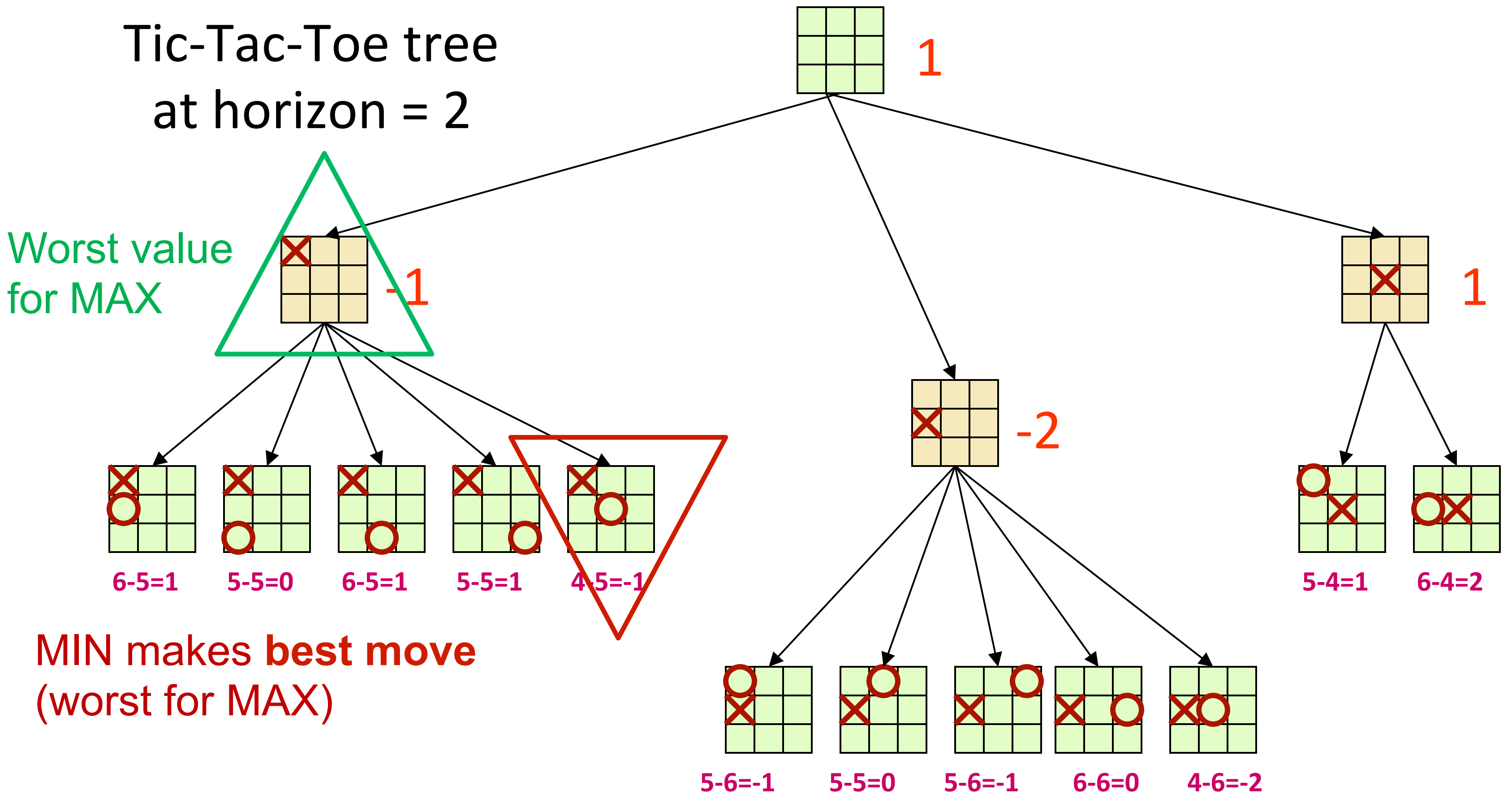
Propagating e-values to root (first move)



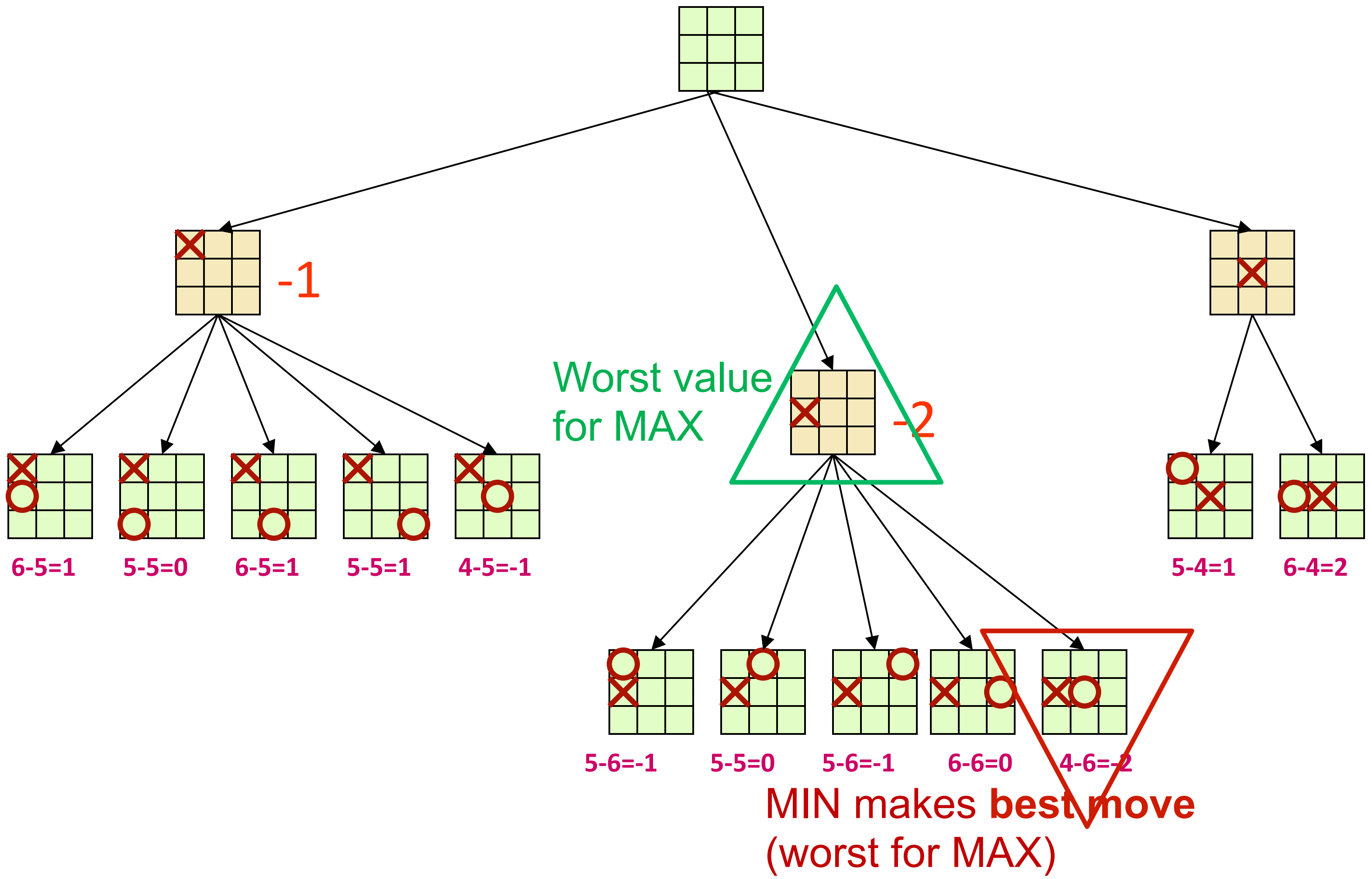
MIN makes **best move** (worst for MAX)

Propagating e-values to root (first move)

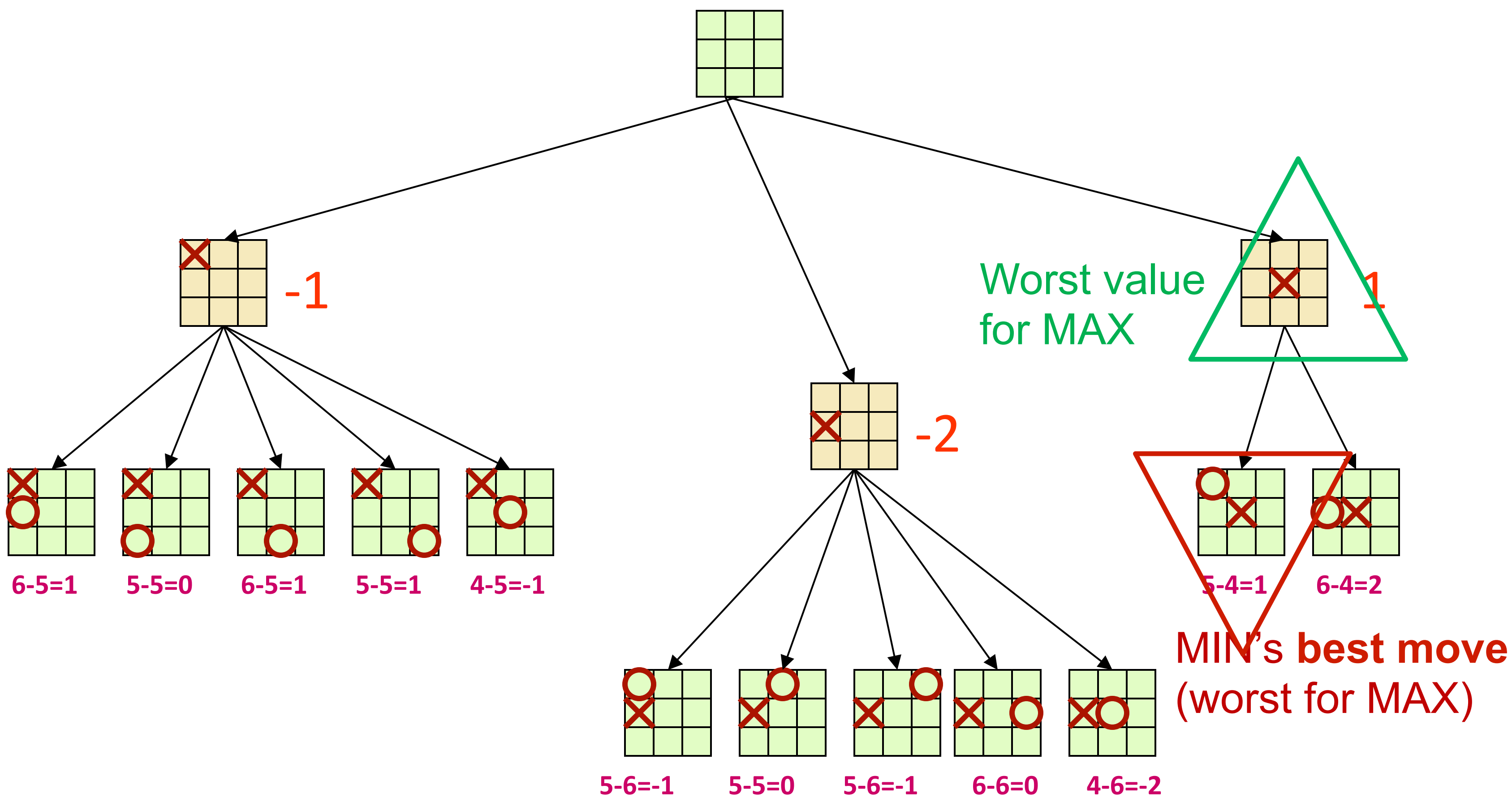
Tic-Tac-Toe tree
at horizon = 2



Propagating e-values to root (first move)

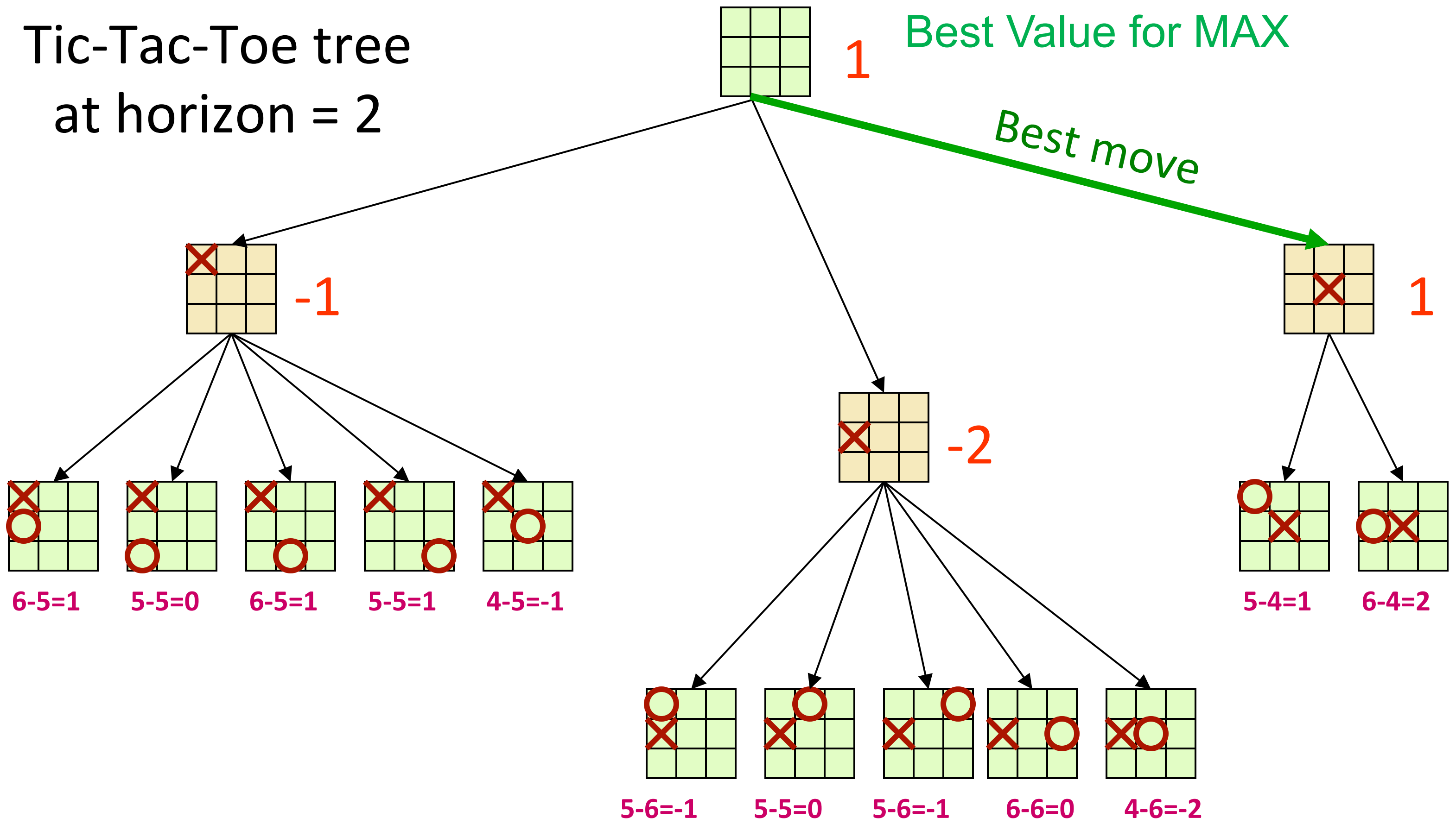


Propagating e-values to root (first move)

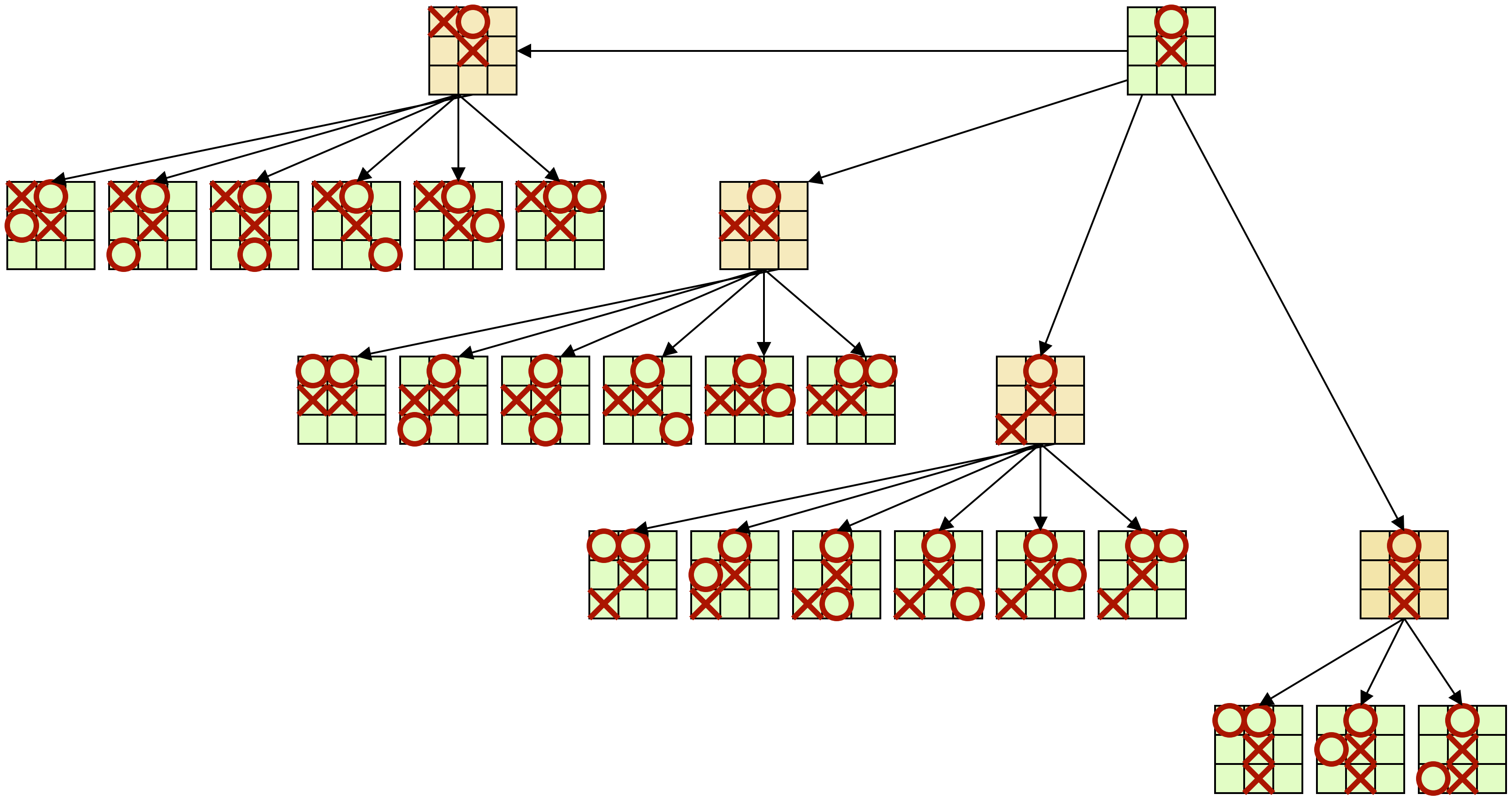


Propagating e-values to root (first move)

Tic-Tac-Toe tree
at horizon = 2

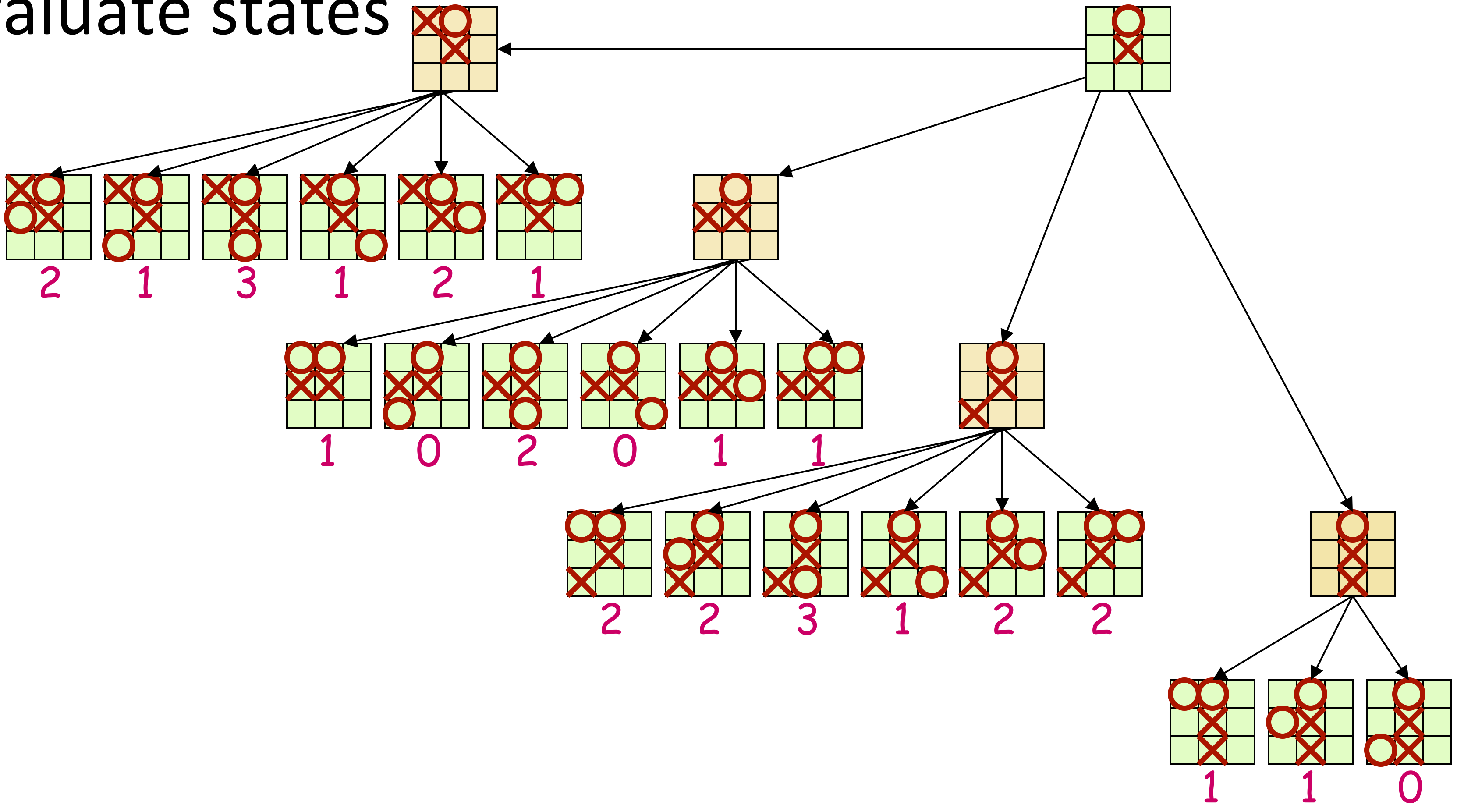


Propagating values (next move)

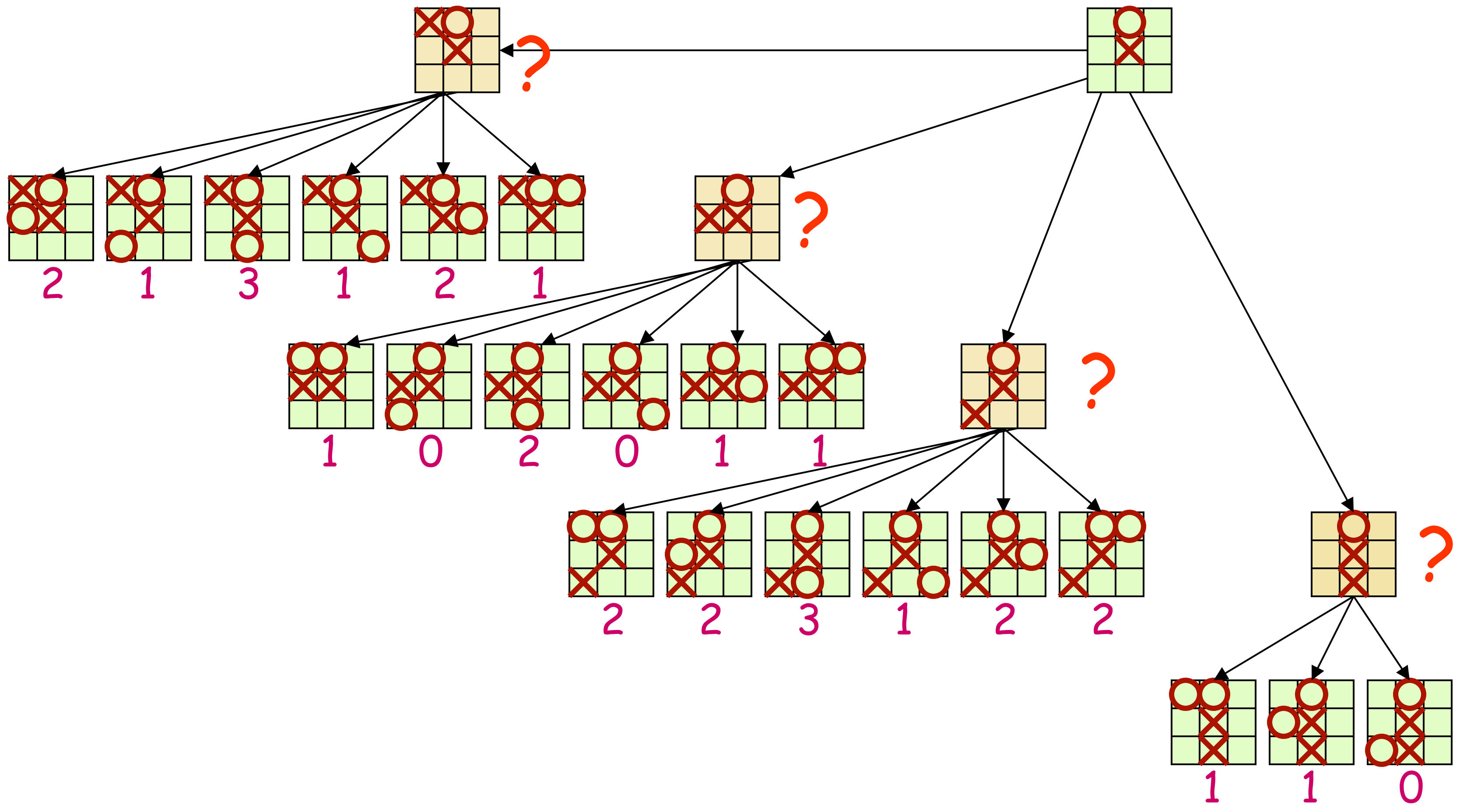


Propagating values (next move)

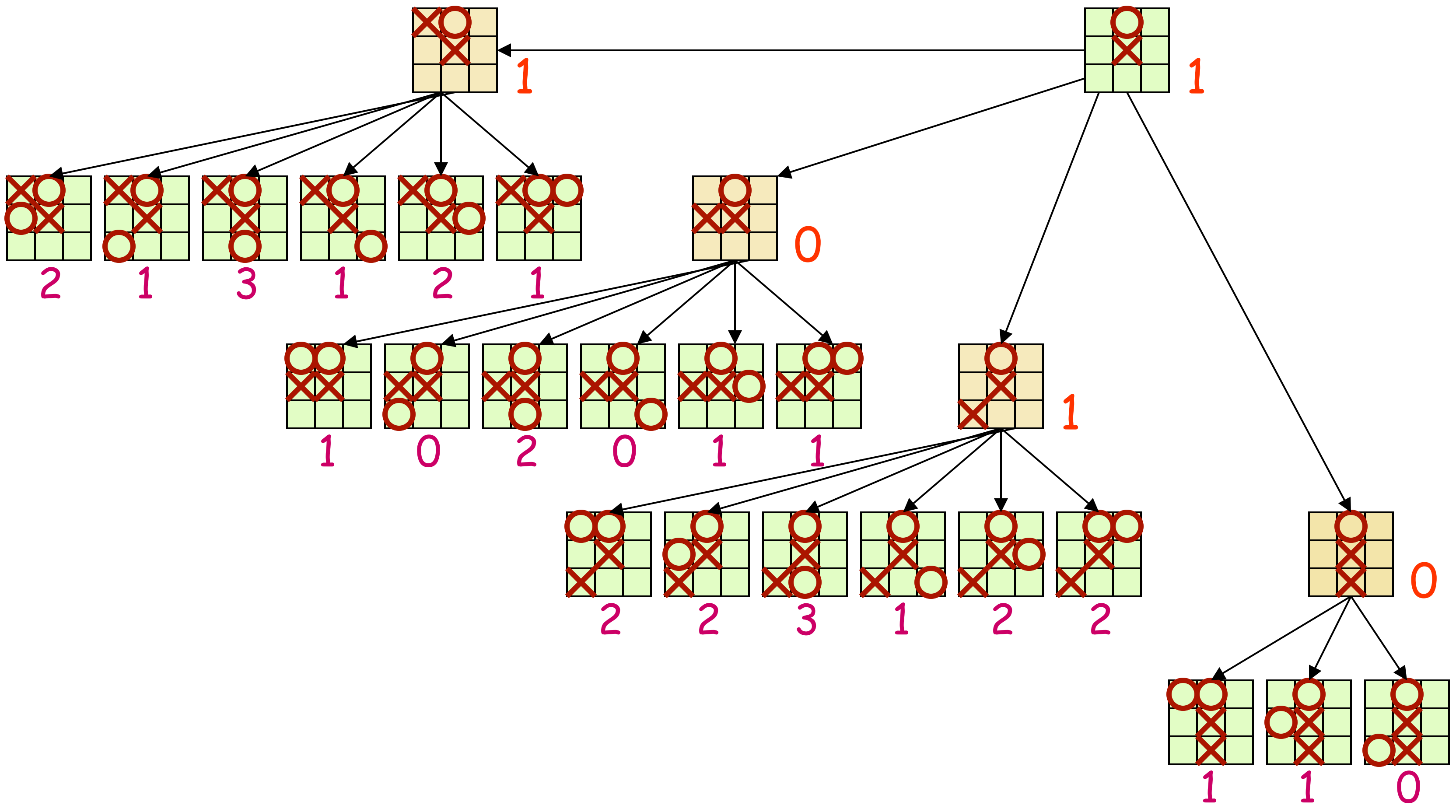
- Evaluate states



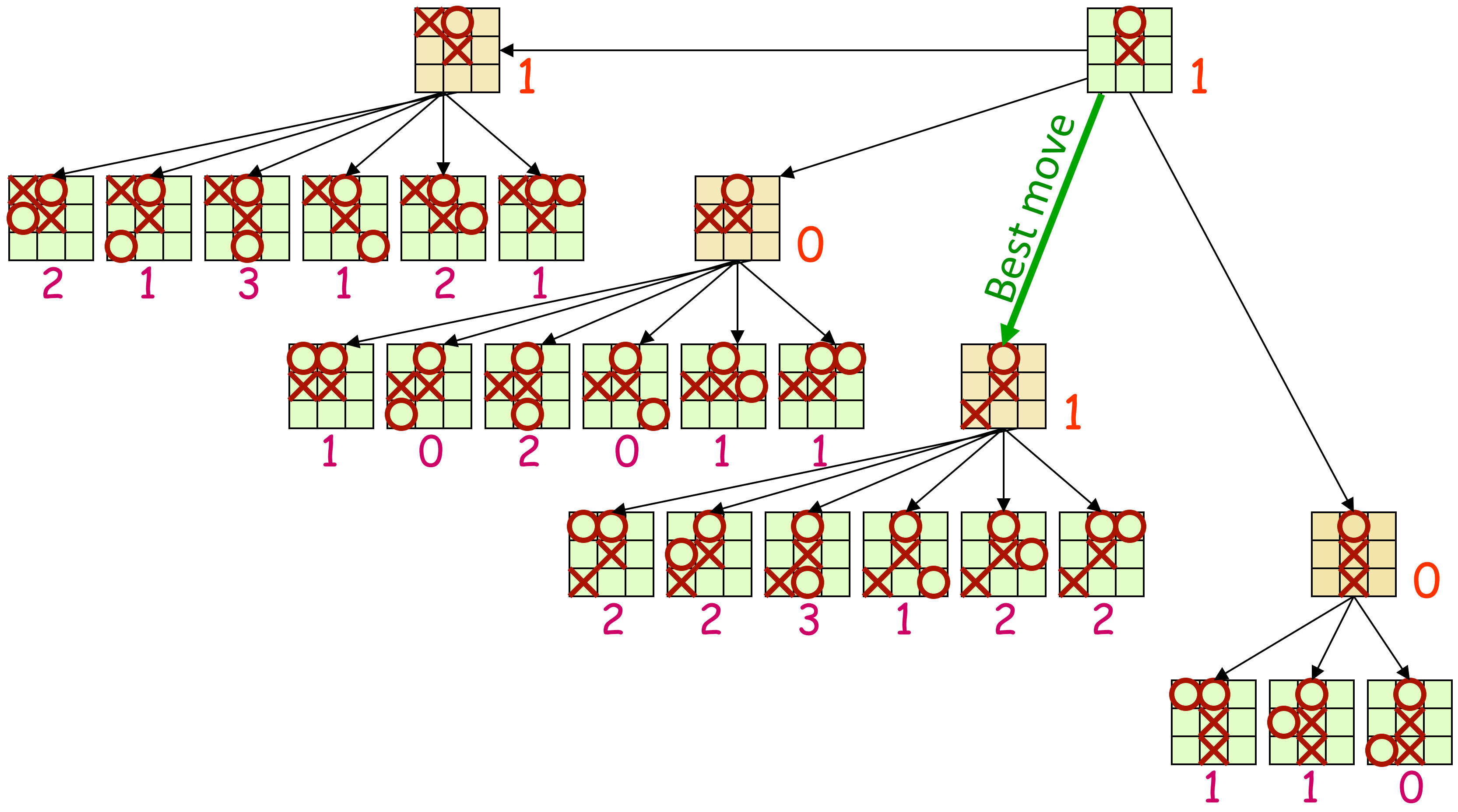
Propagating values (next move)



Propagating values (next move)



Propagating values (next move)



Why use propagated values instead of original *e*?

- At each non-leaf node N, the value is the highest value of the best state that MAX can reach at depth h, if MIN plays well (by the same criterion as MAX applies to itself)
- Evaluation function ***e*** (estimate) improves closer to terminal (win/lose) states
- If ***e*** is to be trusted in the first place, then the **backed-up value is a better estimate** of how favorable STATE(N) is than ***e***(STATE(N))

Minimax Algorithm

1. On each turn, expand the game tree uniformly (**BFS**) from the current state to depth **h**
2. Compute the evaluation function at **every leaf** of the tree
3. Back-up (propagate) the values from the leaves to the root of the tree as follows:
   a. A MAX node gets the maximum of the evaluation of its successors
   b. A MIN node gets the minimum of the evaluation of its successors
4. Select the move toward a MIN node that has the largest backed-up value

Minimax Algorithm: horizon

1. Expand the game tree uniformly from the current state (where it is MAX's turn to play) to depth **h**
2. Compute the evaluation function at every leaf of the tree
3. Back-up (propagate) the values from the leaves to the root of the tree as follows:
   a. A MAX node gets the maximum of the evaluation of its successors
   b. A MIN node gets the minimum of the evaluation of its successors
4. Select the move toward a MIN node that has the largest backed-up value

H=Horizon: Needed to return a decision within allowed time

Game Playing (for MAX)

Repeat until a terminal state is reached

1. Select move using Minimax
2. Execute move
3. Observe MIN's move

Note that at each cycle the large game tree built to horizon h is used to select **only one move**

All is repeated again at the next cycle **(a sub-tree of depth h-2 can be re-used)**

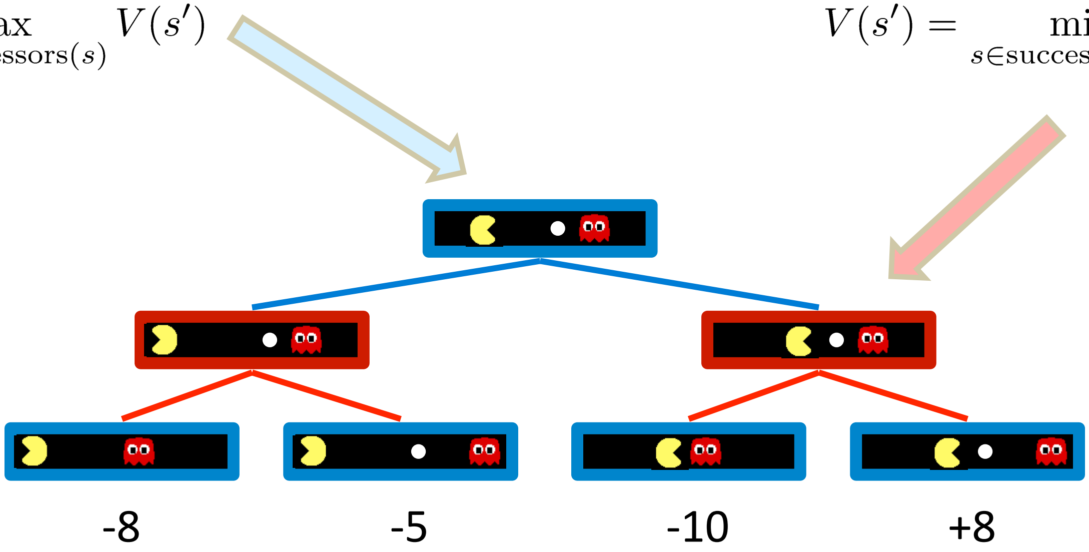
Minimax Values

States Under Agent's Control:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent's Control:

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

$$V(s) = \text{known}$$

Minimax Implementation

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

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

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

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Implementation (Dispatch)

```
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 MIN: return min-value(state)
```

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

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

Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
   inputs: state, current state in game
   v←MAX-VALUE(state)
   return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow \infty$
   for a,s in SUCCESSORS(state) do
      $v \leftarrow$ MAX(v,MIN-VALUE(s))
   return v
```

```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow \infty$
   for a,s in SUCCESSORS(state) do
      $v \leftarrow$ MIN(v,MAX-VALUE(s))
   return v
```

Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v,MIN-VALUE(s))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MIN(v,MAX-VALUE(s))
  return v
```

Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v,MIN-VALUE(s))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MIN(v,MAX-VALUE(s))
  return v
```

Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v←MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v,MIN-VALUE(s))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MIN(v,MAX-VALUE(s))
  return v
```

Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  $v \leftarrow$ MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value $v$
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow \infty$
  for a,s in SUCCESSORS(state) do
    $v \leftarrow$ MAX(v,MIN-VALUE(s))
  return $v$
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow \infty$
  $\boxed{\text{for } a,s \text{ in SUCCESSORS(}state\text{) do}}$
    $v \leftarrow \boxed{\text{MIN(}v\text{,MAX-VALUE(}s\text{))}}$
  return $v$
```

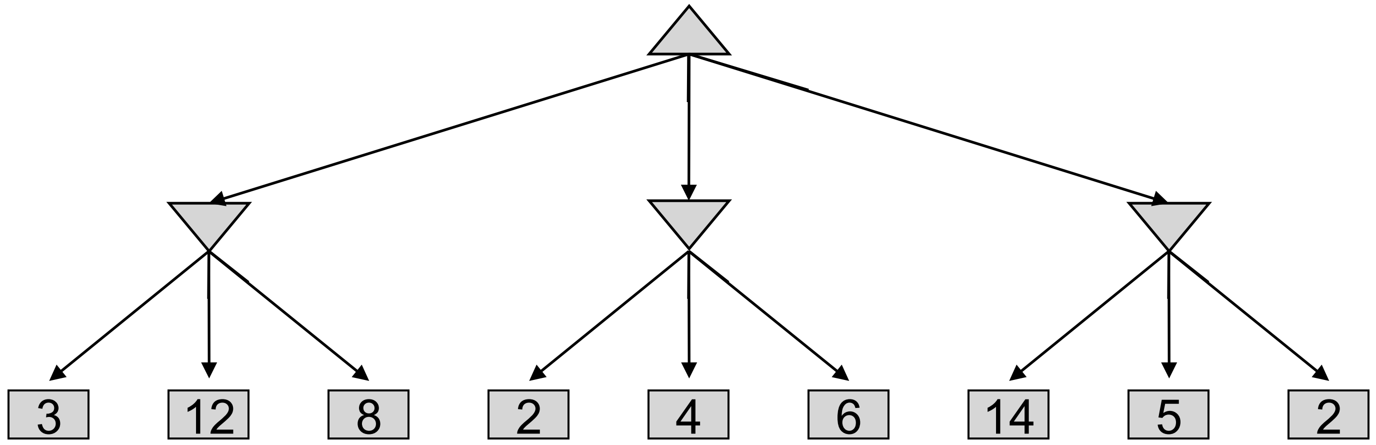
Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  $v \leftarrow$ MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value $v$
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow \infty$
  for a,s in SUCCESSORS(state) do
    $v \leftarrow$ MAX(v,MIN-VALUE(s))
  return $v$
```

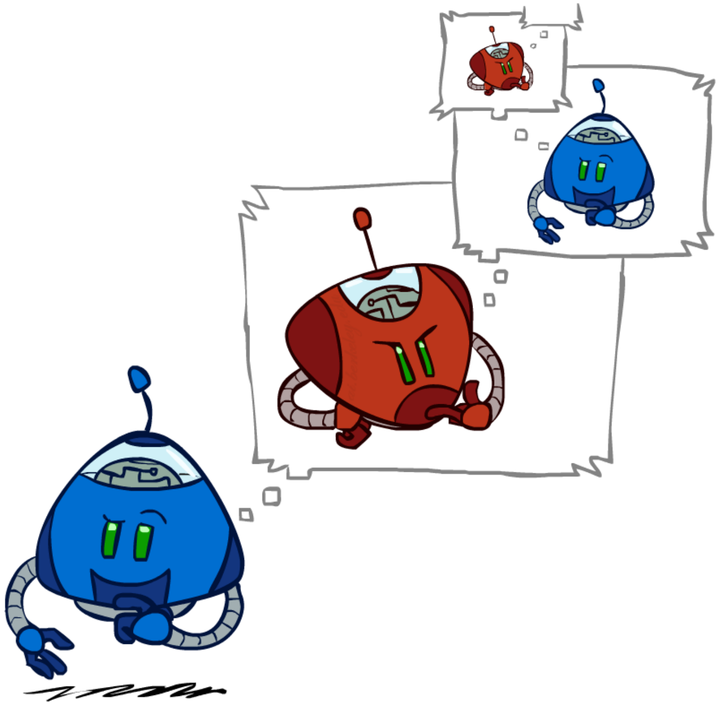
```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow \infty$
  for a,s in SUCCESSORS(state) do
    $v \leftarrow$ MIN(v,MAX-VALUE(s))
  return $v$
```

Minimax Example

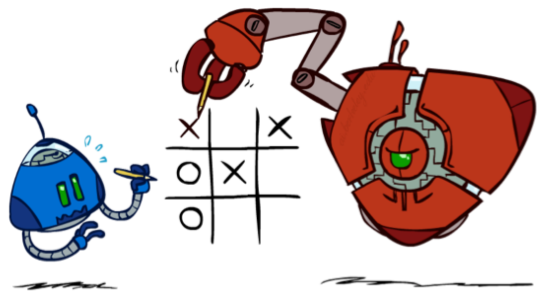


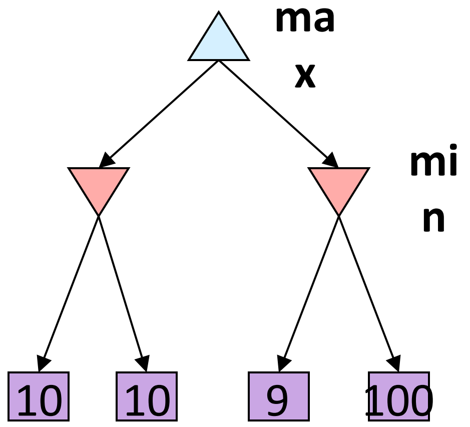
Minimax Efficiency

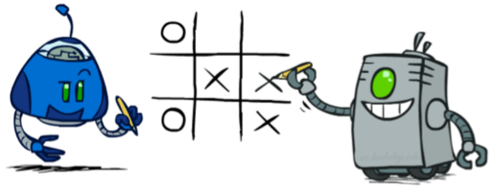
- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$
- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?



Minimax Properties





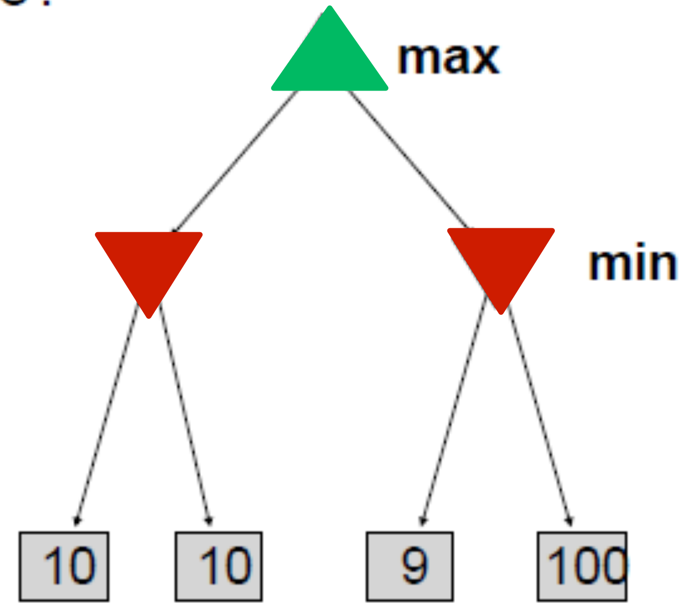


Optimal against a perfect player. Otherwise?

Demo: pacman suicide

Minimax Properties

- Optimal?
  - Yes, against perfect player. Otherwise?
- Time complexity?
  - $O(b^m)$
- Space complexity?
  - $O(bm)$
- For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?

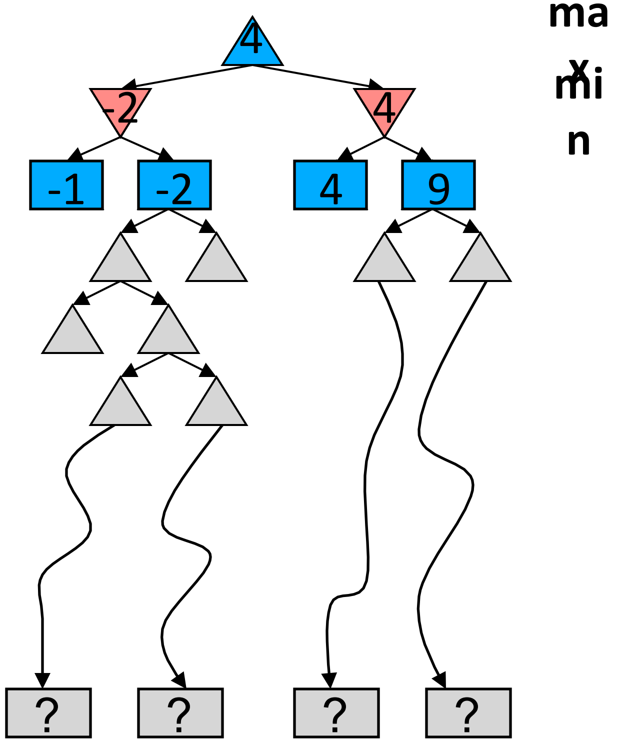


Video of Demo Min vs. Exp (Min)

Video of Demo Min vs. Exp (Exp)

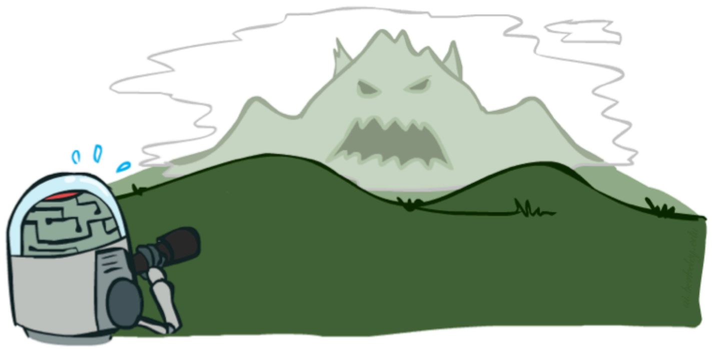
Resource Limits: Depth Limit

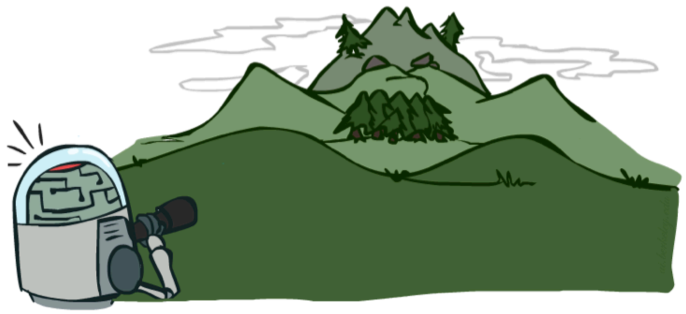
- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - 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



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





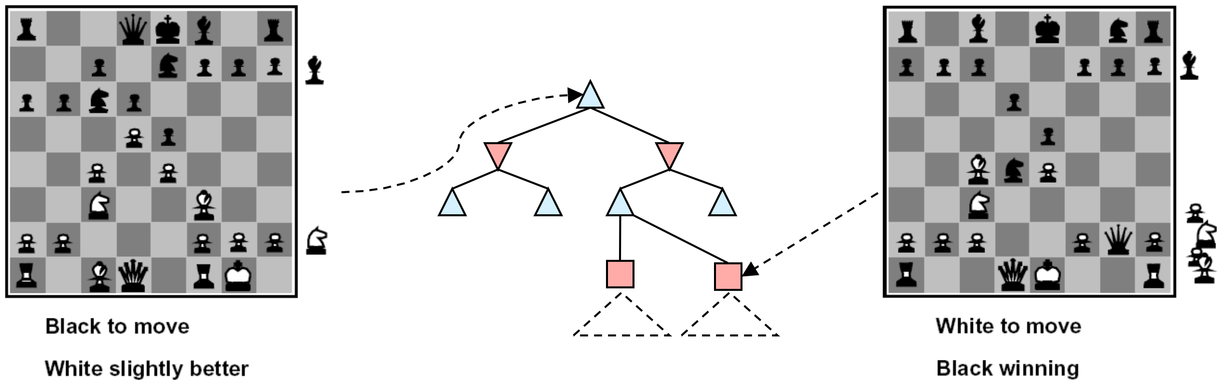
[Demo: depth limited Pacman]

Video of Demo Limited Depth (2)

Video of Demo Limited Depth (10)

Evaluation Functions

- Evaluation functions score non-terminals in depth-limited search



- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

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

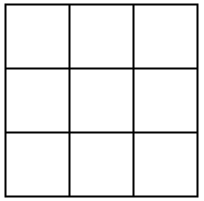
- e.g. $f_1(s)$ = (num white queens – num black queens), etc.

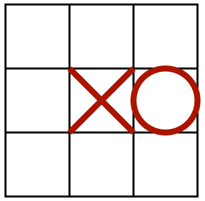
Evaluation Function

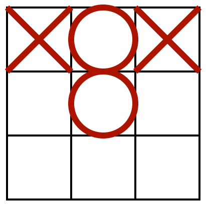
- **Function e**: state s → number **e(s)**
- e(s) is a **heuristic** that estimates how favorable s is for MAX
- e(s) > 0 means that s is favorable to MAX (the larger the better)
- e(s) < 0 means that s is favorable to MIN
- e(s) = 0 means that s is neutral

Example: Tic-tac-Toe

e(s) = number of rows, columns, and diagonals open for MAX - number of rows, columns, and diagonals open for MIN


8-8 = 0


6-4 = 2


3-3 = 0

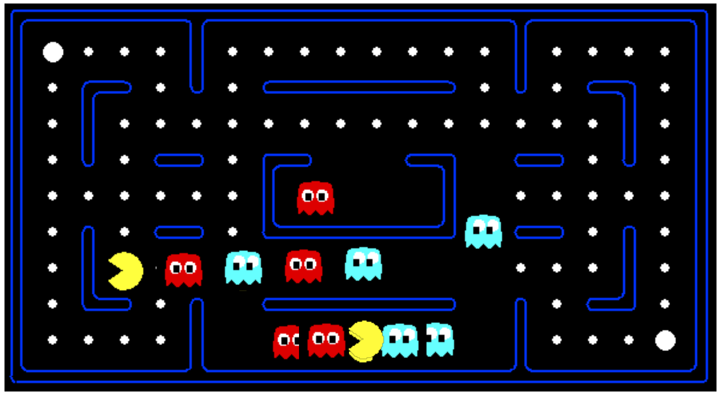
Construction of an Evaluation Function

- Usually a weighted sum of “features”:

$$e(s) = \sum_{i=1}^{n} w_i f_i(s)$$

- Features may include
  - Number of pieces of each type
  - Number of possible moves
  - Number of squares controlled

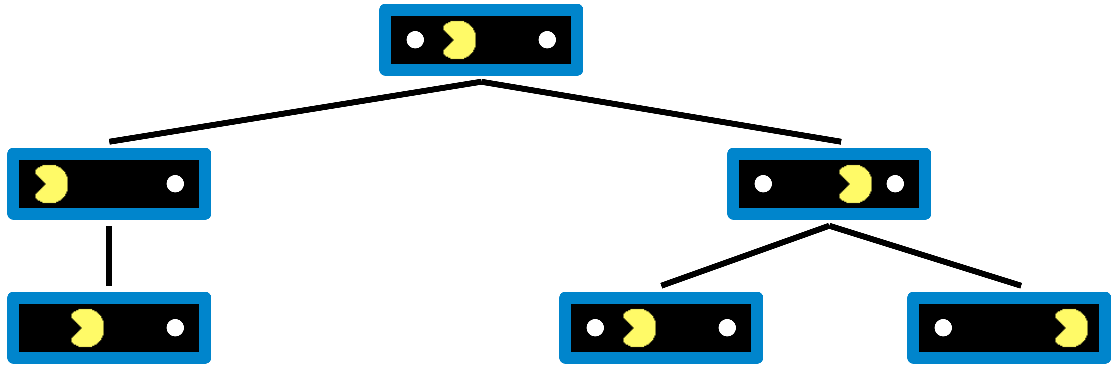
Evaluation for Pacman



[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function), smart ghosts coordinate (L6D6,7,8,10)]

Video of Demo Thrashing (d=2)

Why Pacman Starves



- A danger of replanning agents!
  - He knows his score will go up by eating the dot now (west, east)
  - He knows his score will go up just as much by eating the dot later (east, west)
  - There are no point-scoring opportunities after eating the dot (within the horizon, two here)
  - Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!

Video of Demo Thrashing -- Fixed (d=2)

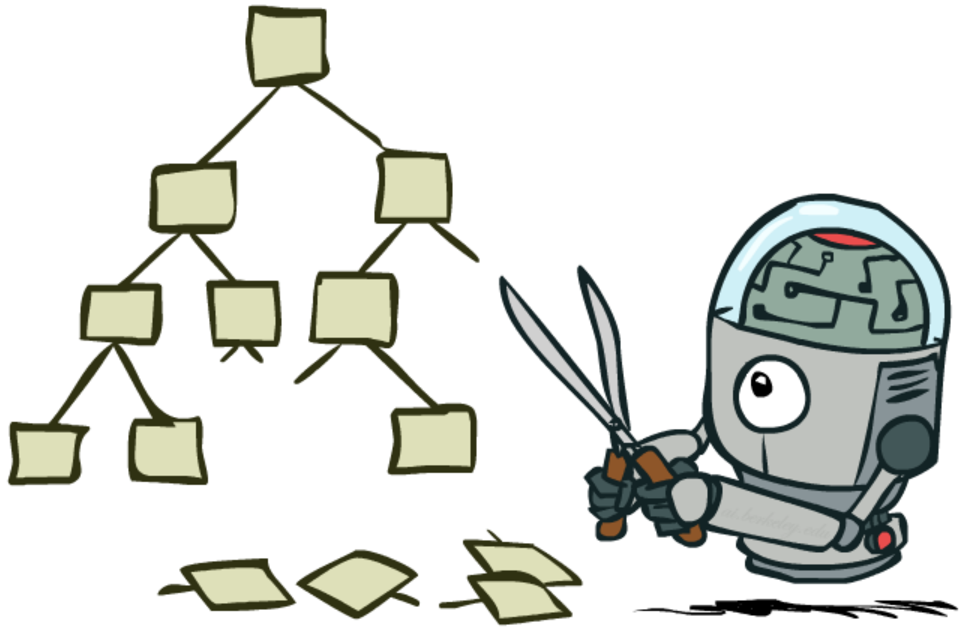
Video of Demo Smart Ghosts (Coordination)

Video of Demo Smart Ghosts (Coordination) – Zoomed In

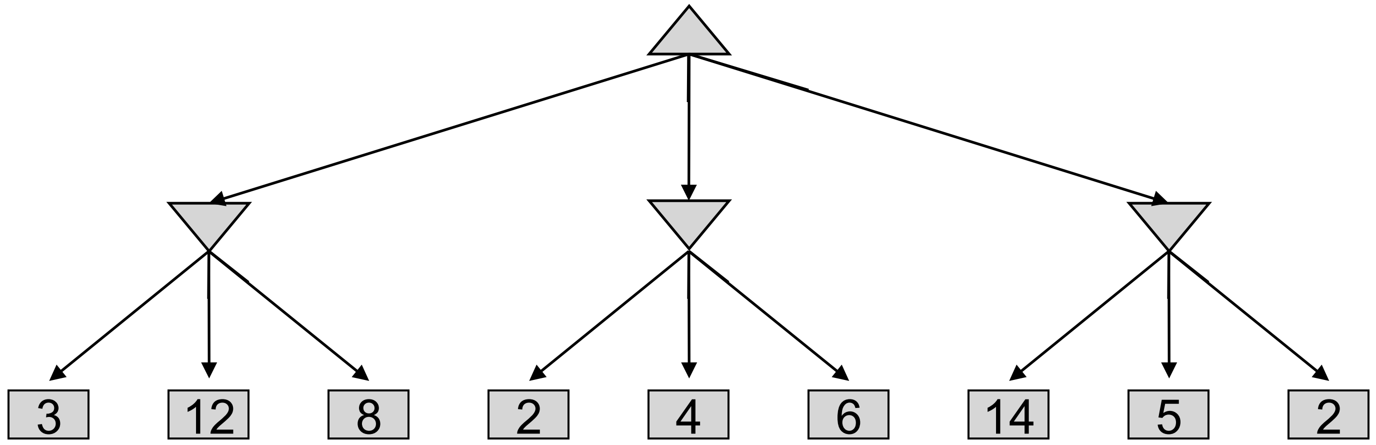
Project 2: Adversarial Pacman

- Due: Wednesday, Feb 15, 8pm
- http://www.mathcs.emory.edu/~eugene/cs325/p2/

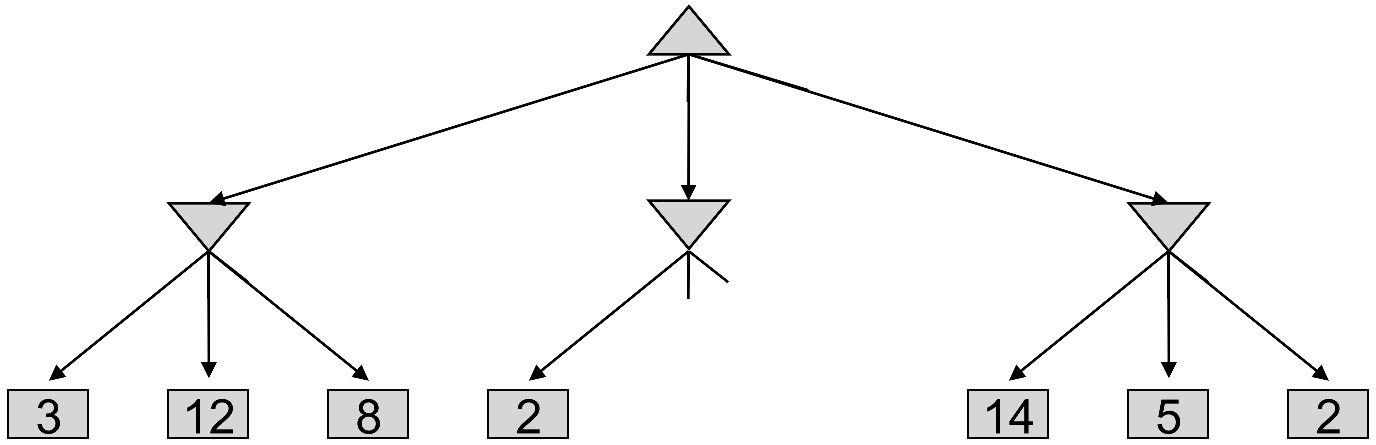
Game Tree Pruning



Minimax Example

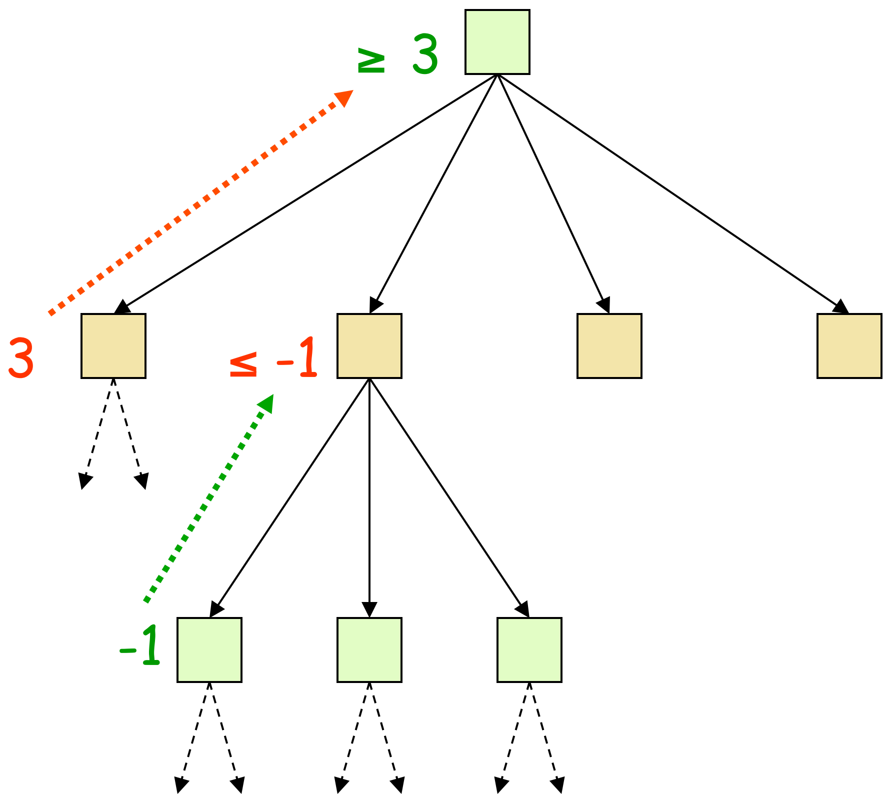


Minimax Pruning



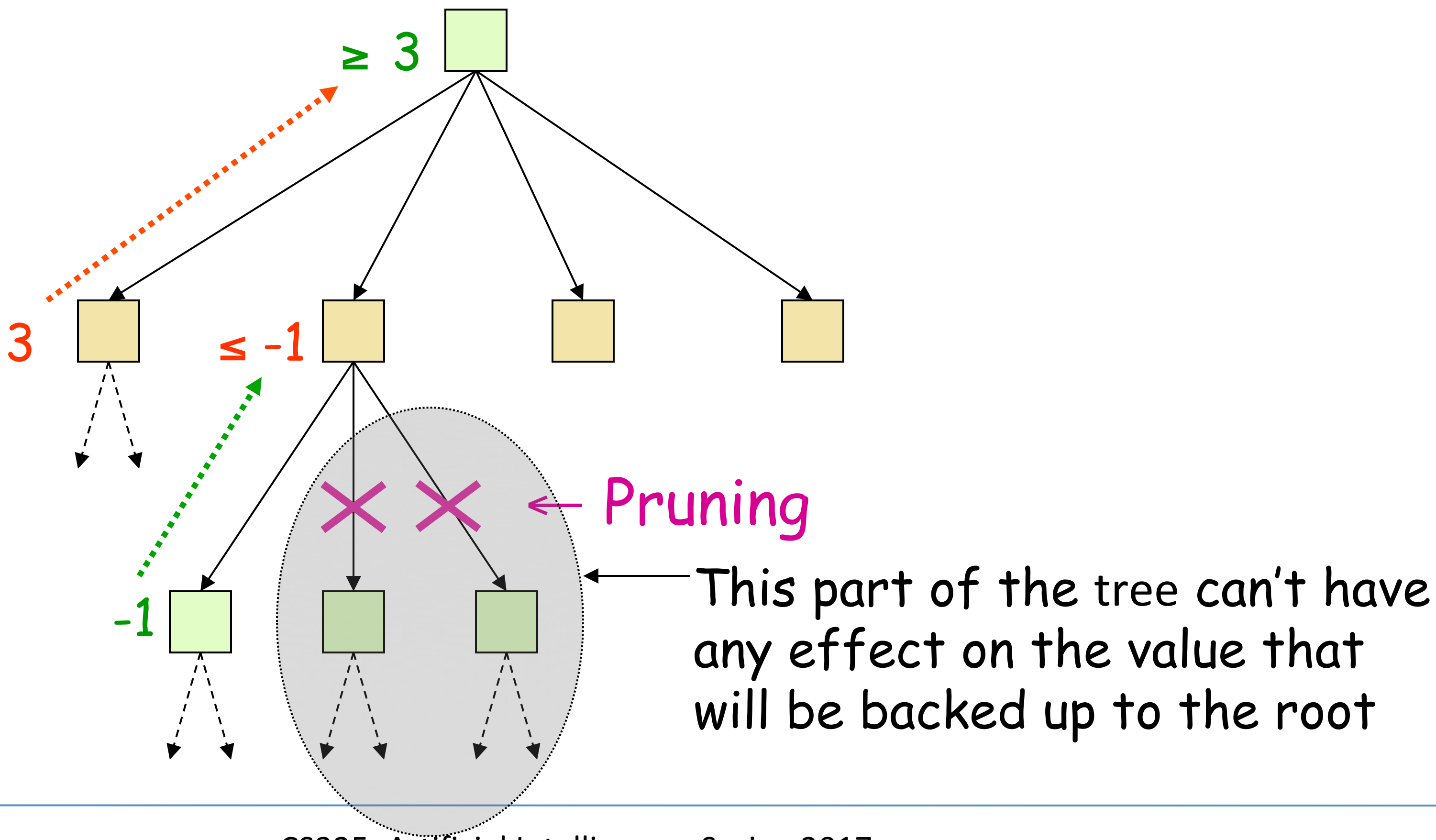
Can we do better?

Yes ! Much better !



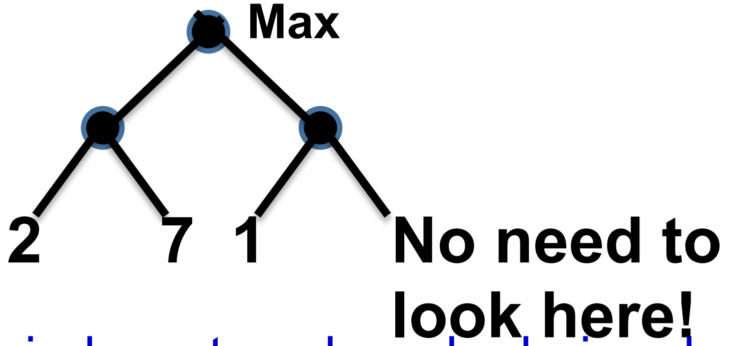
Can we do better?

Yes ! Much better !


≥ 3
3
≤ -1
-1
Pruning
This part of the tree can't have any effect on the value that will be backed up to the root

Alpha-Beta Pruning

- Typically can only look 3-4 ply in allowable chess time
- Alpha-beta pruning simplifies search space without eliminating optimality
  - By applying common sense
  - If one route allows queen to be captured and a better move is available
  - Then don't search further down bad path
  - If one route would be bad for opponent, ignore that route also



Maintain [alpha, beta] window at each node during depth-first search

alpha = lower bound, change at max levels
beta = upper bound, change at min levels

Alpha-Beta Pruning

- Explore the game tree to depth h in depth-first manner
- Back up alpha and beta values whenever possible
- Prune branches that can’t lead to changing the final decision

Alpha-beta Algorithm

- Depth first search – only considers nodes along a single path at any time

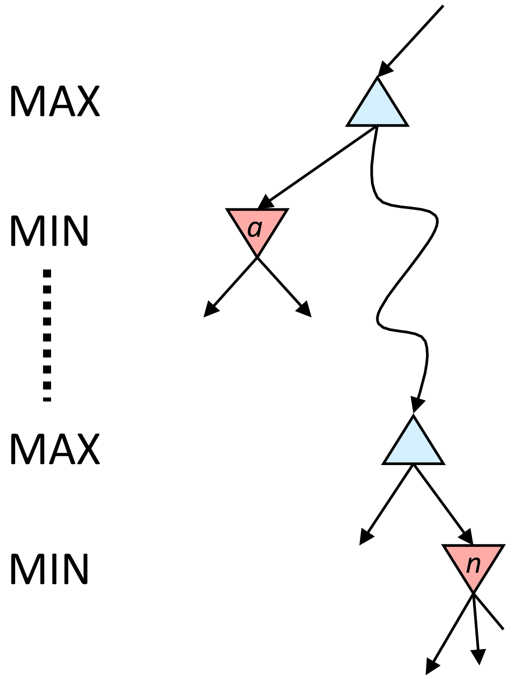
→ α = highest-value choice we have found at any choice point along the path for MAX

→ β = lowest-value choice we have found at any choice point along the path for MIN

- update values of α and β during search, and prune remaining branches as soon as the value is known to be worse than the current α or β value for MAX or MIN

Alpha-Beta Pruning

- General configuration (MIN version)
  - We're computing the MIN-VALUE at some node *n*
  - We're looping over *n*'s children
  - *n*'s estimate of the childrens' min is dropping
  - Who cares about *n*'s value? MAX
  - Let *a* be the best value that MAX can get at any choice point along the current path from the root
  - If *n* becomes worse than *a*, MAX will avoid it, so we can stop considering *n*'s other children (it's already bad enough that it won't be played)



- MAX version is symmetric

Alpha-Beta Implementation

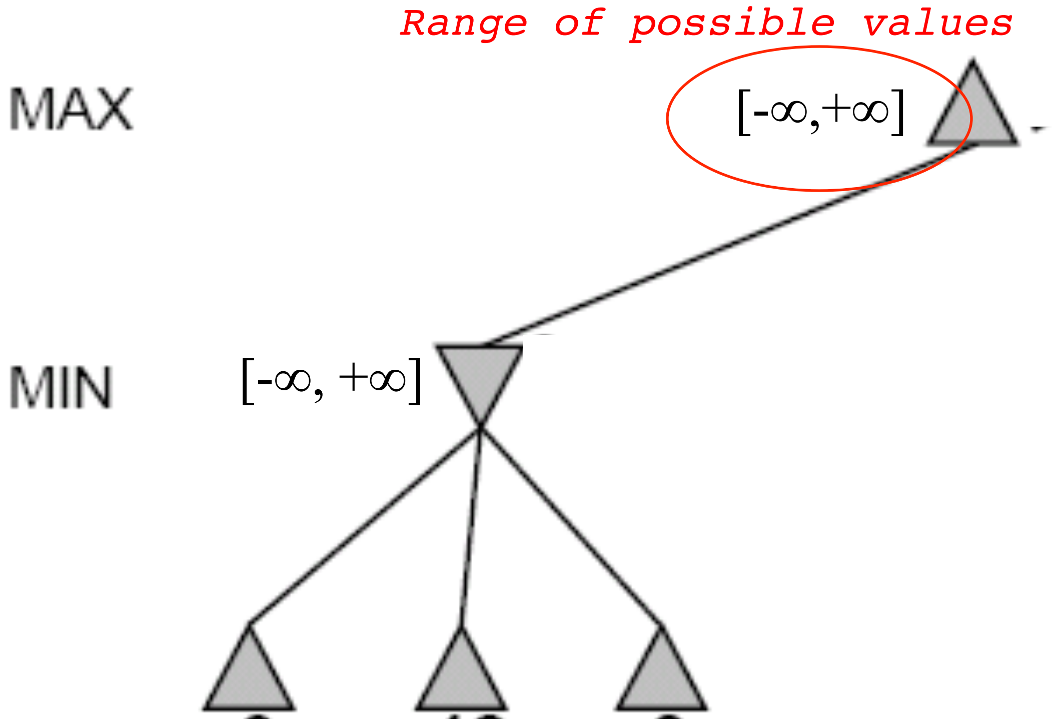
α: MAX's best option on path to root
β: MIN's best option on path to root

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

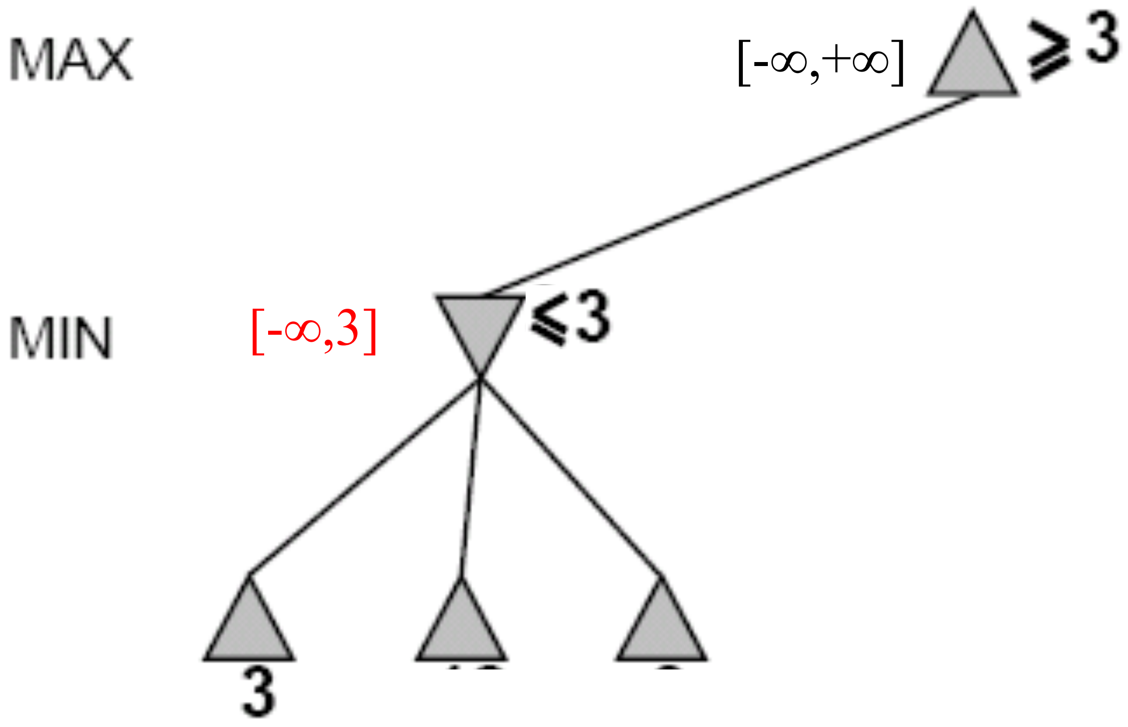
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Alpha-Beta Example

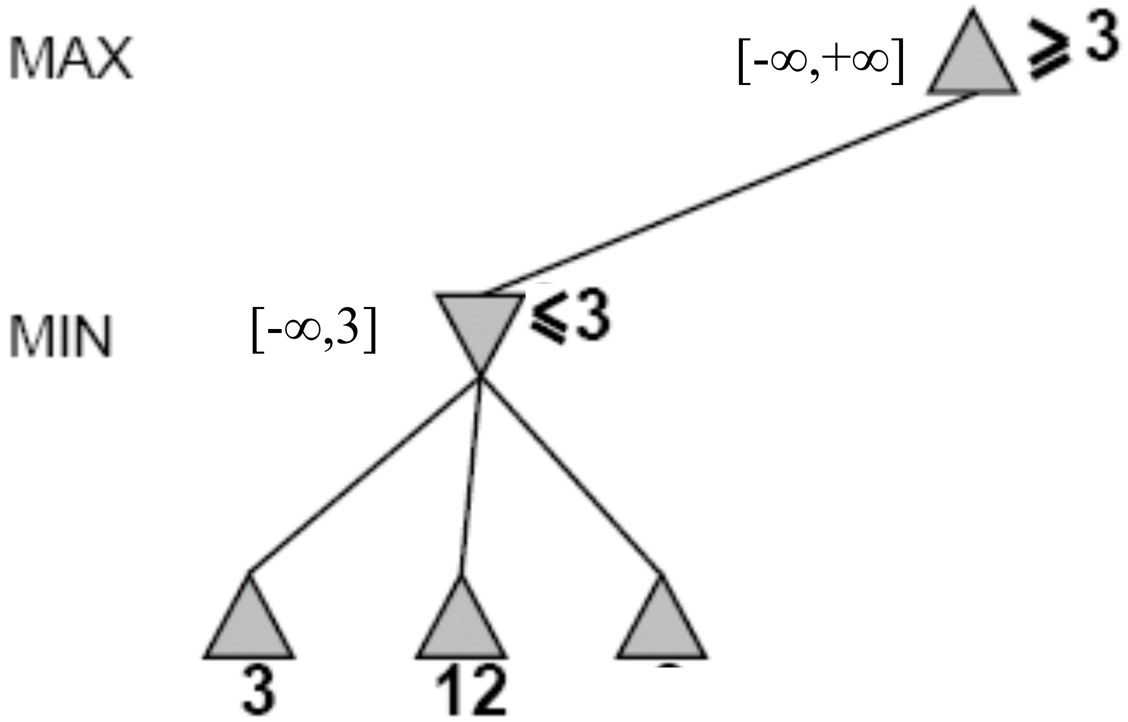
Do DF-search until first leaf



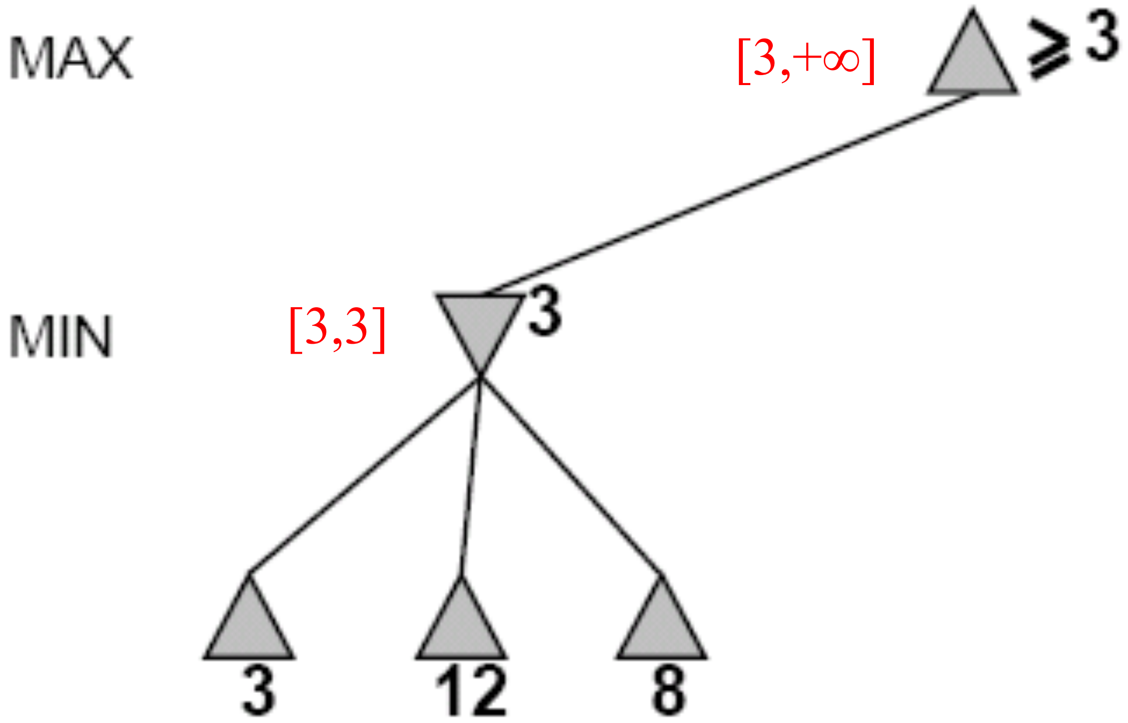
Alpha-Beta Example (continued)



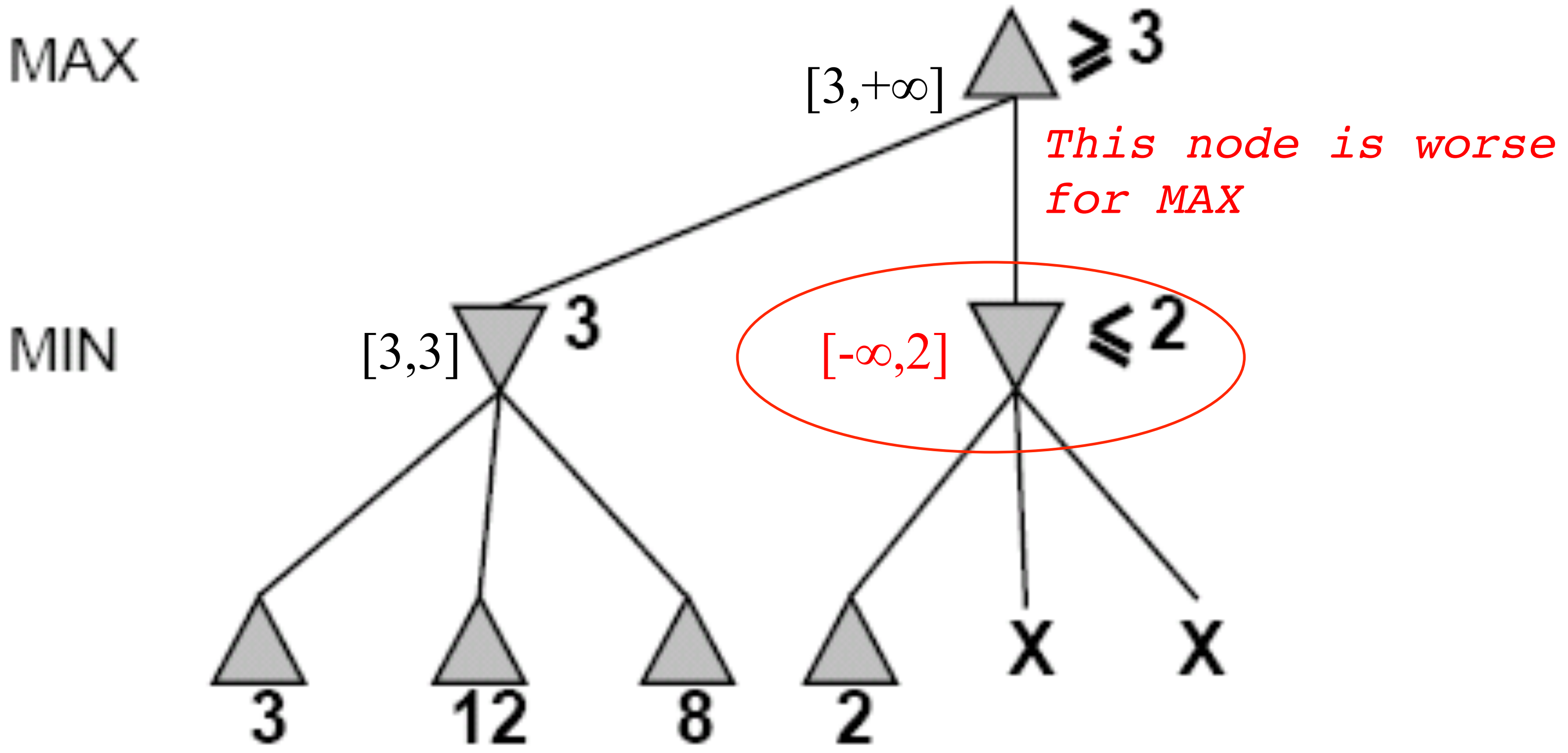
Alpha-Beta Example (continued)



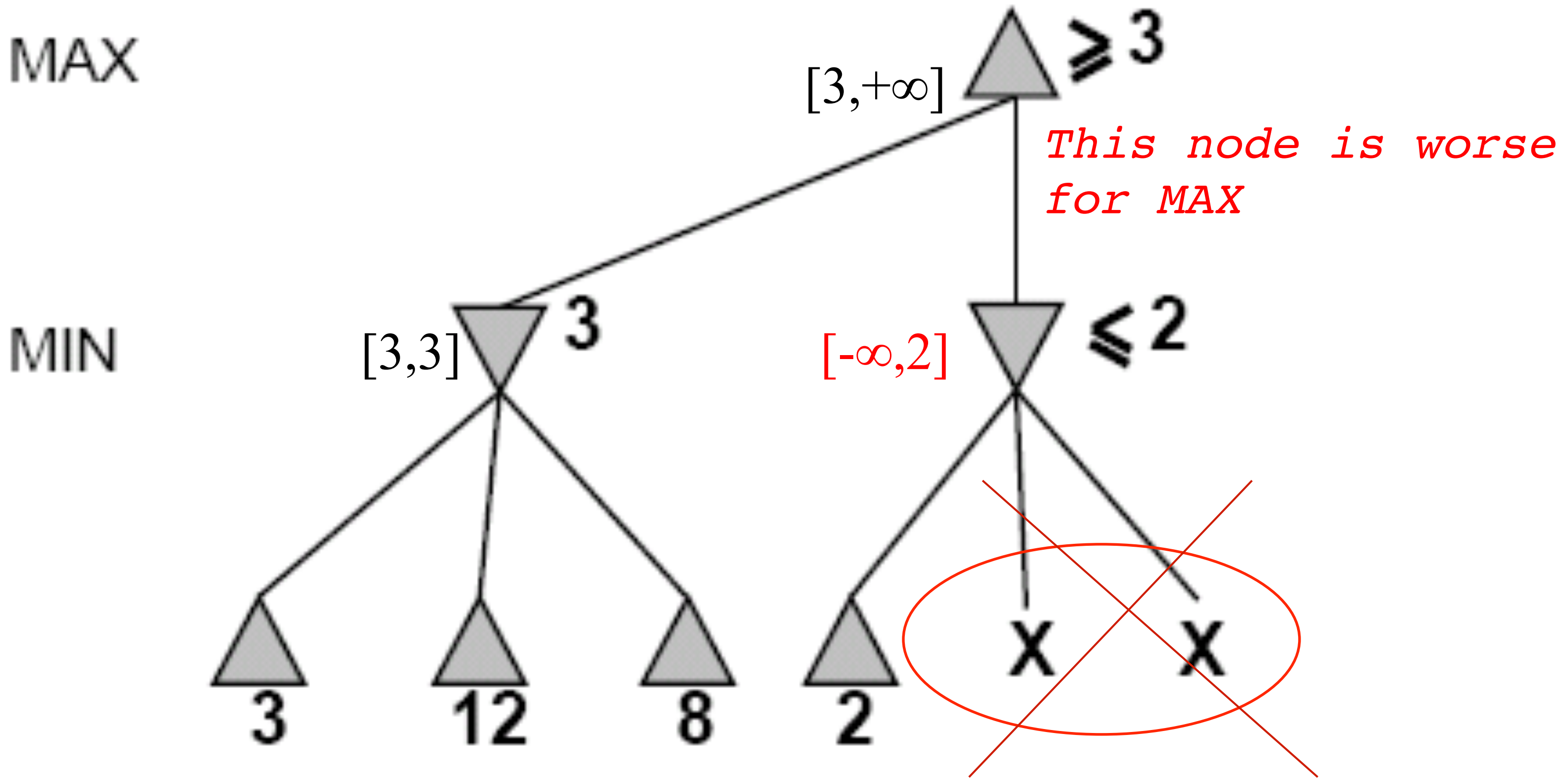
Alpha-Beta Example (continued)



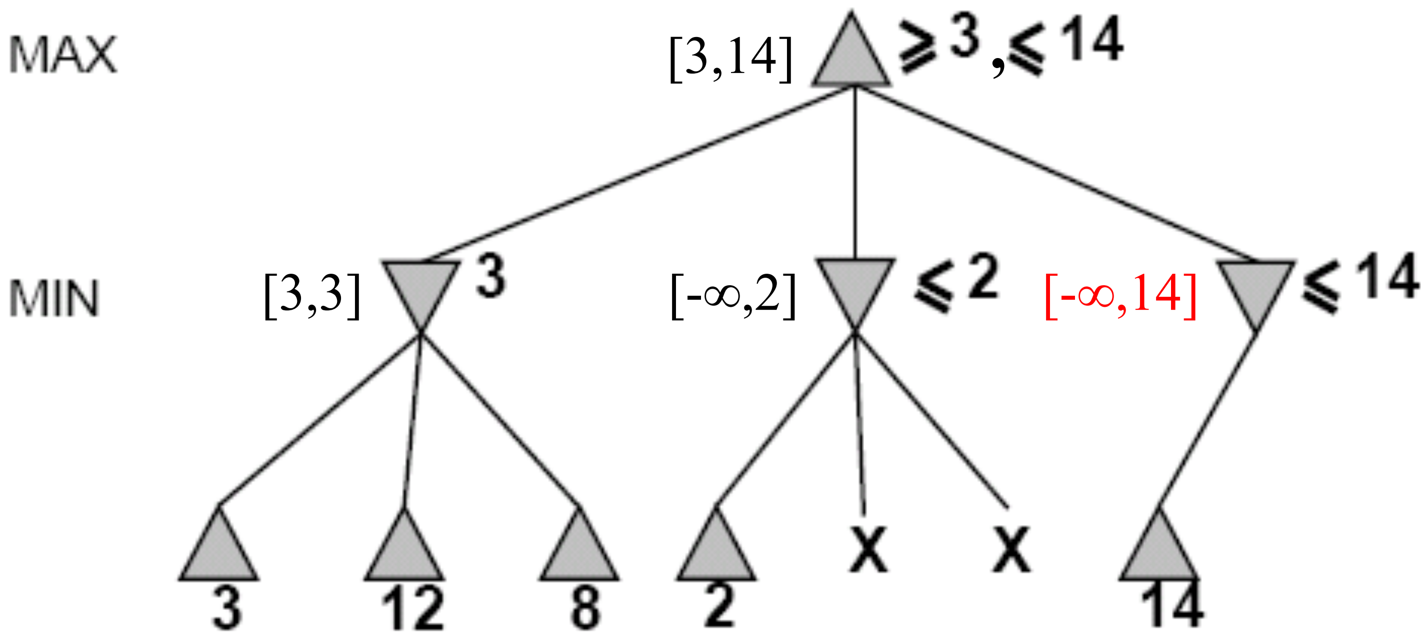
Alpha-Beta Example (continued)



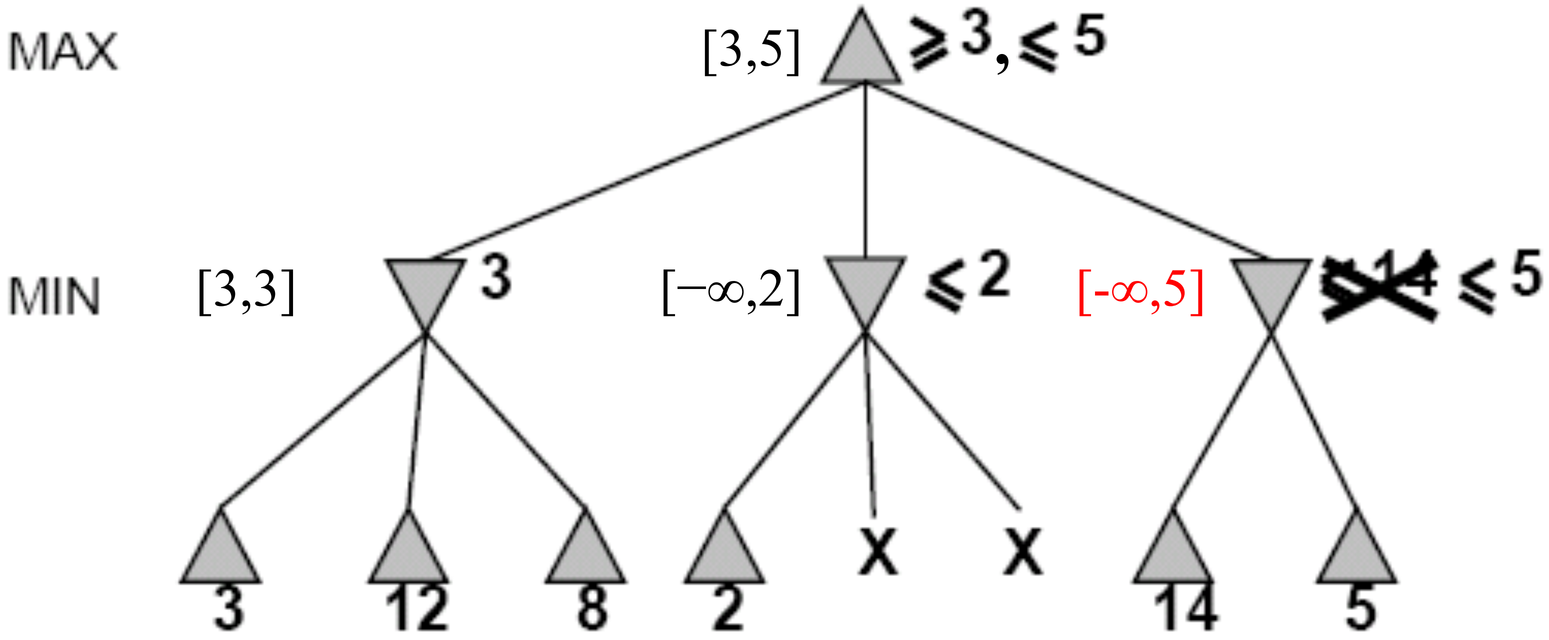
Alpha-Beta Example (continued)



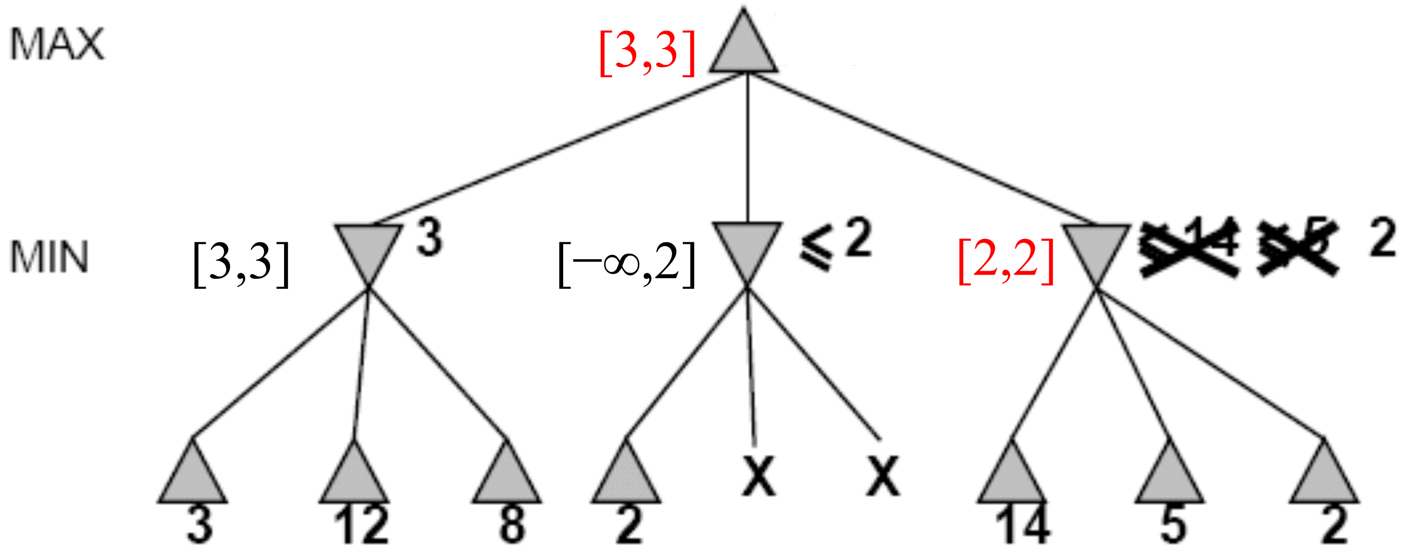
Alpha-Beta Example (continued)



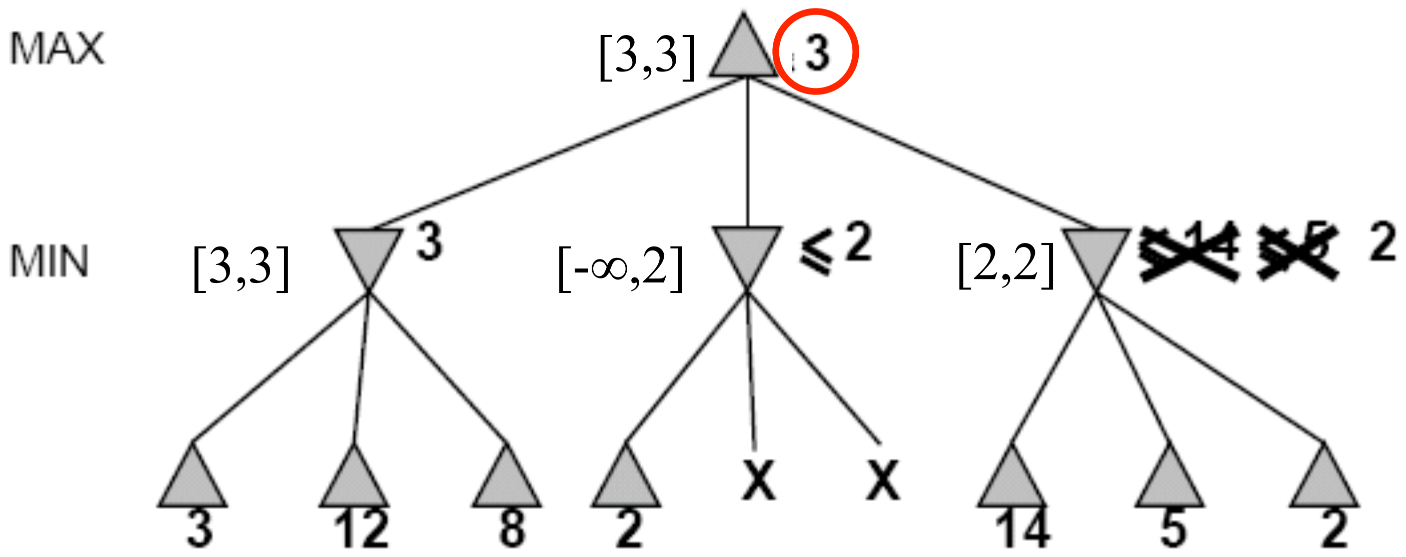
Alpha-Beta Example (continued)



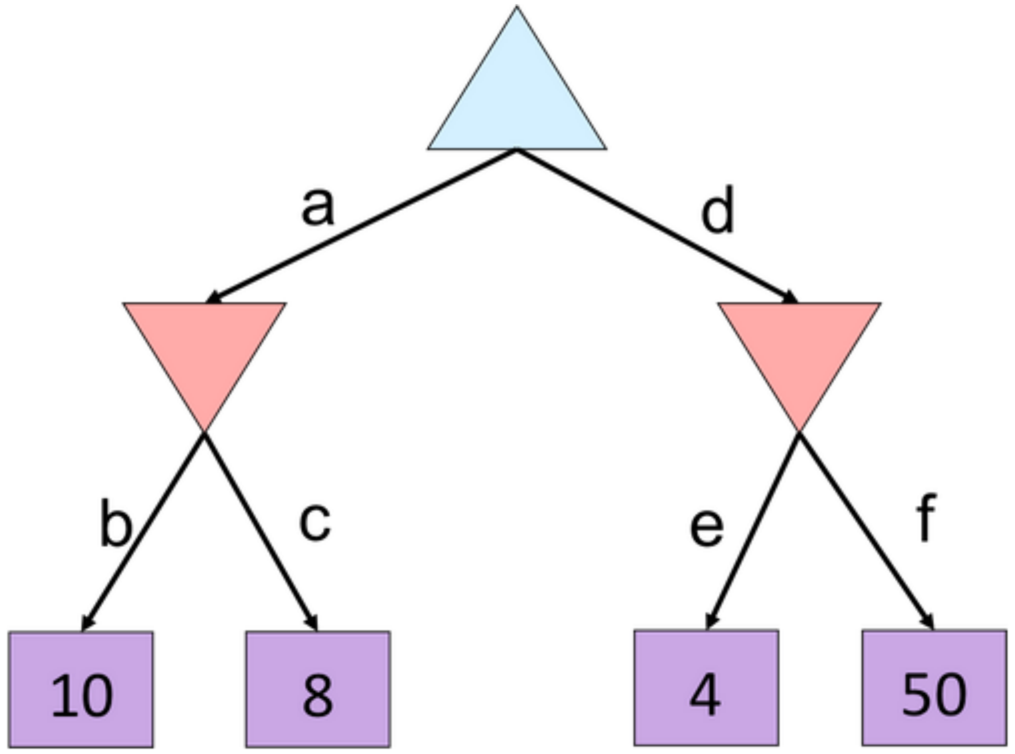
Alpha-Beta Example (continued)



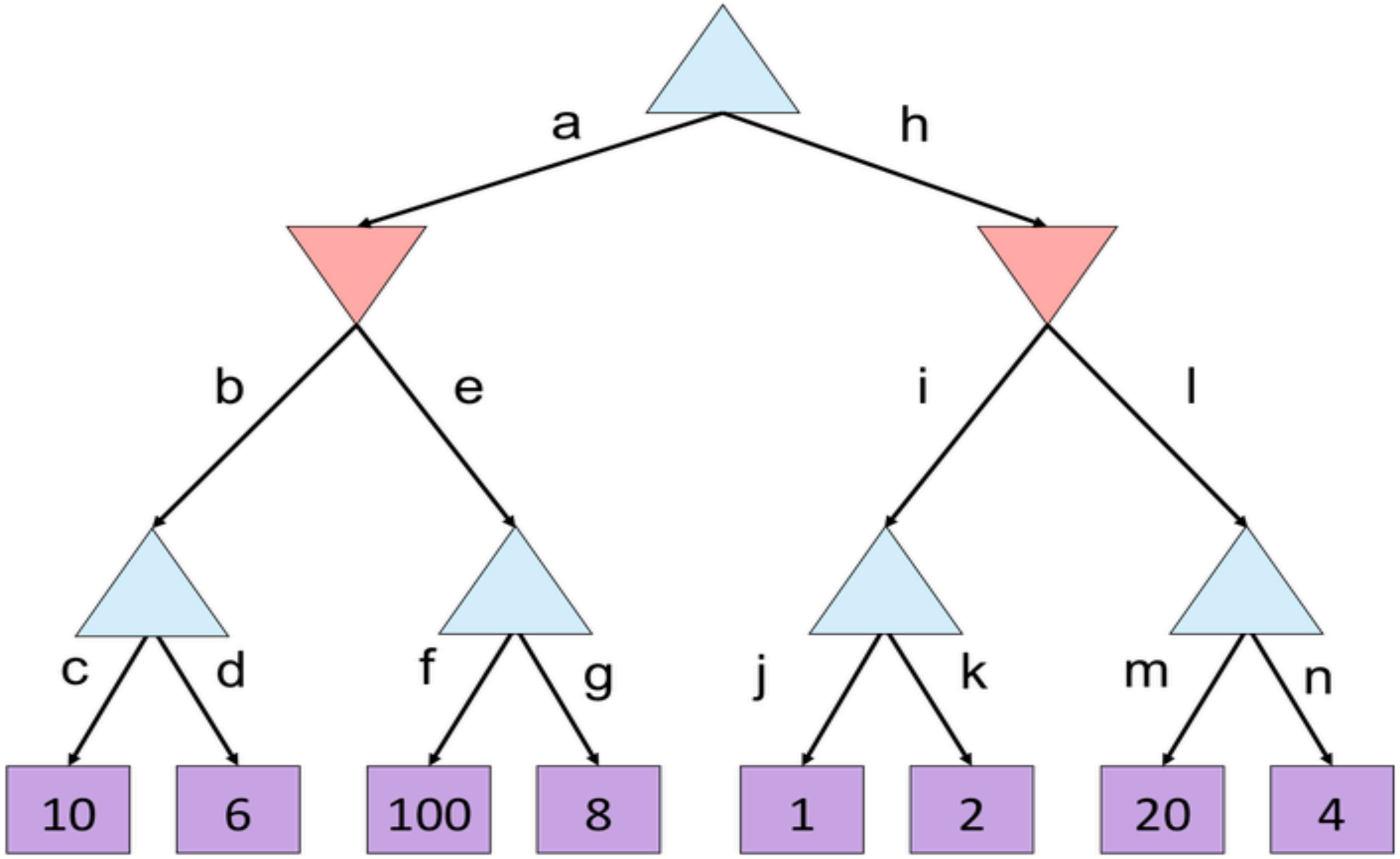
Alpha-Beta Example (continued)



Alpha-Beta Quiz



Alpha-Beta Quiz 2



More Alpha-Beta Pruning Examples

- Generic game tree visualization: http://homepage.ufp.pt/jtorres/ensino/ia/alfabeta.html
- Connect Four: https://gimu.org/connect-four-js/

Bad and Good Cases for Alpha-Beta Pruning

- Bad: Worst moves encountered first

```
                        4                         MAX
            +----------------+----------------+
            2                 3                 4          MIN
      +----+----+      +----+----+      +----+----+
      6     4     2     7     5     3     8     6     4        MAX
      +--+  +--+  +--+ +-+-+  +--+ +--+  +--+  +--+ +--+--+
      6   5  4   3  2  1 1 3 7   4  5  2  3  8   2  1  6 1   2   4
```

- Good: Good moves ordered first

```
                        4                         MAX
            +----------------+----------------+
            4                 3                 2          MIN
       +----+----+      +----+----+      +----+----+
       4     6     8     3     x     x     2     x     x        MAX
     +--+  +--+  +--+ +--+              +-+-+
     4  2  6   x  8   x  3  2            1  2 1
```

- If we can order moves, we can get more benefit from alpha-beta pruning

Alpha Beta Properties

- Pruning does not affect final result
- Good move ordering improves effectiveness of pruning
- With **perfect ordering**, time complexity is $O(b^{m/2})$

Alpha-Beta Pruning Properties

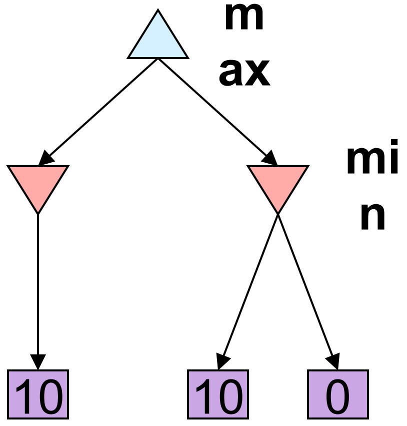
- This pruning has no effect on minimax value computed for the root!

- 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)



Analysis of Alpha-Beta Search

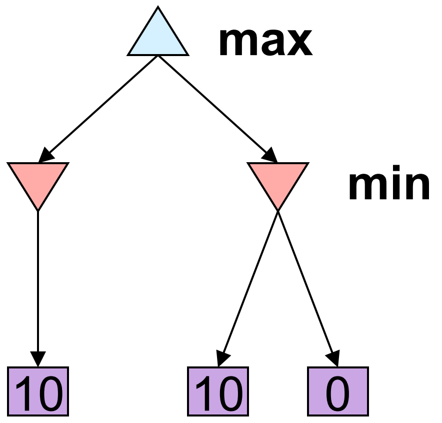
- Worst-Case
  - branches are ordered so that no pruning takes place. In this case alpha-beta gives no improvement over exhaustive search

- Best-Case
  - each player's best move is the left-most alternative (i.e., evaluated first)
  - in practice, performance is closer to best rather than worst-case

- In practice often get $O(b^{(d/2)})$ rather than $O(b^d)$
  - this is the same as having a branching factor of sqrt(b),
    - since $(\text{sqrt}(b))^d = b^{(d/2)}$
    - i.e., we have effectively gone from b to square root of b
  - e.g., in chess go from b ~ 35 to b ~ 6
    - this permits much deeper search in the same amount of time

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)



To be continued on Tuesday…

- Project 2 out (Due Feb 15): look over code now:
  http://www.mathcs.emory.edu/~eugene/cs325/p2/

- Questions? Ask on Piazza or in class on Tuesday

- Instructor away Monday-Wed (no office hours Wed)

- TA office hours as normal