CSC242: Intro to AI

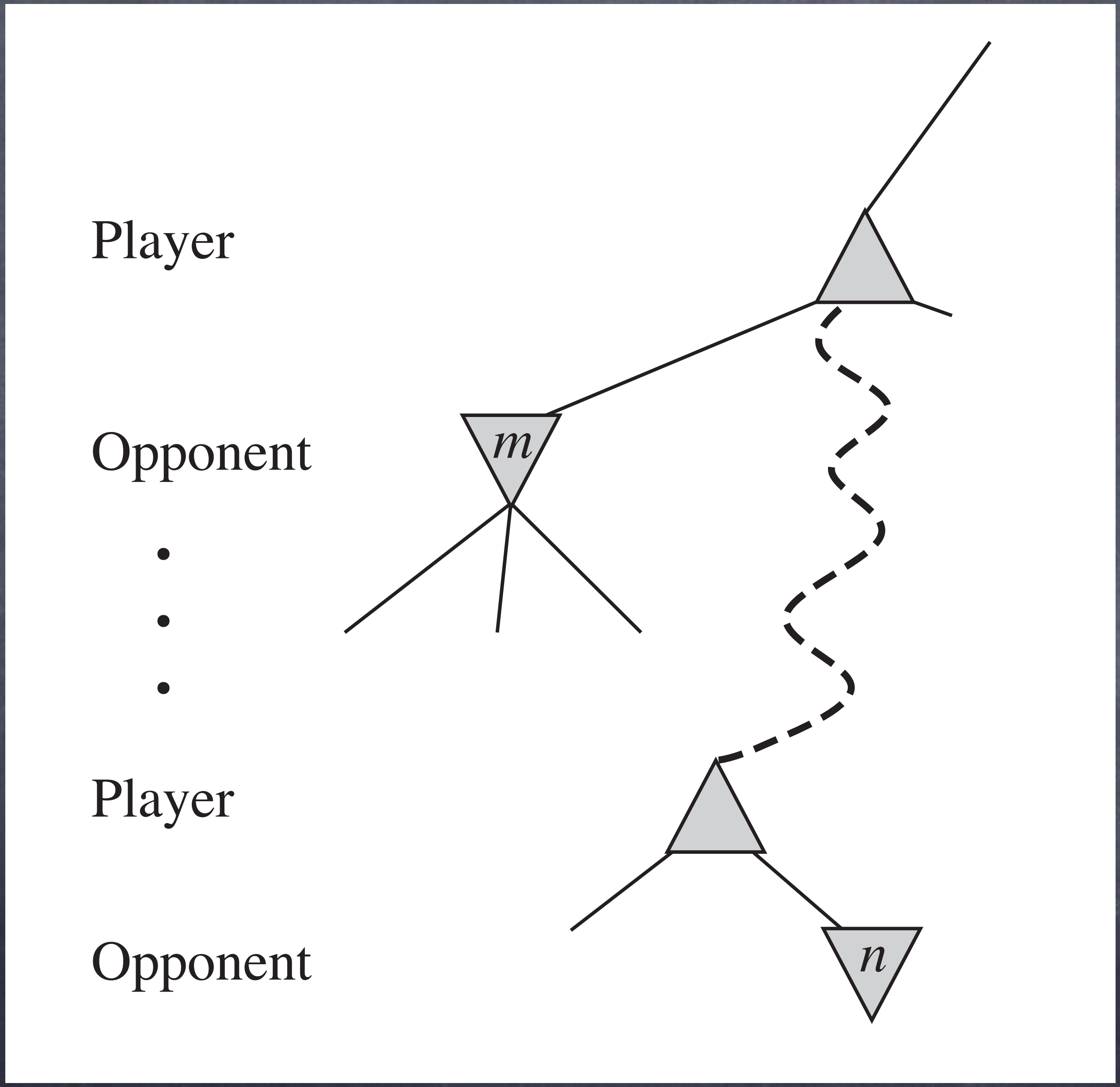
Adversarial Search

Part II

Today

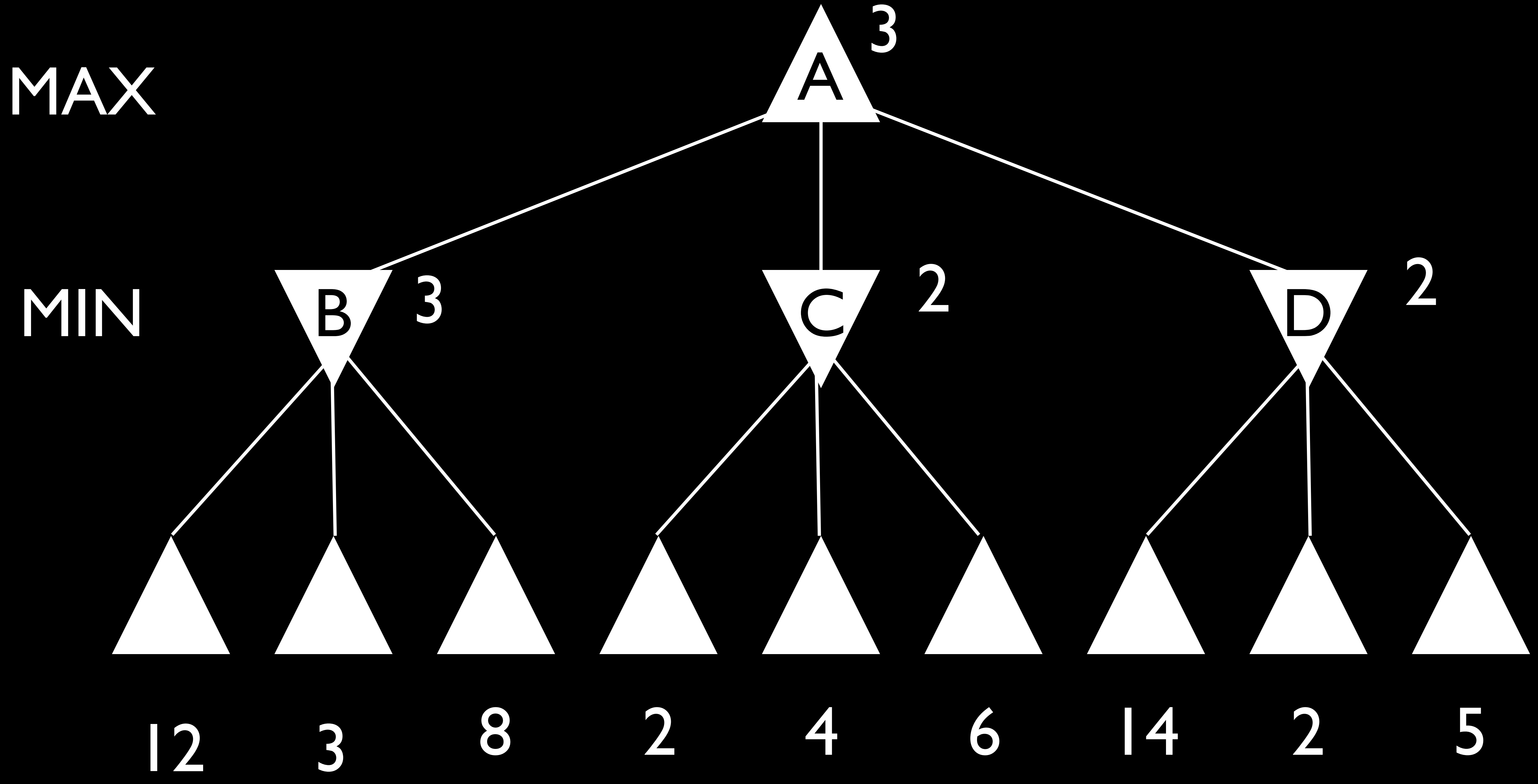
- How to improve MINIMAX to make it more practical
- Details of Othello project
- How to generalize MINIMAX for uncertainty and hidden information

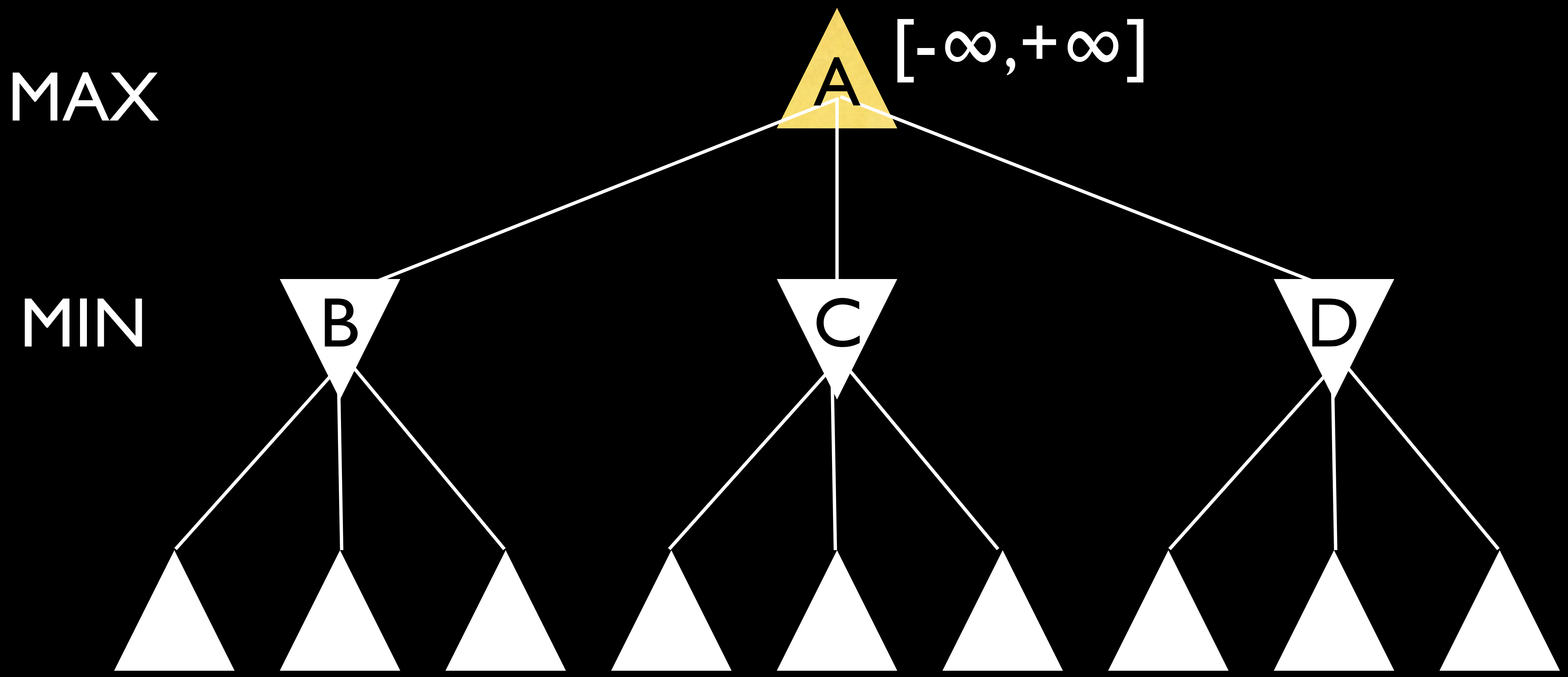
Alpha-Beta Pruning

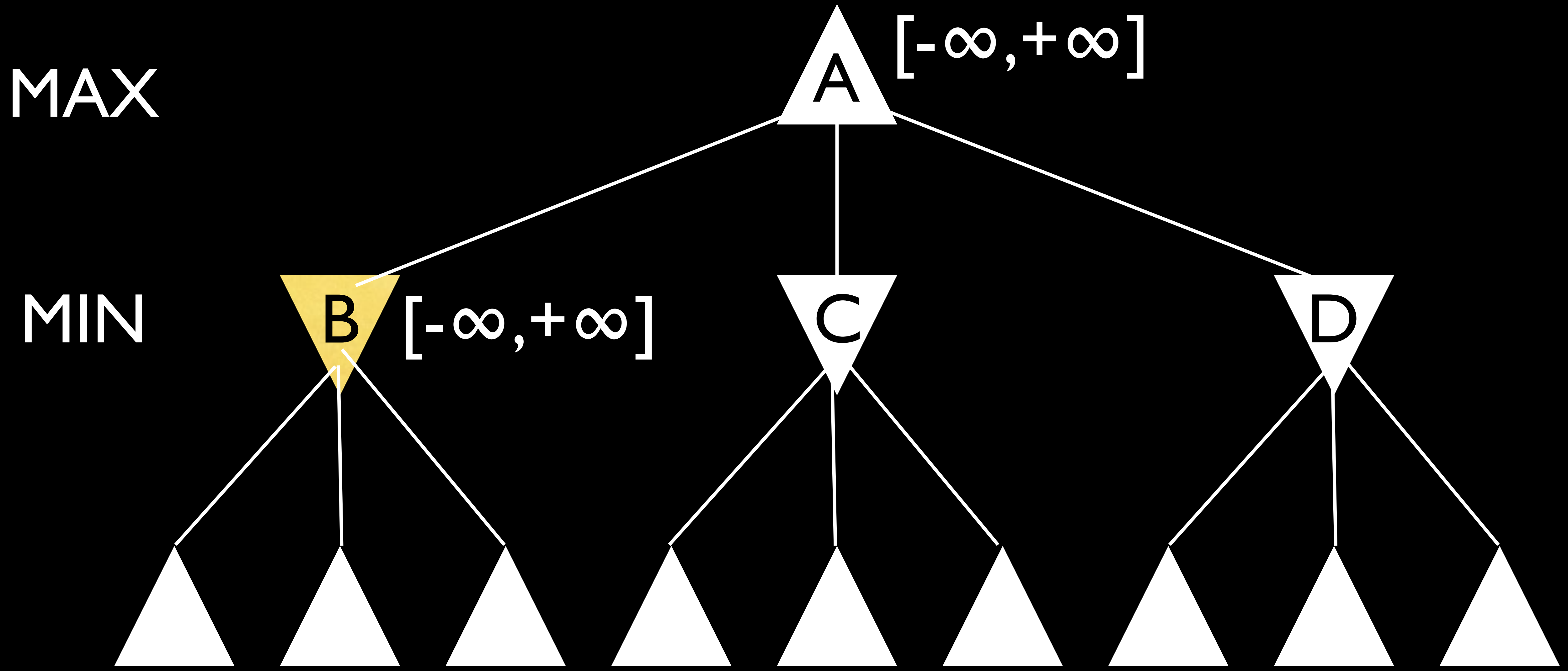


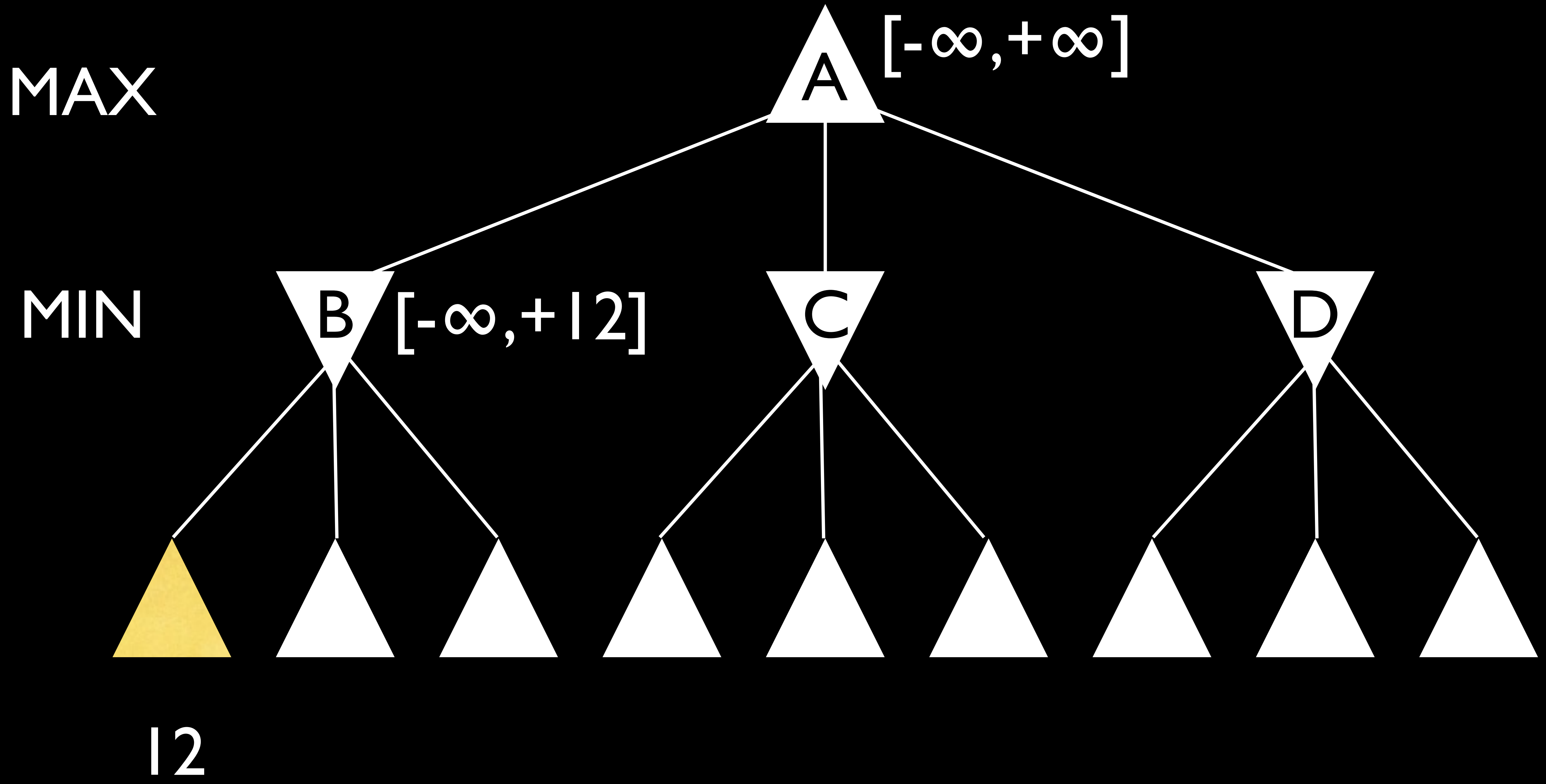
Alpha-Beta Pruning

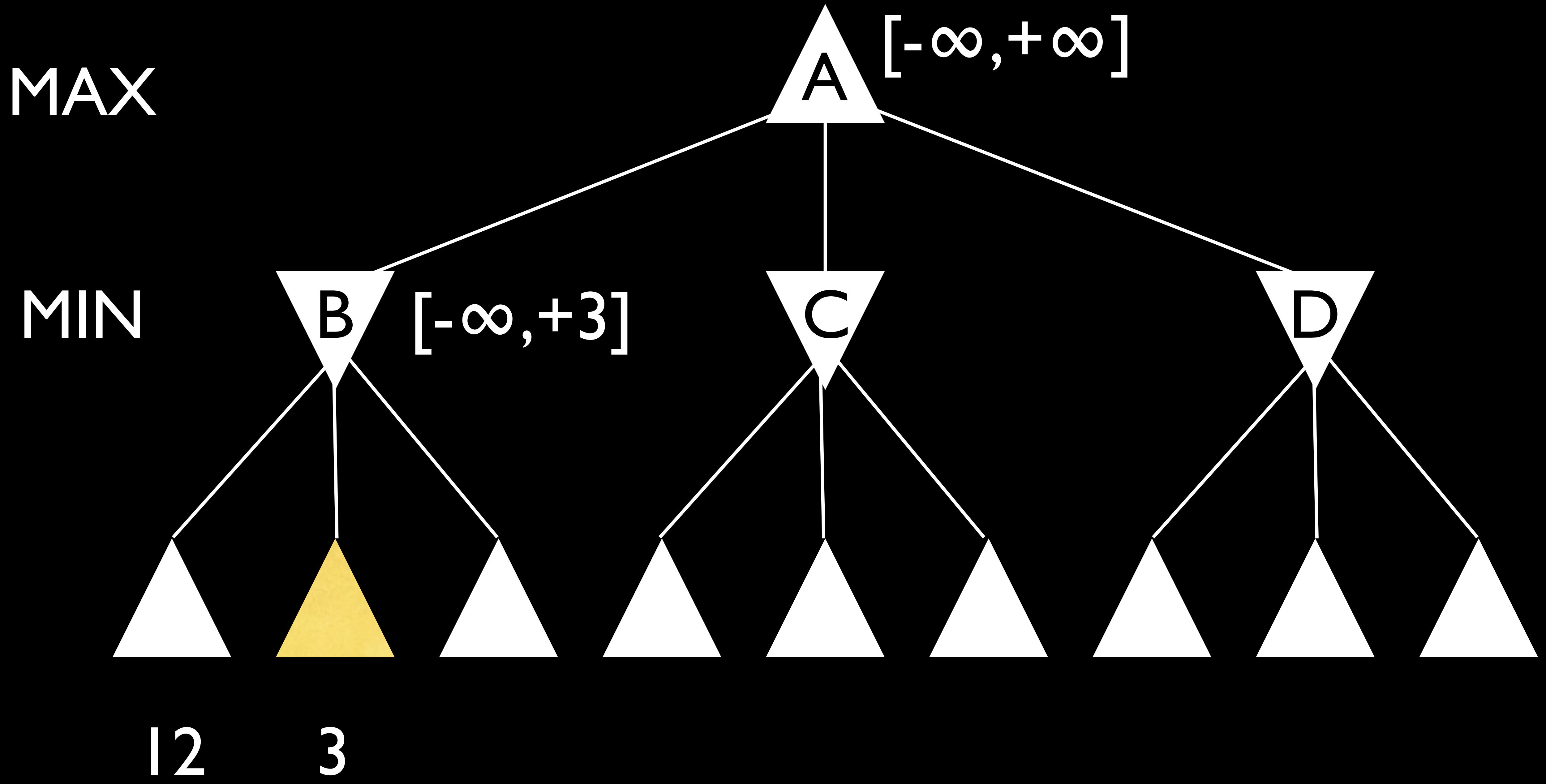
- How can we make MiniMax run faster, without sacrificing optimality?
- During MINIMAX search keep track of
  - α: value of best choice so far for MAX (lower bound on MAX utility)
  - β: value of best choice so far for MIN (upper bound on MIN utility)
- Prune when value of node is known to be worse than α (for MAX) or β (for MIN)

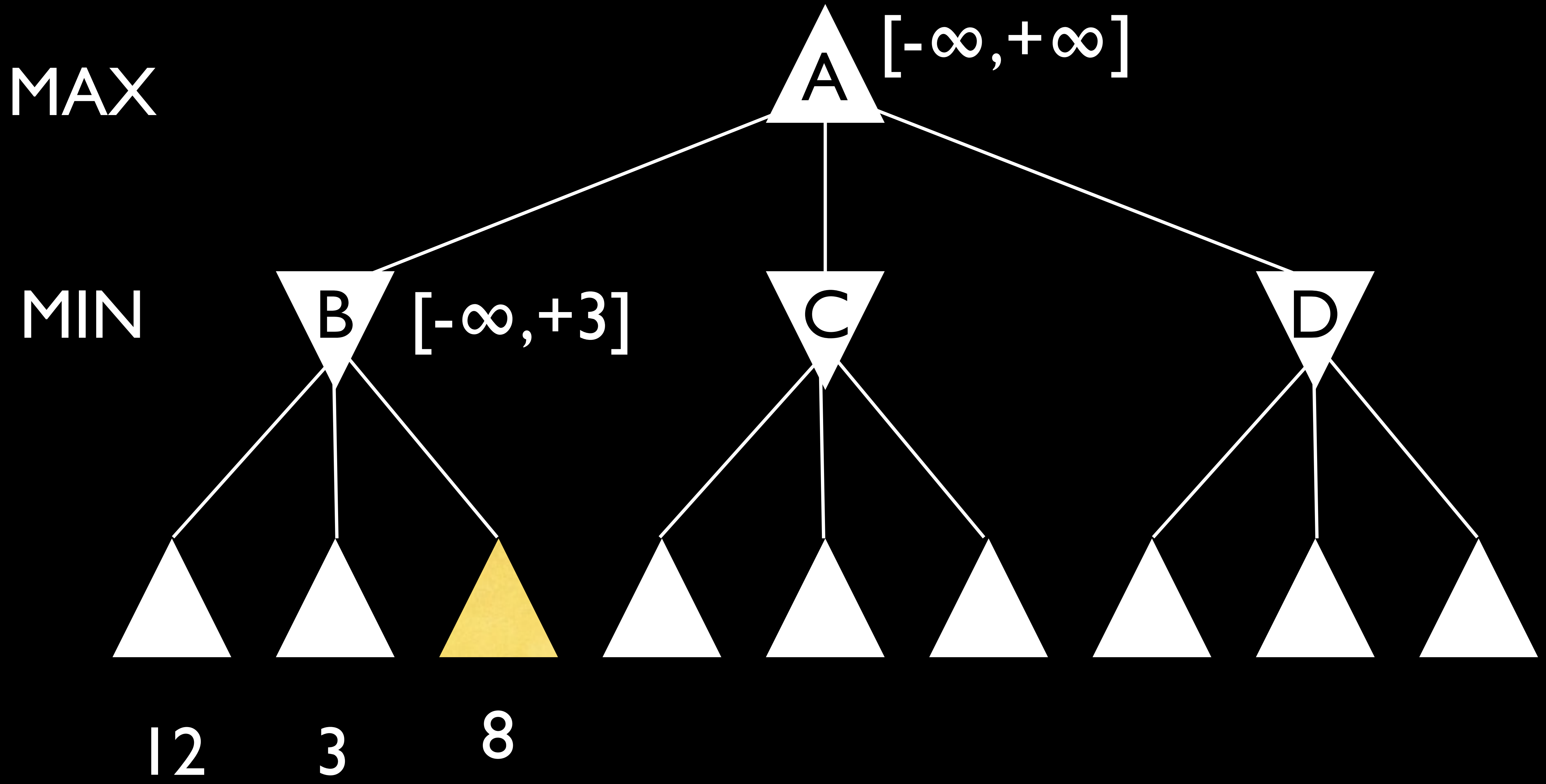

MAX
A
3
MIN
B
3
C
2
D
2
12
3
8
2
4
6
14
2
5

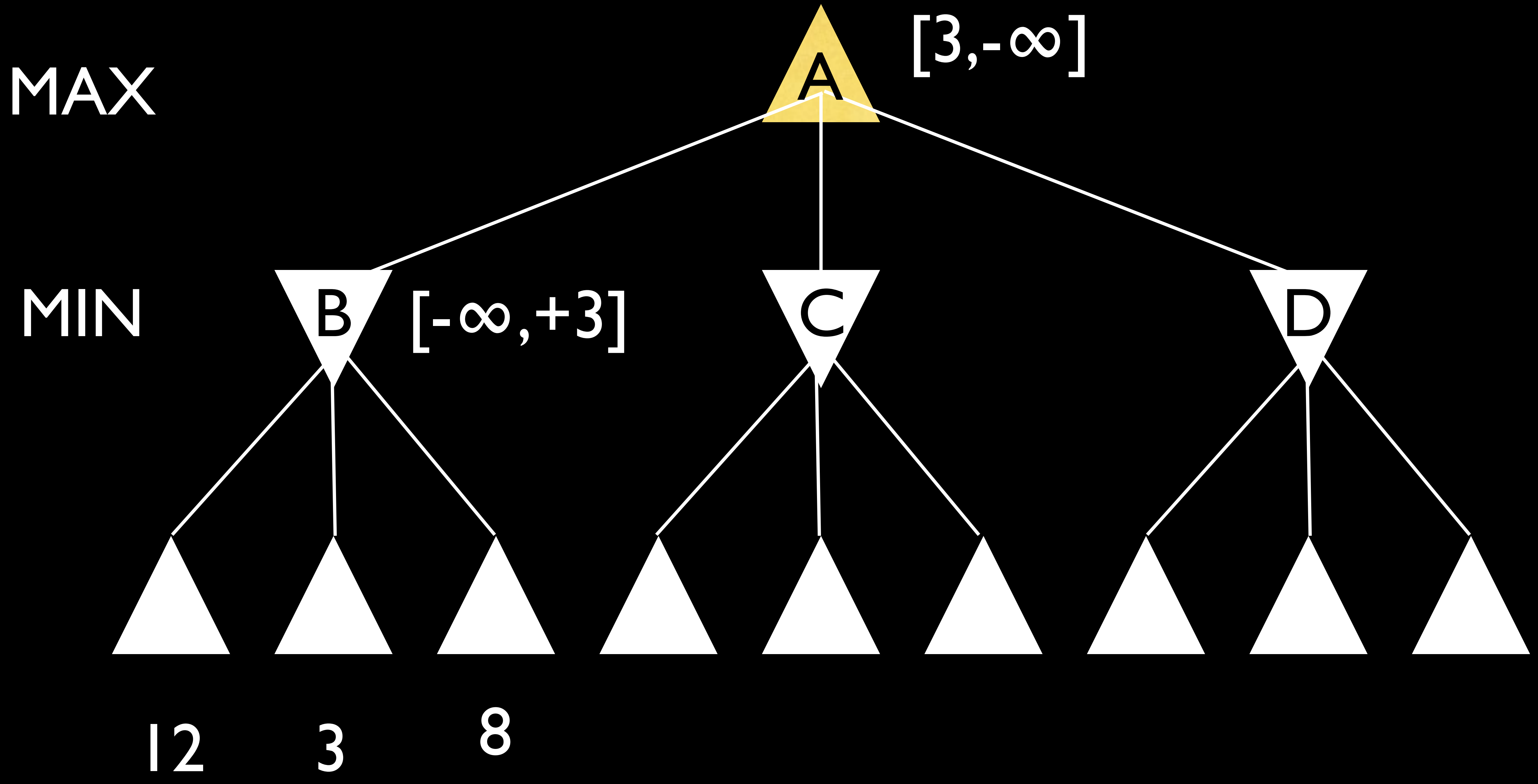

MAX
MIN
A
[-∞,+∞]
B
C
D

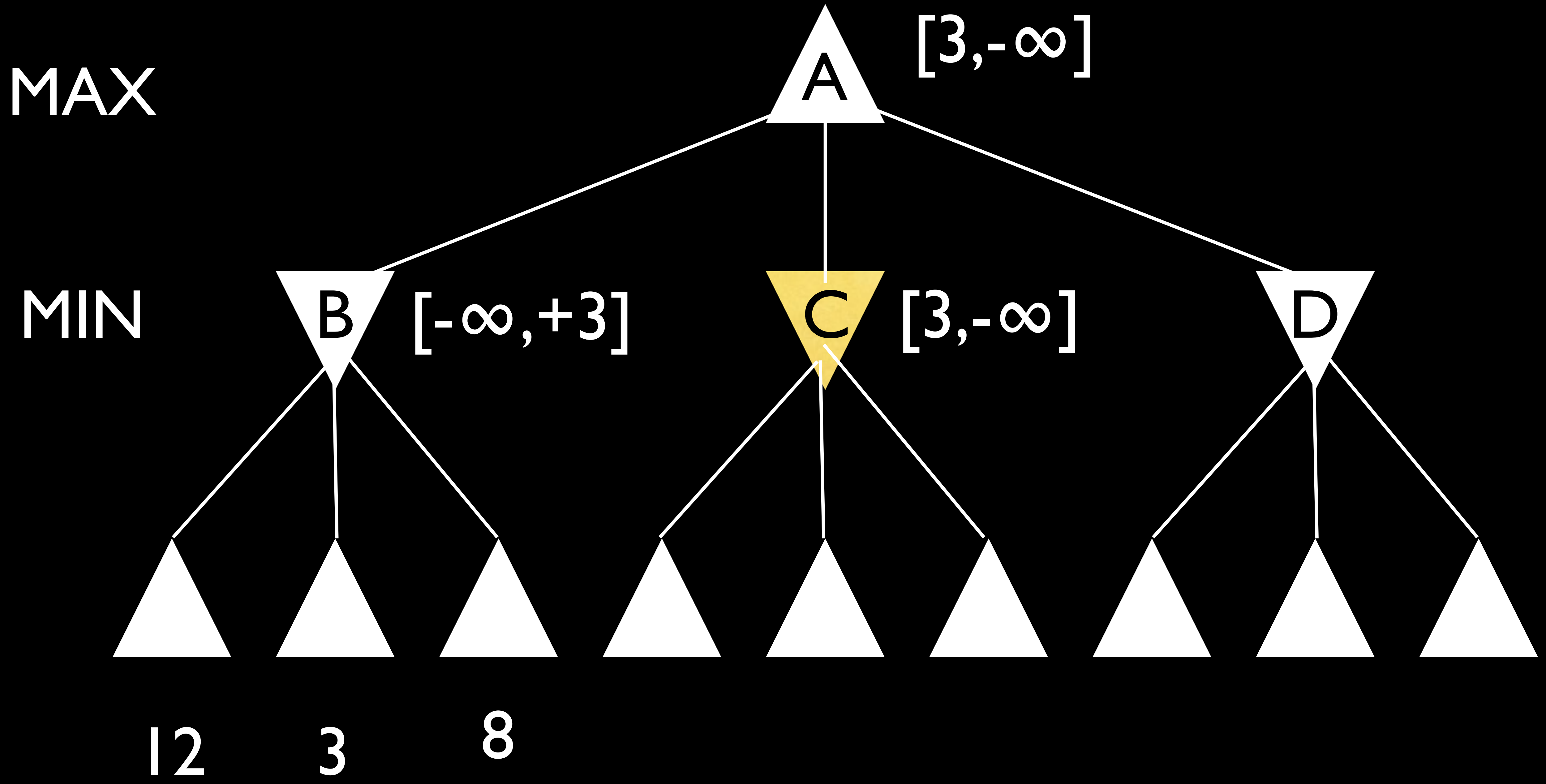

MAX
MIN
A
[-∞,+∞]
B
[-∞,+∞]
C
D

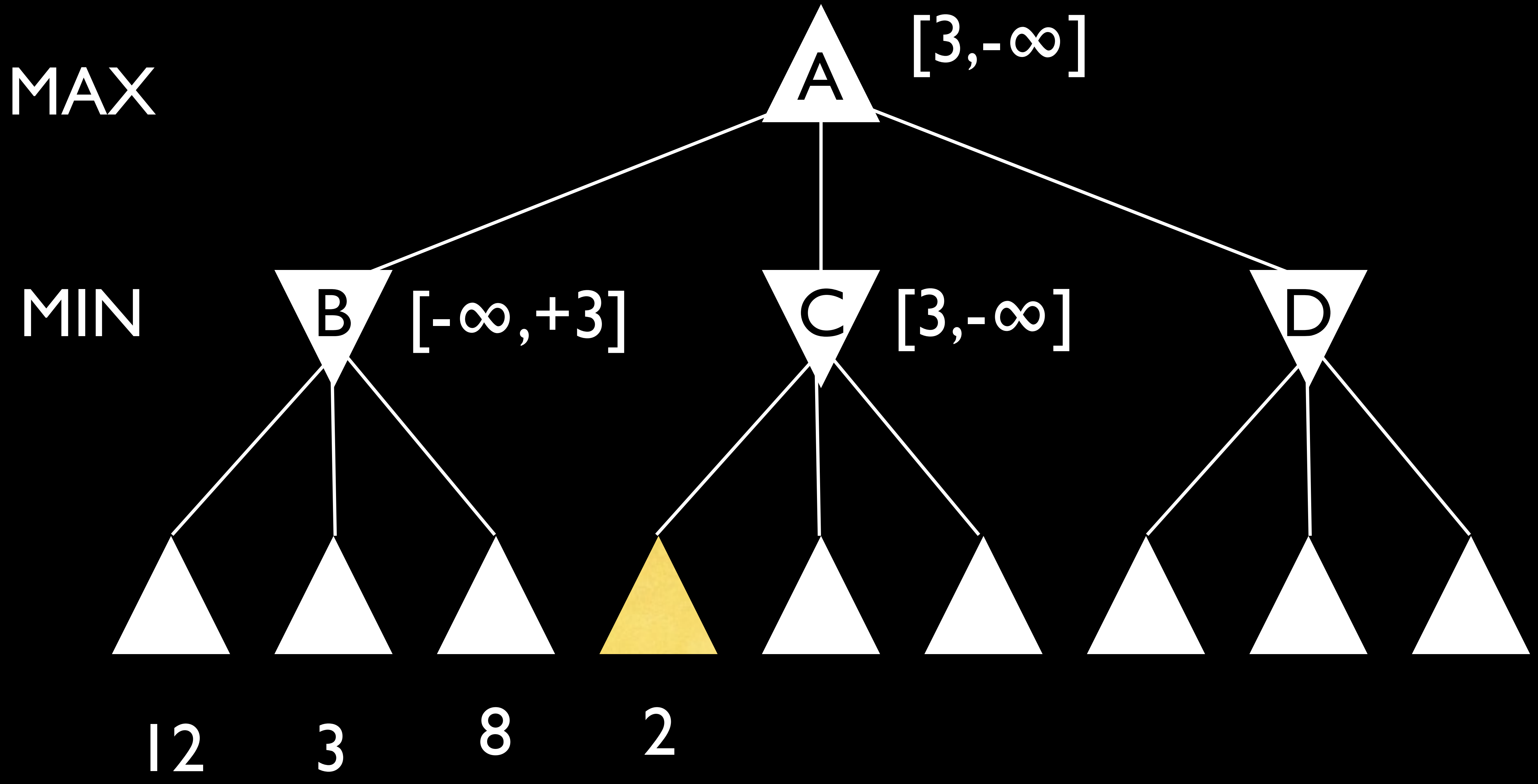

MAX
MIN
A
[-∞,+∞]
B
[-∞,+12]
C
D
12

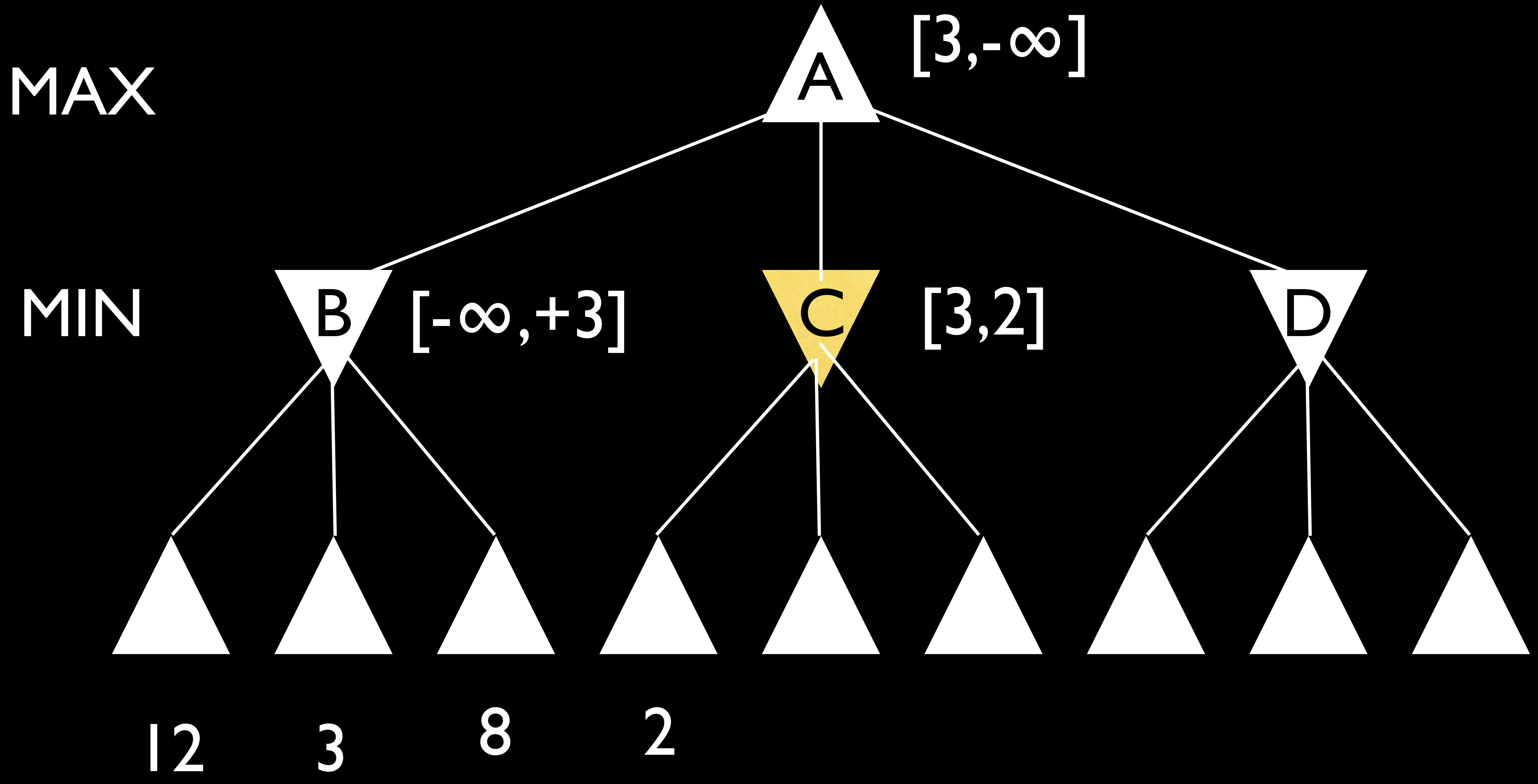

MAX
MIN
A
[-∞,+∞]
B
[-∞,+3]
C
D
12
3

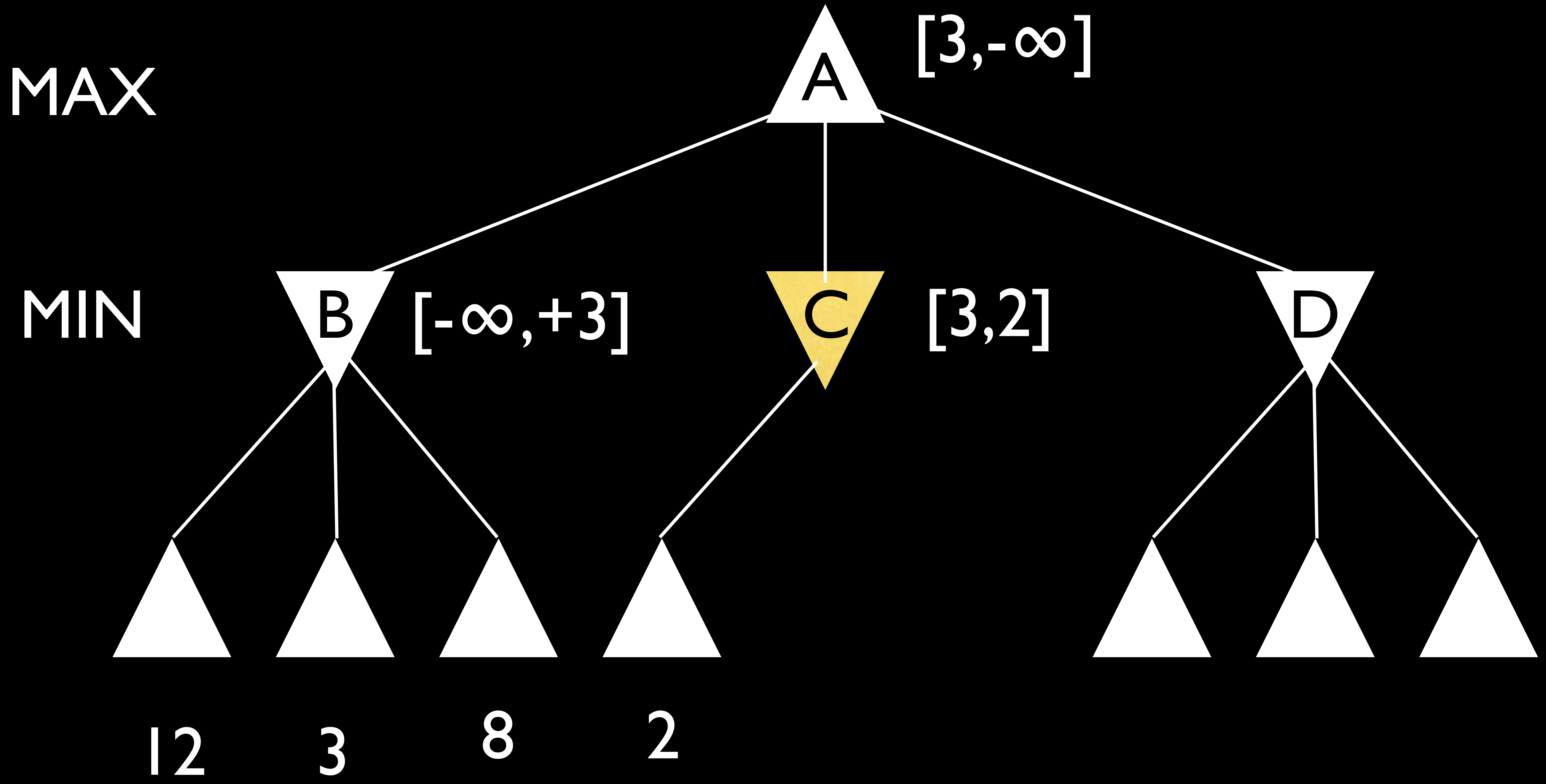

MAX
MIN
A
[-∞,+∞]
B
[-∞,+3]
C
D
12
3
8

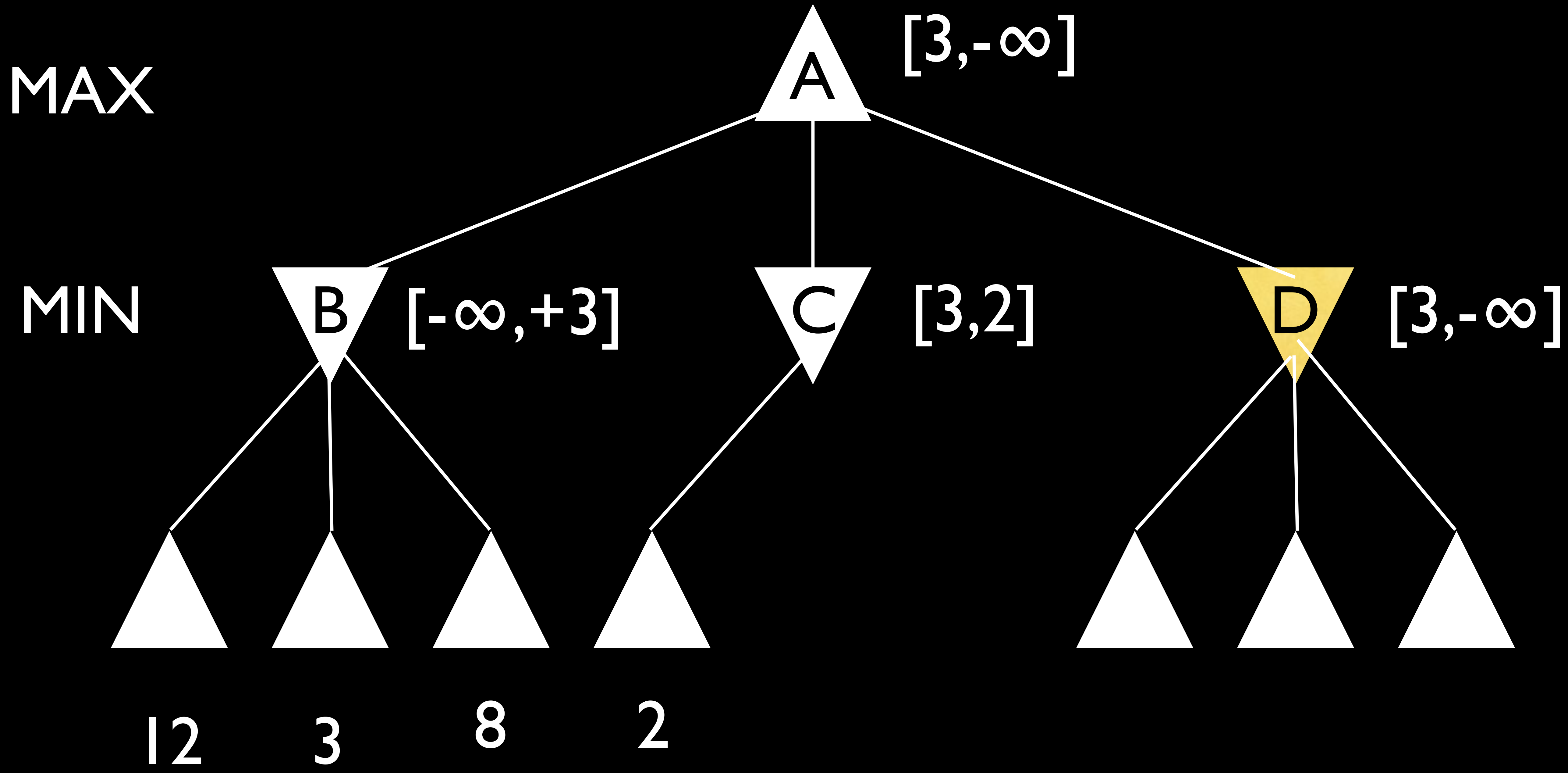

MAX
MIN
A
[3,-∞]
B
[-∞,+3]
C
D
12
3
8

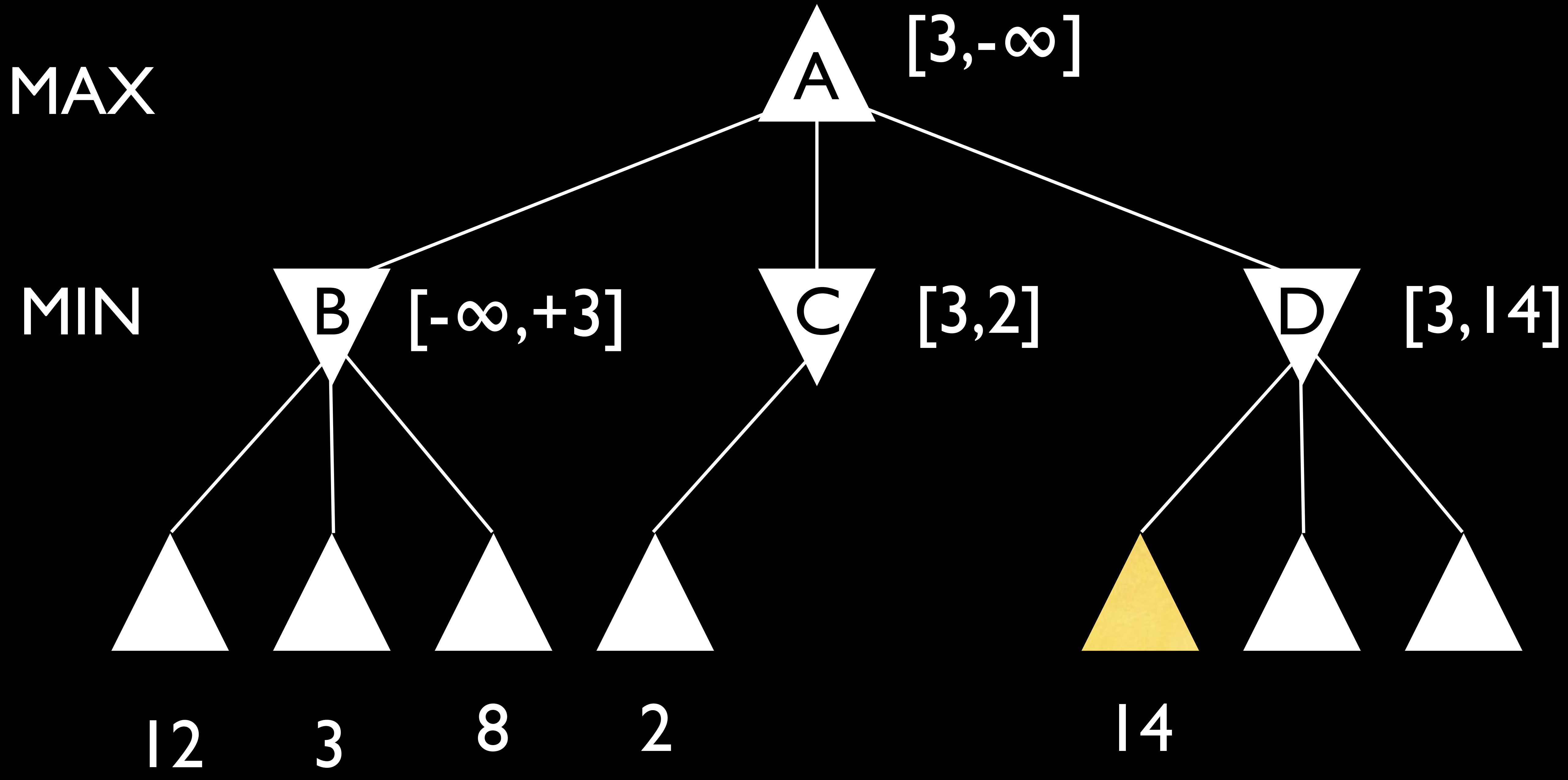

MAX
MIN
A
[3,-∞]
B
[-∞,+3]
C
[3,-∞]
D
12
3
8

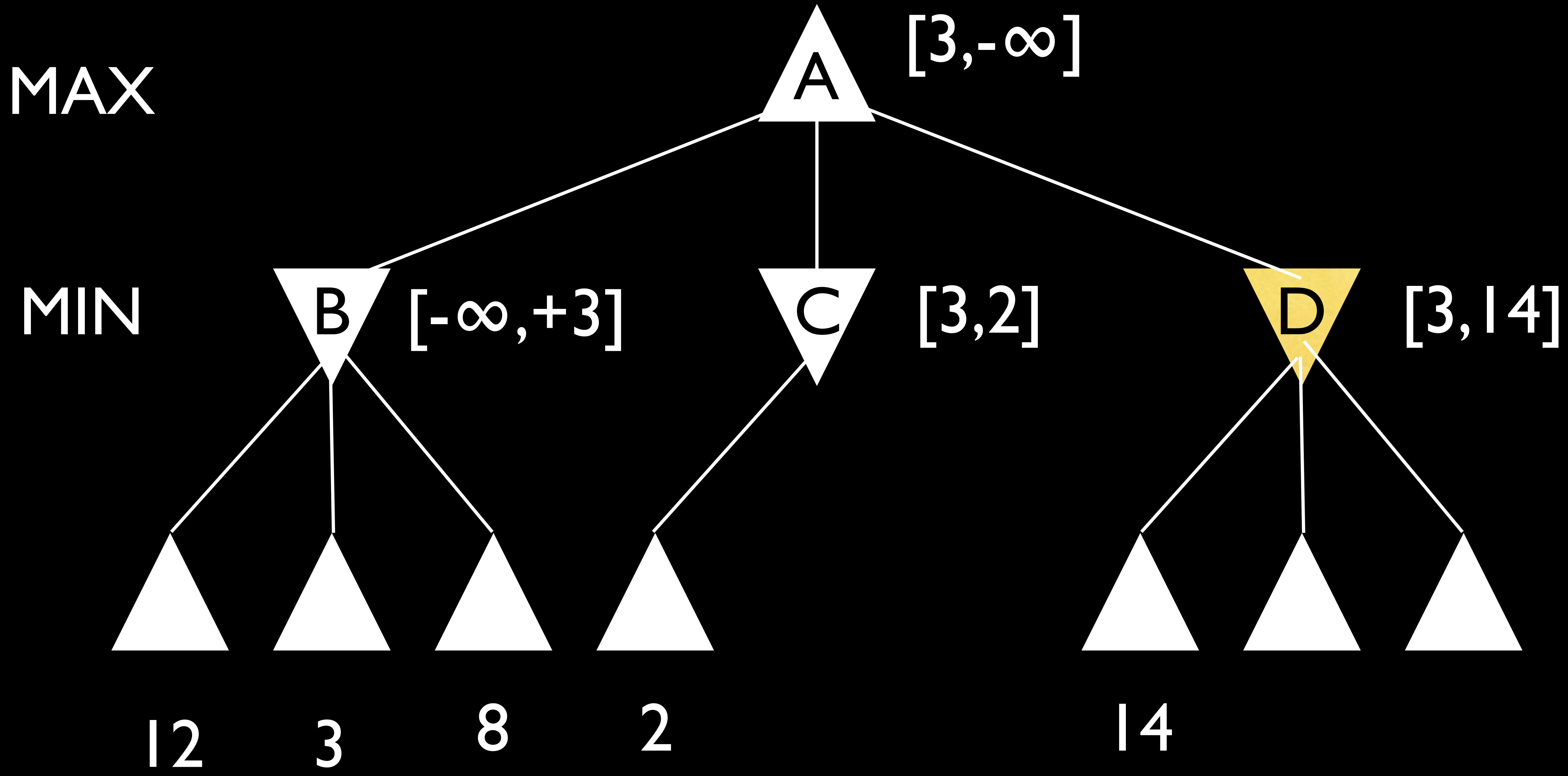

MAX
MIN
A
[3,-∞]
B
[-∞,+3]
C
[3,-∞]
D
12
3
8
2

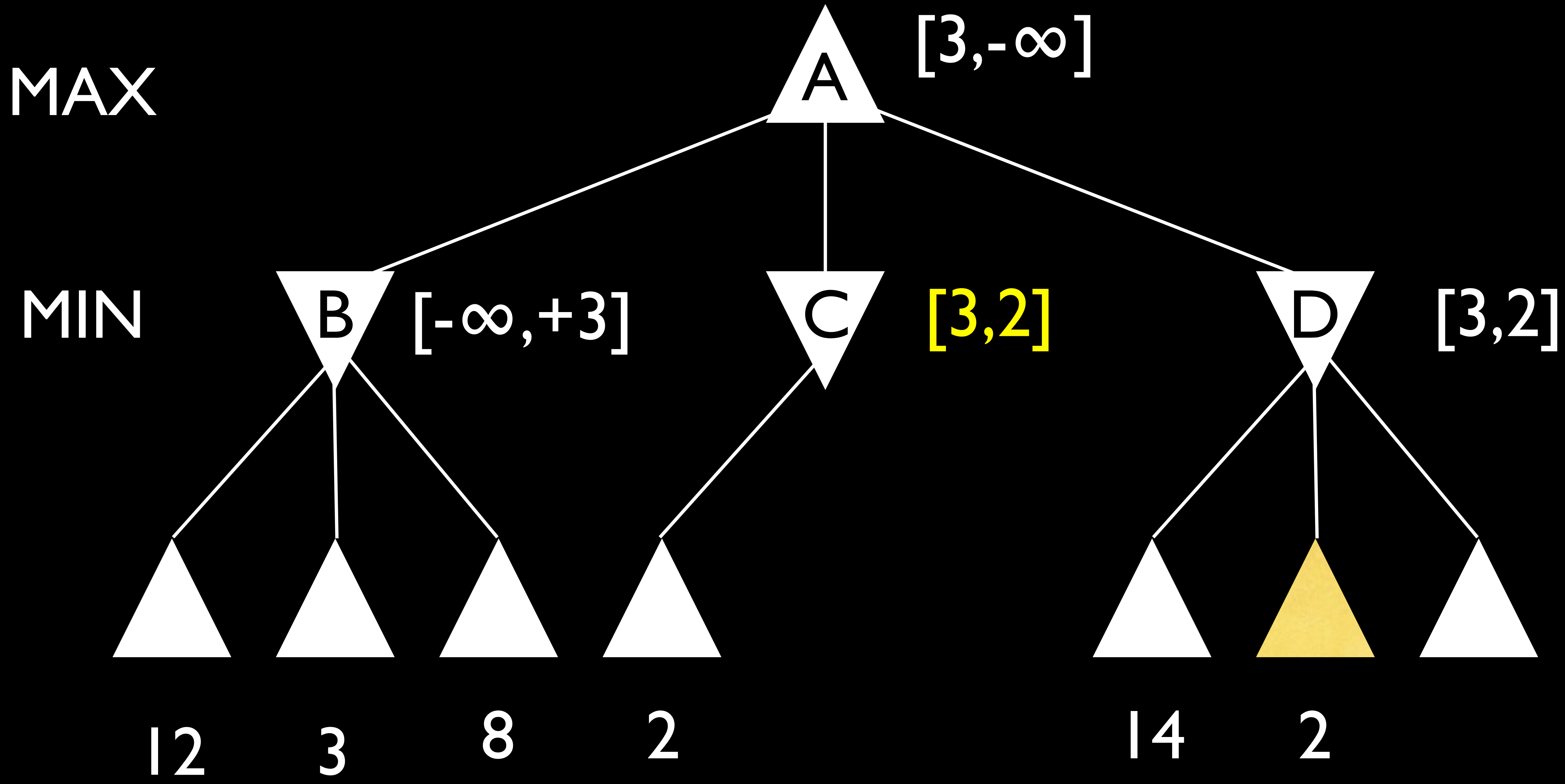

MAX
MIN
A
[3,-∞]
B
[-∞,+3]
C
[3,2]
D
12
3
8
2

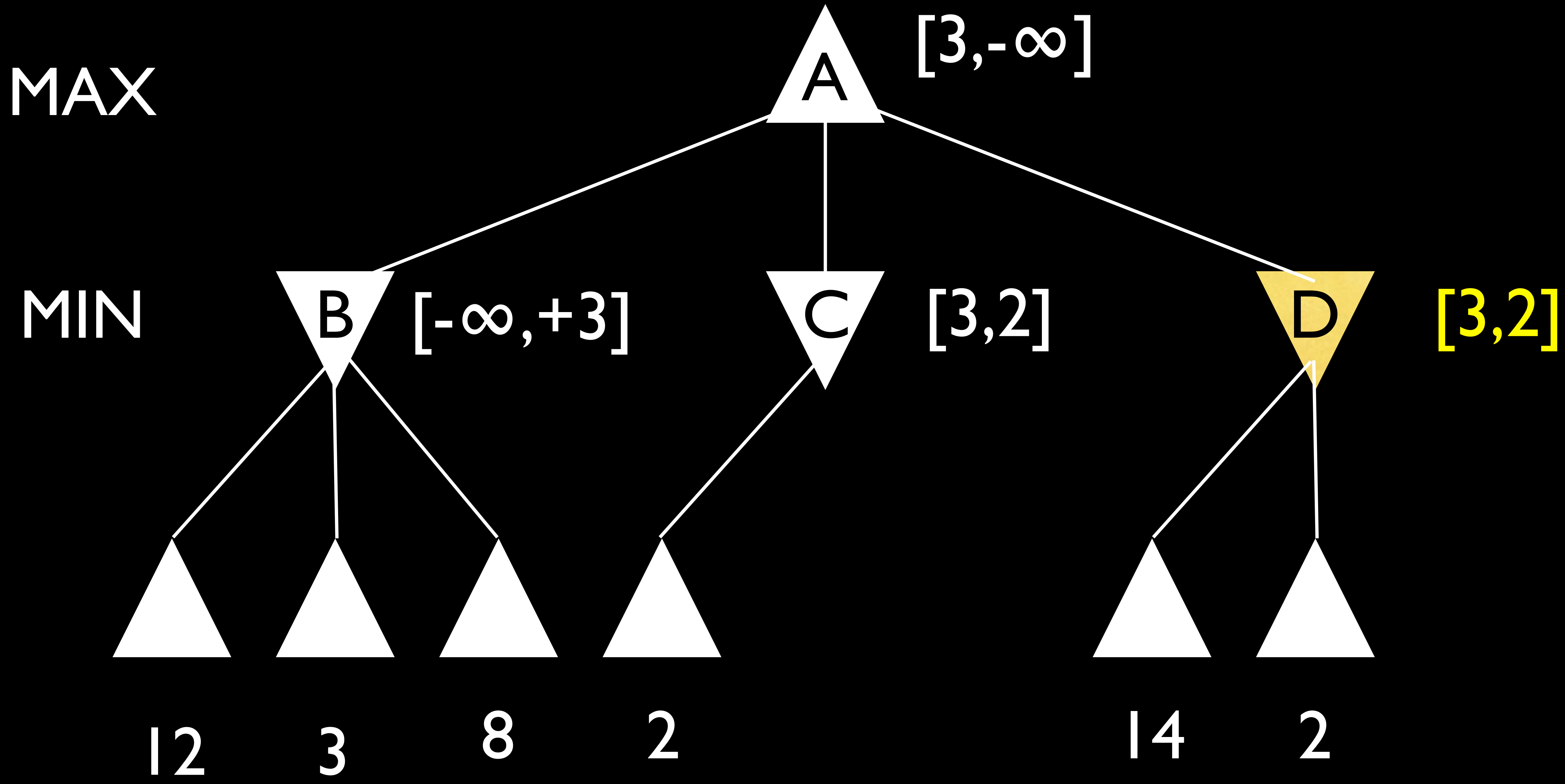

MAX
MIN
A
[3,-∞]
B
[-∞,+3]
C
[3,2]
D
12
3
8
2

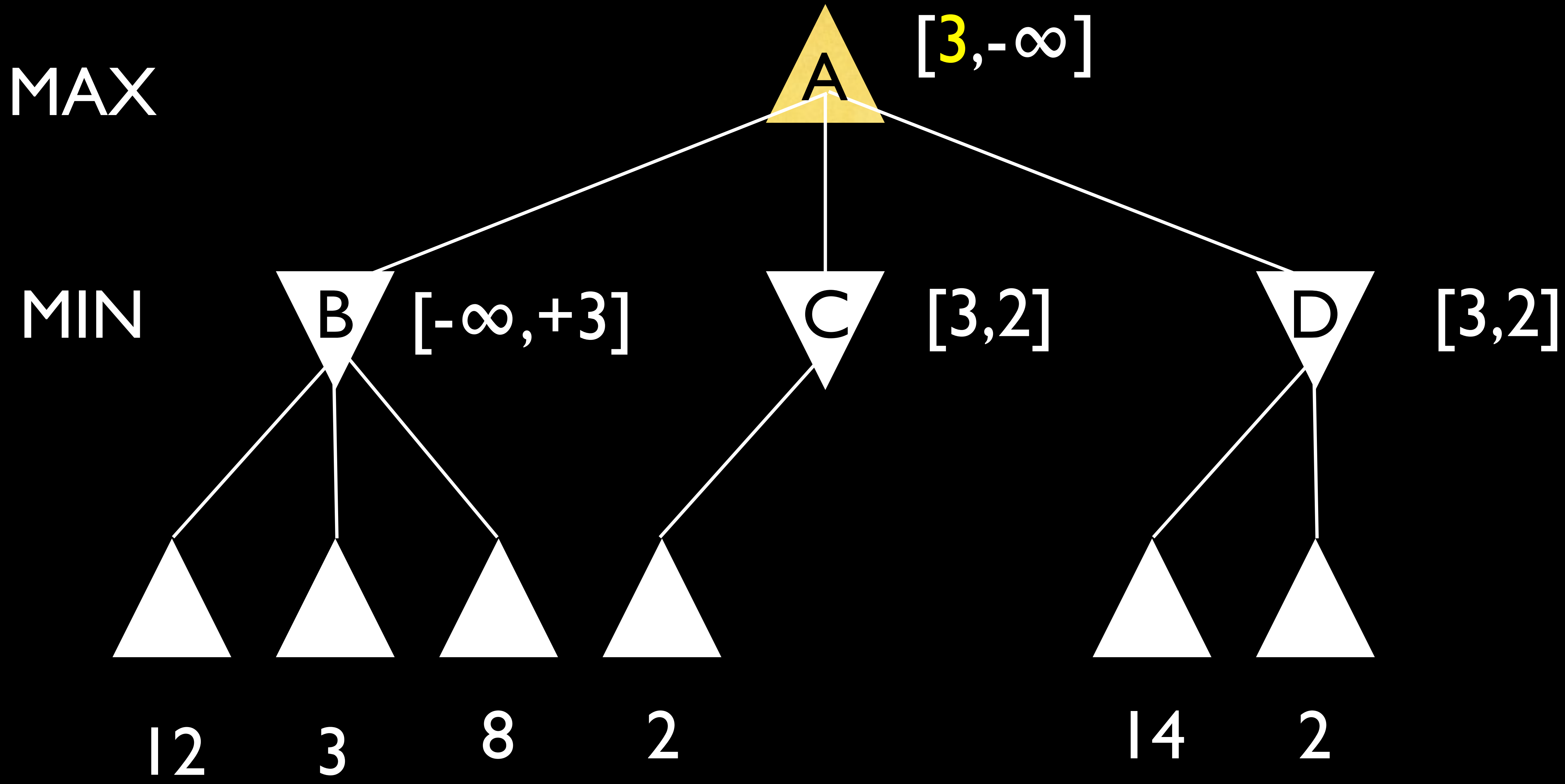

MAX
A
[3,-∞]
MIN
B
[-∞,+3]
C
[3,2]
D
[3,-∞]
12
3
8
2


MAX
MIN
A
[3,-∞]
B
[-∞,+3]
C
[3,2]
D
[3,14]
12
3
8
2
14


MAX
MIN
A
[3,-∞]
B
[-∞,+3]
C
[3,2]
D
[3,14]
12
3
8
2
14


MAX
MIN
A
[3,-∞]
B
[-∞,+3]
C
[3,2]
D
[3,2]
12
3
8
2
14
2


MAX
MIN
A
[3,-∞]
B
[-∞,+3]
C
[3,2]
D
[3,2]
12
3
8
2
14
2


MAX
MIN
A
[3,-∞]
B
[-∞,+3]
C
[3,2]
D
[3,2]
12
3
8
2
14
2

Alpha-Beta H-Minimax

```
AlphaBeta(origin, 0, -inf, +inf, player)

function AlphaBeta(state, depth, alpha, beta, player)
    if CutoffTest(state, depth) then
        return Eval(State)
    if player == +1 then
        for each action in Actions(state, player)
            beta = max(beta, AlphaBeta(Result(state,action),
                   depth+1, alpha, beta, -player))
            if beta <= alpha then break /* beta cut-off */
        return alpha
    else /* player == -1 */
        for each action in Actions(state, player)
            alpha = min(alpha,AlphaBeta(Result(state,action),
                    depth+1, alpha, beta, -player))
            if beta <= alpha then break /* alpha cut-off */
        return beta
 end
```

Alpha-Beta H-Minimax

```
AlphaBeta(origin, 0, -inf, +inf, player)

function AlphaBeta(state, depth, alpha, beta, player)
    if CutoffTest(state, depth) then
        return Eval(State)
    if player == +1 then
        for each action in Actions(state, player)
            beta = max(beta, AlphaBeta(Result(state,action),
                   depth+1, alpha, beta, -player))
            if beta <= alpha then break /* beta cut-off */
        return alpha
    else /* player == -1 */
        for each action in Actions(state, player)
            alpha = min(alpha,AlphaBeta(Result(state,action),
                    depth+1, alpha, beta, -player))
            if beta <= alpha then break /* alpha cut-off */
        return beta
 end
```

Alpha-Beta H-Minimax

```
AlphaBeta(origin, 0, -inf, +inf, player)

function AlphaBeta(state, depth, alpha, beta, player)
    if CutoffTest(state, depth) then
        return Eval(State)
    if player == +1 then
        for each action in Actions(state, player)
            beta = max(beta, AlphaBeta(Result(state,action),
                   depth+1, alpha, beta, -player))
            if beta <= alpha then break /* beta cut-off */
        return alpha
    else /* player == -1 */
        for each action in Actions(state, player)
            alpha = min(alpha,AlphaBeta(Result(state,action),
                    depth+1, alpha, beta, -player))
            if beta <= alpha then break /* alpha cut-off */
        return beta
 end
```

Alpha-Beta H-Minimax

```
AlphaBeta(origin, 0, -inf, +inf, player)

function AlphaBeta(state, depth, alpha, beta, player)
    if CutoffTest(state, depth) then
        return Eval(State)
    if player == +1 then
        for each action in Actions(state, player)
            beta = max(beta, AlphaBeta(Result(state,action),
                   depth+1, alpha, beta, -player))
            if beta <= alpha then break /* beta cut-off */
        return alpha
    else /* player == -1 */
        for each action in Actions(state, player)
            alpha = min(alpha,AlphaBeta(Result(state,action),
                    depth+1, alpha, beta, -player))
            if beta <= alpha then break /* alpha cut-off */
        return beta
 end
```

Alpha-Beta H-Minimax

```
AlphaBeta(origin, 0, -inf, +inf, player)

function AlphaBeta(state, depth, alpha, beta, player)
    if CutoffTest(state, depth) then
        return Eval(State)
    if player == +1 then
        for each action in Actions(state, player)
            beta = max(beta, AlphaBeta(Result(state,action),
                   depth+1, alpha, beta, -player))
            if beta <= alpha then break /* beta cut-off */
        return alpha
    else /* player == -1 */
        for each action in Actions(state, player)
            alpha = min(alpha,AlphaBeta(Result(state,action),
                    depth+1, alpha, beta, -player))
            if beta <= alpha then break /* alpha cut-off */
        return beta
 end
```

Alpha-Beta H-Minimax

```
AlphaBeta(origin, 0, -inf, +inf, player)

function AlphaBeta(state, depth, alpha, beta, player)
    if CutoffTest(state, depth) then
        return Eval(State)
    if player == +1 then
        for each action in Actions(state, player)
            beta = max(beta, AlphaBeta(Result(state,action),
                   depth+1, alpha, beta, -player))
            if beta <= alpha then break /* beta cut-off */
        return alpha
    else /* player == -1 */
        for each action in Actions(state, player)
            alpha = min(alpha,AlphaBeta(Result(state,action),
                    depth+1, alpha, beta, -player))
            if beta <= alpha then break /* alpha cut-off */
        return beta
 end
```

Alpha-Beta H-Minimax

```
AlphaBeta(origin, 0, -inf, +inf, player)

function AlphaBeta(state, depth, alpha, beta, player)
    if CutoffTest(state, depth) then
        return Eval(State)
    if player == +1 then
        for each action in Actions(state, player)
            beta = max(beta, AlphaBeta(Result(state,action),
                   depth+1, alpha, beta, -player))
            if beta <= alpha then break /* beta cut-off */
        return alpha
    else /* player == -1 */
        for each action in Actions(state, player)
            alpha = min(alpha,AlphaBeta(Result(state,action),
                    depth+1, alpha, beta, -player))
            if beta <= alpha then break /* alpha cut-off */
        return beta
 end
```

Alpha-Beta H-Minimax

```
AlphaBeta(origin, 0, -inf, +inf, player)

function AlphaBeta(state, depth, alpha, beta, player)
    if CutoffTest(state, depth) then
        return Eval(State)
    if player == +1 then
        for each action in Actions(state, player)
            beta = max(beta, AlphaBeta(Result(state,action),
                       depth+1, alpha, beta, -player))
            if beta <= alpha then break /* beta cut-off */
        return alpha
    else /* player == -1 */
        for each action in Actions(state, player)
            alpha = min(alpha,AlphaBeta(Result(state,action),
                        depth+1, alpha, beta, -player))
            if beta <= alpha then break /* alpha cut-off */
        return beta
 end
```

Alpha-Beta H-Minimax

```
AlphaBeta(origin, 0, -inf, +inf, player)

function AlphaBeta(state, depth, alpha, beta, player)
    if CutoffTest(state, depth) then
        return Eval(State)
    if player == +1 then
        for each action in Actions(state, player)
            beta = max(beta, AlphaBeta(Result(state,action),
                   depth+1, alpha, beta, -player))
            if beta <= alpha then break /* beta cut-off */
        return alpha
    else /* player == -1 */
        for each action in Actions(state, player)
            alpha = min(alpha,AlphaBeta(Result(state,action),
                    depth+1, alpha, beta, -player))
            if beta <= alpha then break /* alpha cut-off */
        return beta
 end
```

Alpha-Beta H-Minimax

```
AlphaBeta(origin, 0, -inf, +inf, player)

function AlphaBeta(state, depth, alpha, beta, player)
    if CutoffTest(state, depth) then
        return Eval(State)
    if player == +1 then
        for each action in Actions(state, player)
            beta = max(beta, AlphaBeta(Result(state,action),
                   depth+1, alpha, beta, -player))
            if beta <= alpha then break /* beta cut-off */
        return alpha
    else /* player == -1 */
        for each action in Actions(state, player)
            alpha = min(alpha,AlphaBeta(Result(state,action),
                    depth+1, alpha, beta, -player))
            if beta <= alpha then break /* alpha cut-off */
        return beta
 end
```

Alpha-Beta H-Minimax Returning Move

```
function AlphaBeta(state, depth, alpha, beta, player)
    if CutoffTest(state, depth) then return (Eval(State), null)
    best = null /* to handle case where no move is possible */
    if player == +1 then
        for each action in Actions(state, player)
            child = Result(state, action)
            value = AlphaBeta(child, depth+1, alpha, beta, -player)
            if value > alpha then
                alpha = value
                best = action
            if beta <= alpha then break /* beta cut-off */
        return (alpha, best)
    else /* player == -1 */
        for each action in Actions(state, player)
            child = Result(state, action)
            value = AlphaBeta(child, depth+1, alpha, beta, -player)
            if value < beta then
                beta = value
                best = action
            if beta <= alpha then break /* alpha cut-off */
        return (beta, best)
 en
```

Alpha-Beta H-Minimax Returning Move

```
function AlphaBeta(state, depth, alpha, beta, player)
    if CutoffTest(state, depth) then return (Eval(State), null)
    best = null /* to handle case where no move is possible */
    if player == +1 then
        for each action in Actions(state, player)
            child = Result(state, action)
            value = AlphaBeta(child, depth+1, alpha, beta, -player)
            if value > alpha then
                alpha = value
                best = action
            if beta <= alpha then break /* beta cut-off */
        return (alpha, best)
    else /* player == -1 */
        for each action in Actions(state, player)
            child = Result(state, action)
            value = AlphaBeta(child, depth+1, alpha, beta, -player)
            if value < beta then
                beta = value
                best = action
            if beta <= alpha then break /* alpha cut-off */
        return (beta, best)
 en
```

Alpha-Beta H-Minimax Returning Move

```
function AlphaBeta(state, depth, alpha, beta, player)
    if CutoffTest(state, depth) then return (Eval(State), null)
    best = null /* to handle case where no move is possible */
    if player == +1 then
        for each action in Actions(state, player)
            child = Result(state, action)
            value = AlphaBeta(child, depth+1, alpha, beta, -player)
            if value > alpha then
                alpha = value
                best = action
            if beta <= alpha then break /* beta cut-off */
        return (alpha, best)
    else /* player == -1 */
        for each action in Actions(state, player)
            child = Result(state, action)
            value = AlphaBeta(child, depth+1, alpha, beta, -player)
            if value < beta then
                beta = value
                best = action
            if beta <= alpha then break /* alpha cut-off */
        return (beta, best)
 en
```

Alpha-Beta H-Minimax Returning Move

```
function AlphaBeta(state, depth, alpha, beta, player)
    if CutoffTest(state, depth) then return (Eval(State), null)
    best = null /* to handle case where no move is possible */
    if player == +1 then
        for each action in Actions(state, player)
            child = Result(state, action)
            value = AlphaBeta(child, depth+1, alpha, beta, -player)
            if value > alpha then
                alpha = value
                best = action
            if beta <= alpha then break /* beta cut-off */
        return (alpha, best) /* best is ignored if not needed */
    else /* player == -1 */
        for each action in Actions(state, player)
            child = Result(state, action)
            value = AlphaBeta(child, depth+1, alpha, beta, -player)
            if value < beta then
                beta = value
                best = action
            if beta <= alpha then break /* alpha cut-off */
        return (beta, best)
 en
```

Alpha-Beta Pruning

- Still MINIMAX search
  - Optimal (if you search to the terminals)
  - Optimal with respect to your heuristic function otherwise

Alpha-Beta Pruning Analysis

Alpha-Beta Pruning Analysis

Ideal case:

Always explore the best successor first: $O(b^{m/2})$

Branching factor: $b^{1/2} = \sqrt{b}$

Explore twice as deep a tree in same time

Alpha-Beta Pruning Analysis

Random case:

Explore successors in random order: $O(b^{3m/4})$

Branching factor: $b^{3/4}$

Explore 4/3 as deep a tree in same time

Alpha-Beta Pruning Analysis

"Smart" case:

Example: in chess, expand a successor early if it is a capture

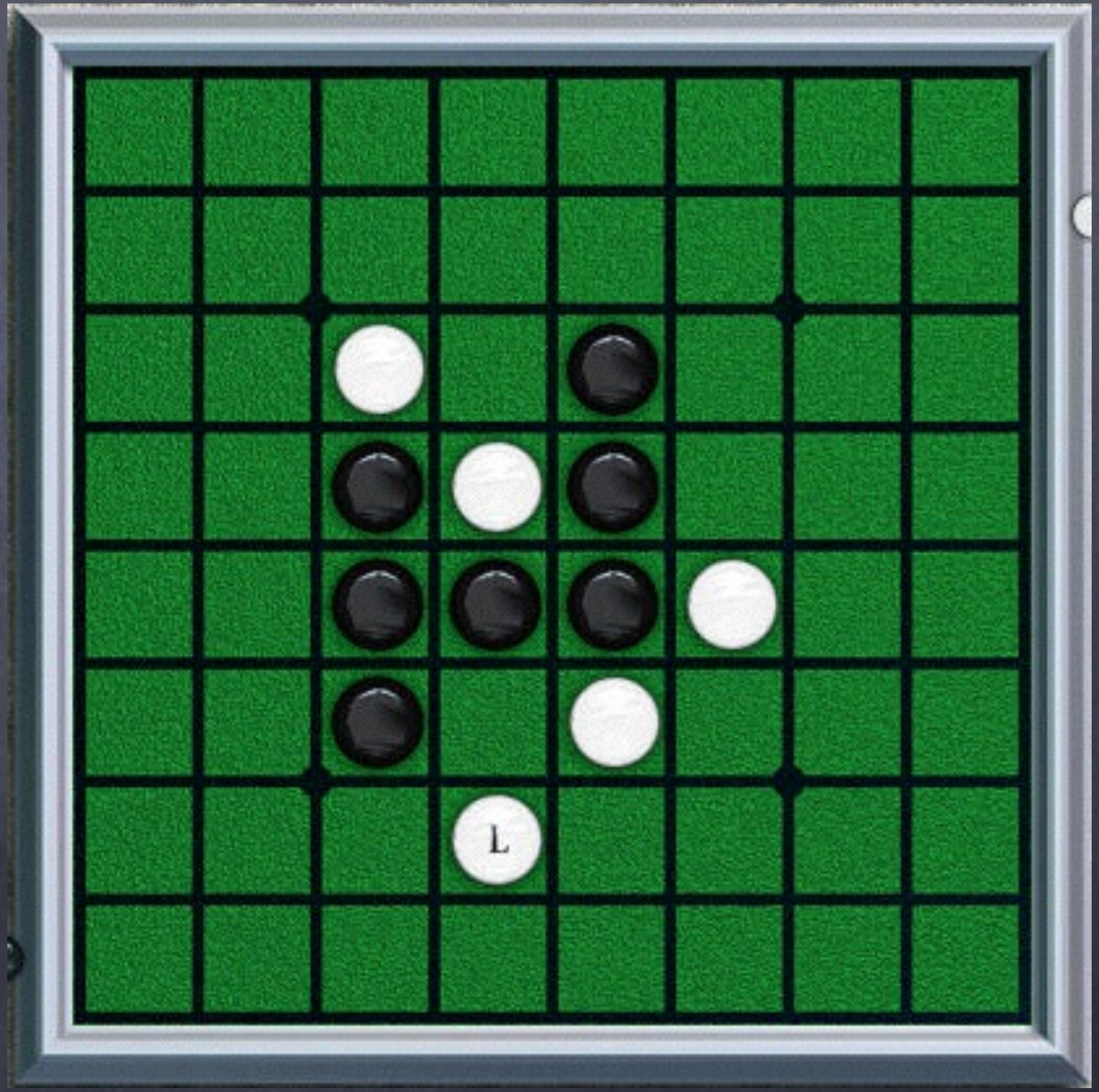
Use the static evaluation function to determine the order for expanding children

Alpha-Beta Summary

- Easy bookkeeping modification of basic MINIMAX algorithm
- Not hard to come up with "useful" node orderings
- Even random gets you 33% deeper search
- Works with other ways of improving game tree search

Project 1: Othello (Reversi)

- Write a (champion!) Othello player
- Phase I: 4-Ply Alpha-Beta Search
- Phase II: Time-limited Search
- Phase III: Time-budgeted Search



What We Give You

- Othello GUI program
- Binary code for a player (to play against your program)
- Pseudocode for alpha-beta heuristic game tree search

What You Give Us

- Othello player program
- Reads moves from <stdin>, writes moves to <stdout>
- Runs on Linux (at least)
- Written in any language - Java, C++, Python, Ruby, Lisp, ...
- Write-up describing your design choices (worth 25% of project grade)

Input / Output Language

- Input: Moves by opponent
    - 1 8 # upper left hand corner
- Output: Moves by player
    - 8 1 # lower right hand corner
- Our program determines if a player wins or makes and illegal moves (immediate loss)

Project Subtasks

- Choose data structures and programming language
- Turn alpha-beta pseudocode into real code
- Design, implement, and test
    - Move generator (Actions(state,player))
    - Transition function (Apply(state,action))
    - Static evaluator (Eval(state))
    - Cut-off test (CutoffTest(state,depth))

Phases

- Phase I tests correct implementation of move generator and your static evaluation function
- Phases II and III will require a more sophisticated cut-off function
  - Use your time per move or time budget for game wisely
  - Real-time decision making!

Teams

- Posted by 7pm tonight on course web page:
  - 2 person teams (randomly assigned by me)
  - Specification of the <stdio> input/output language
- GUI is currently in beta-test - email Dan Scarafoni <dscarafo@u.rochester.edu> to join the beta test and/or report bugs
  - Final release no later than Sunday

GUI Demo (?)

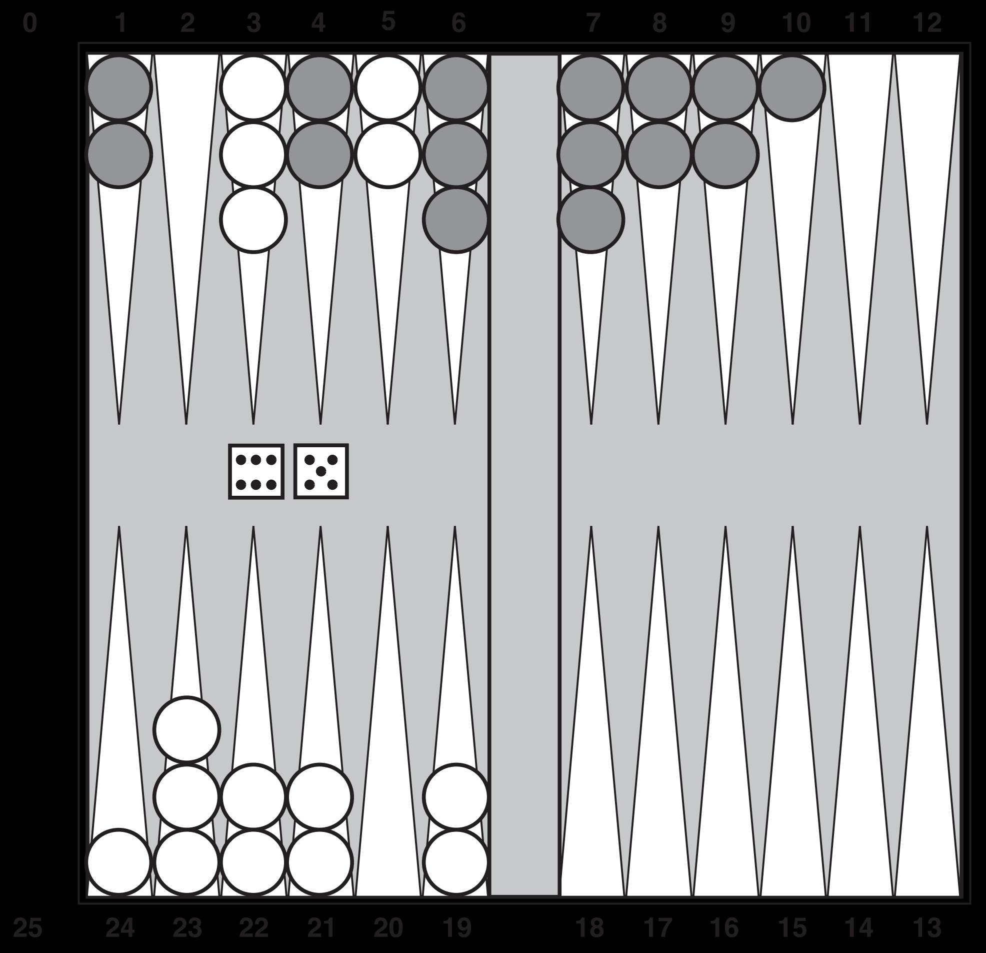
Beyond Deterministic Games of Perfect Information

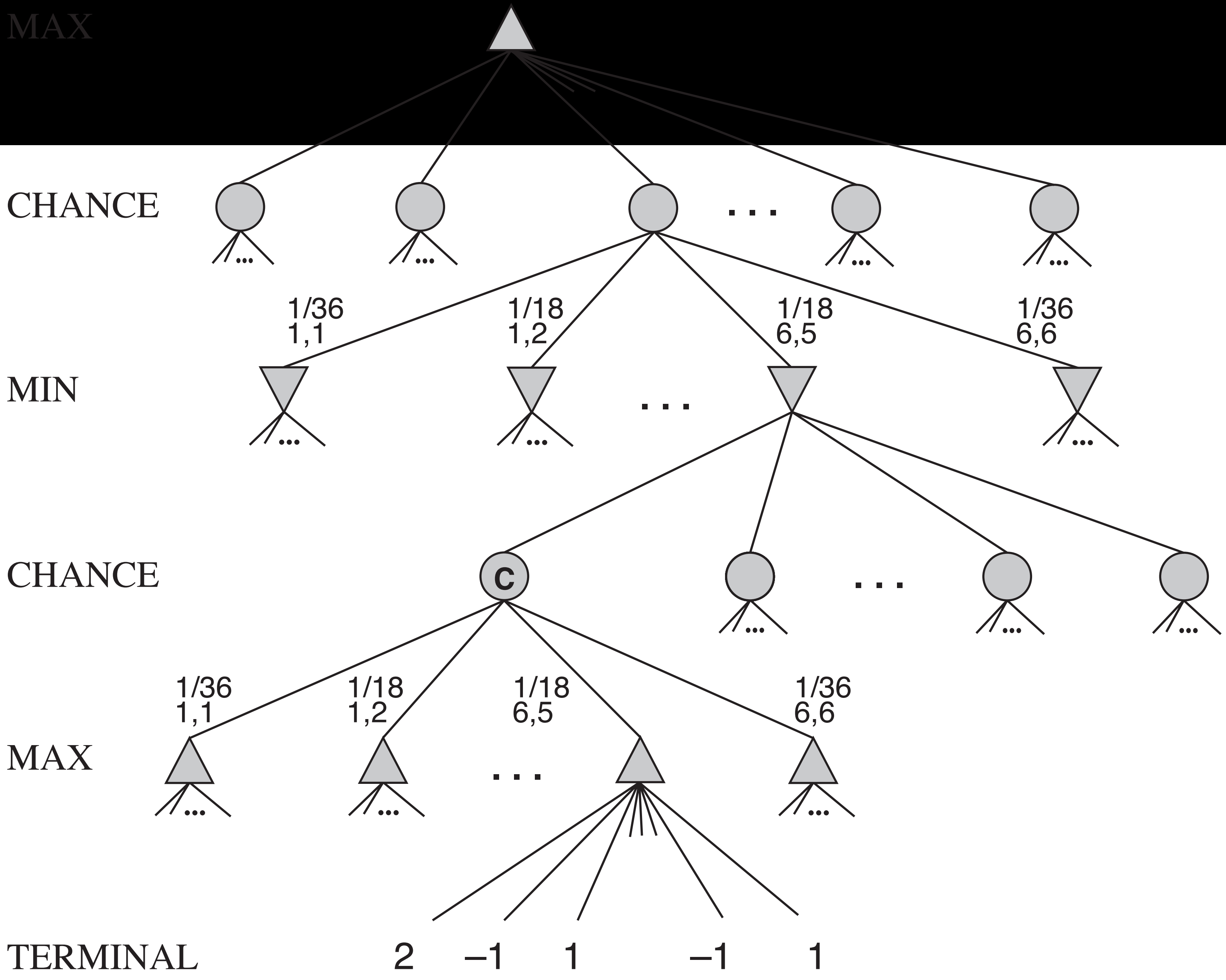
Types of Games

Deterministic (no chance)	Nondeterministic (dice, cards, etc.)
Perfect information (fully observable)	Imperfect information (partially observable)
Zero-sum (total payoff the same in any game)	Arbitrary utility functions

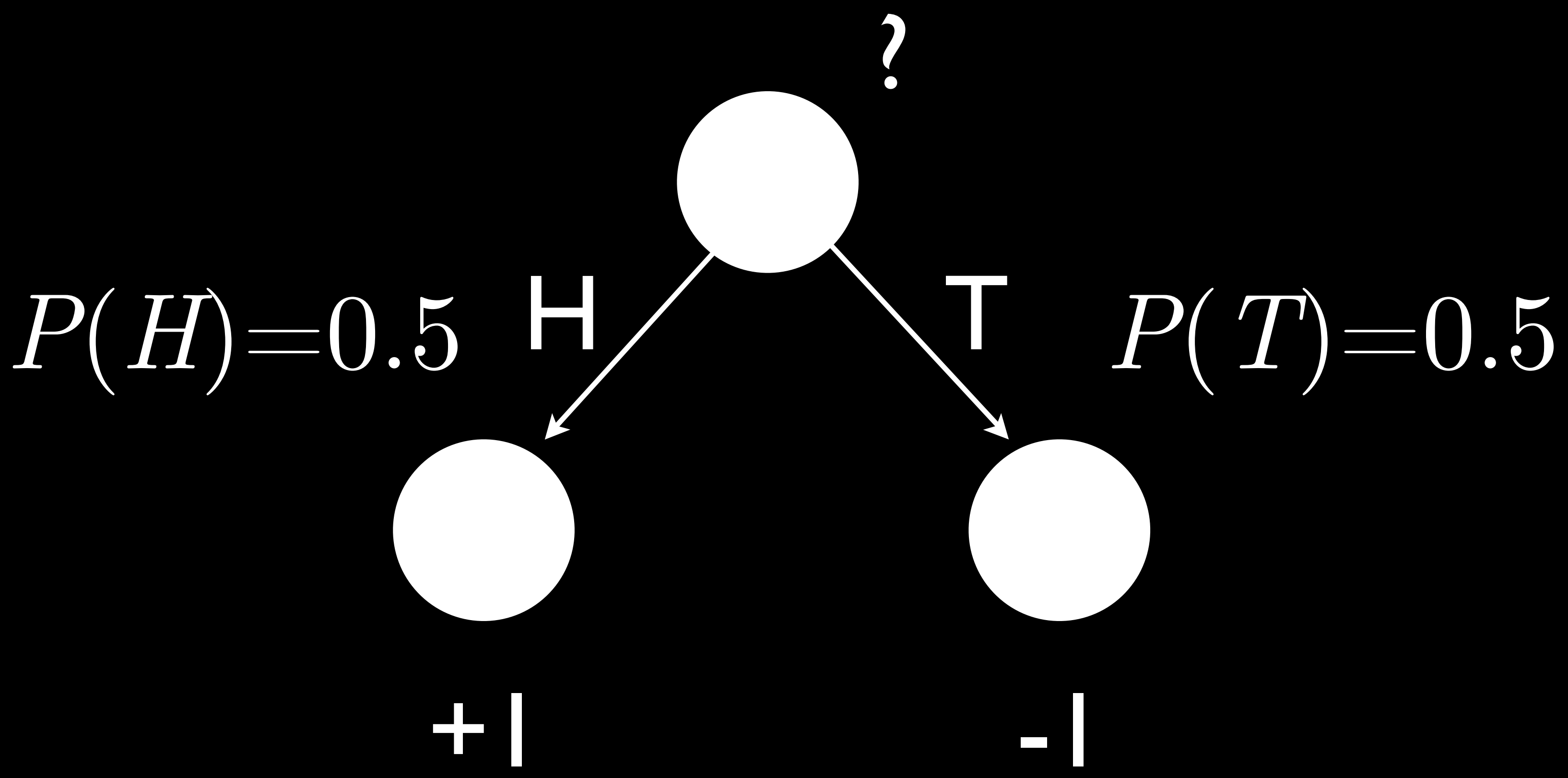
Stochastic Games

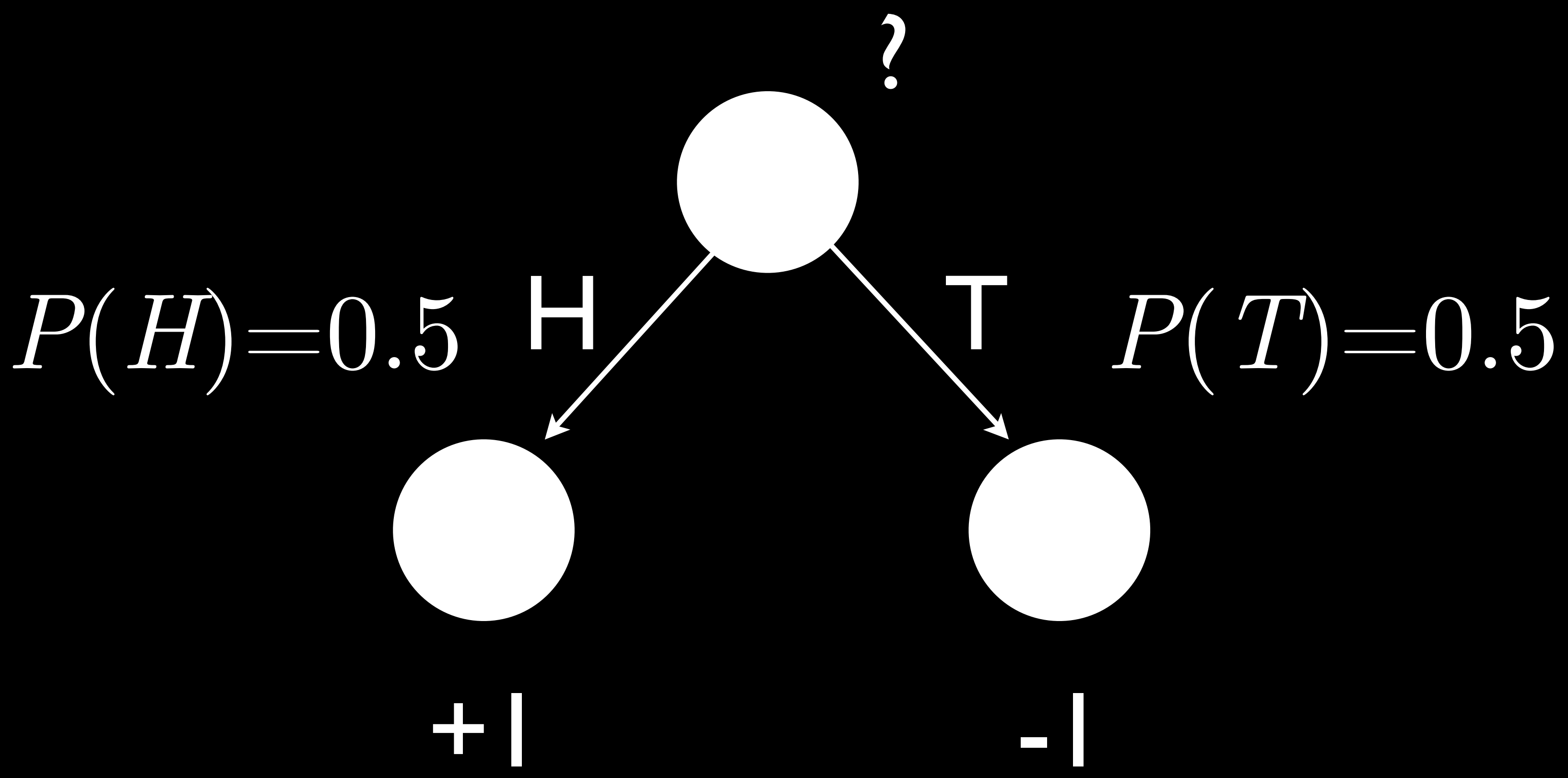
- A player's possible moves depend on chance (random) elements, e.g., dice
- Must generalize our notion of a game tree to include "chance" nodes

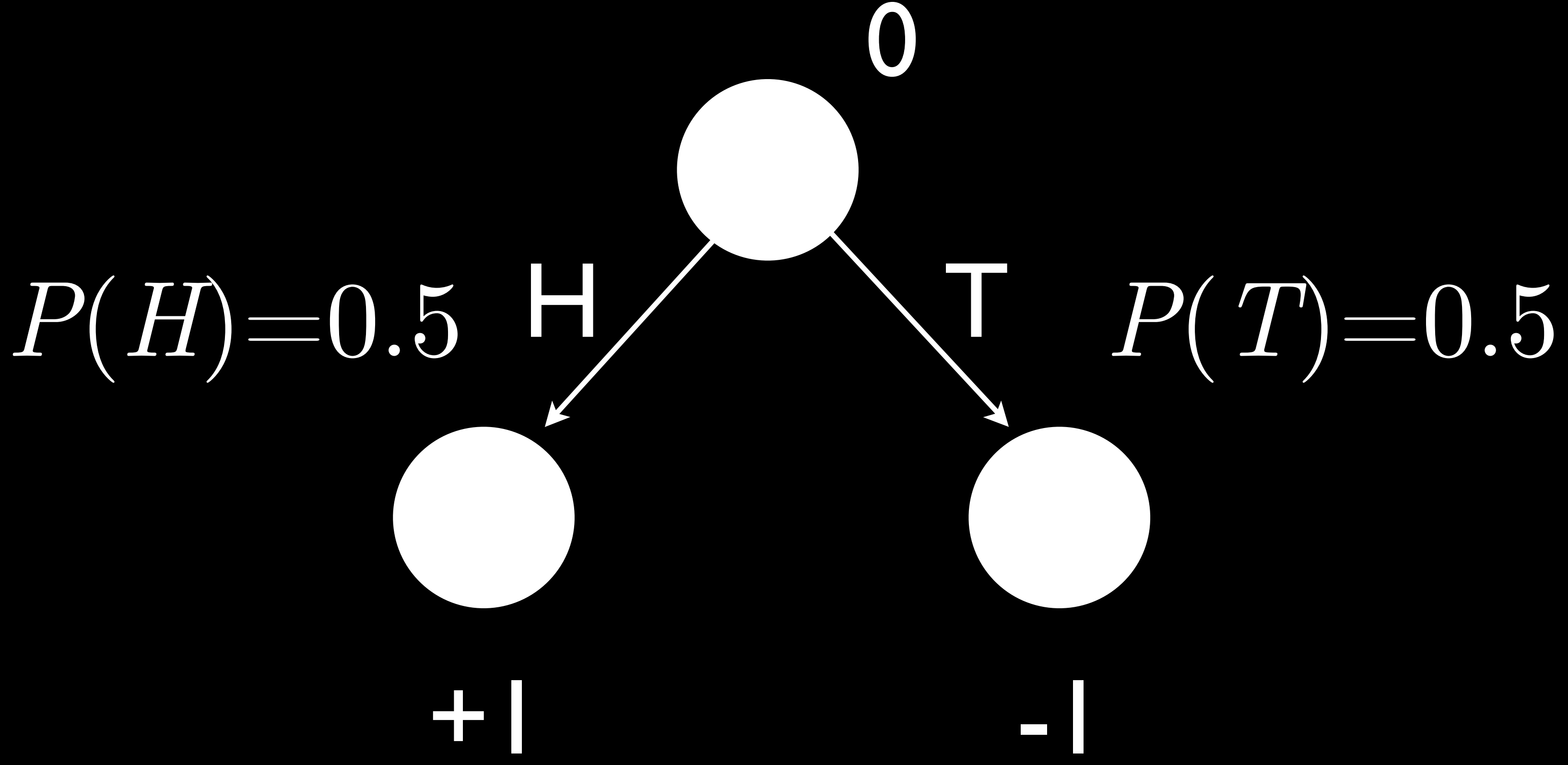


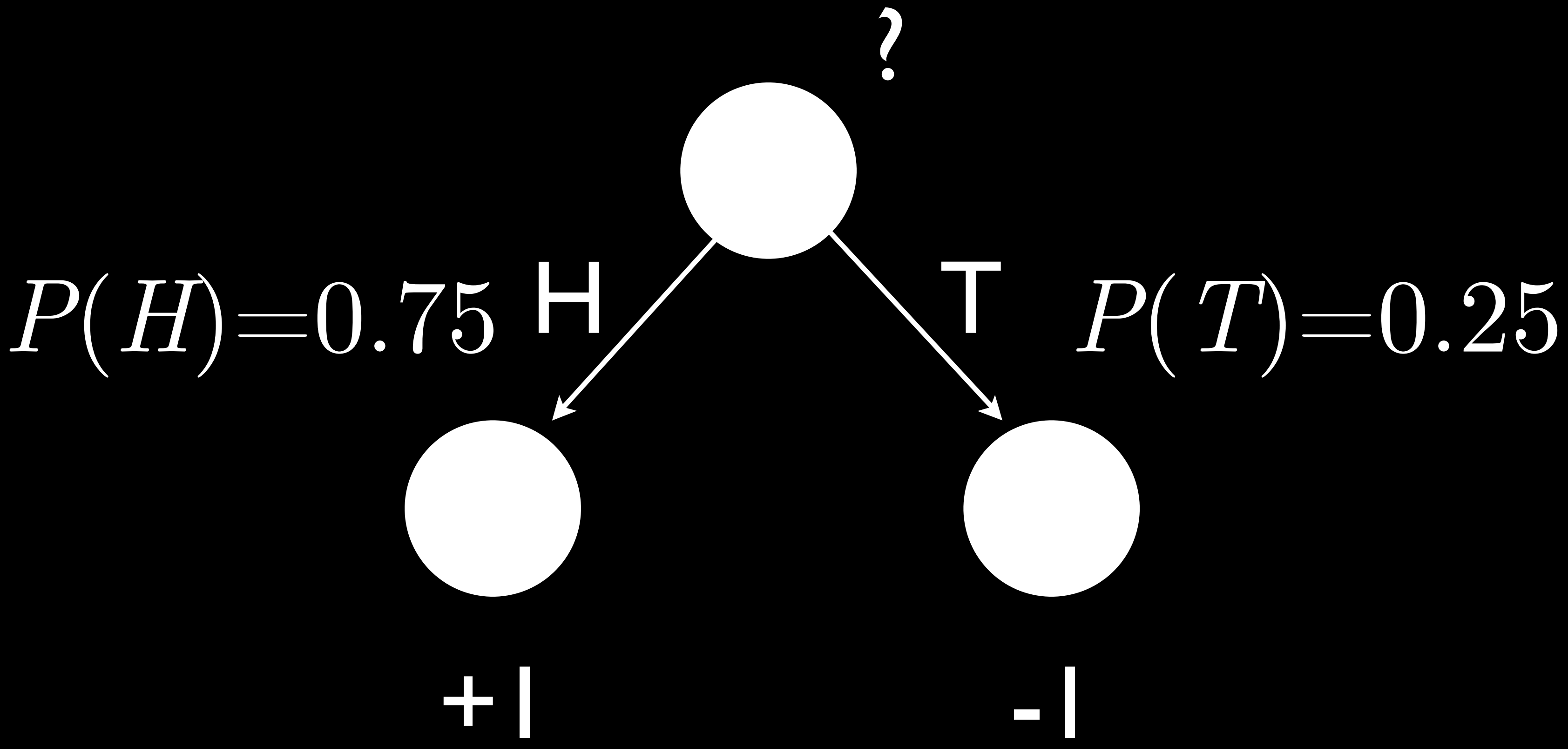


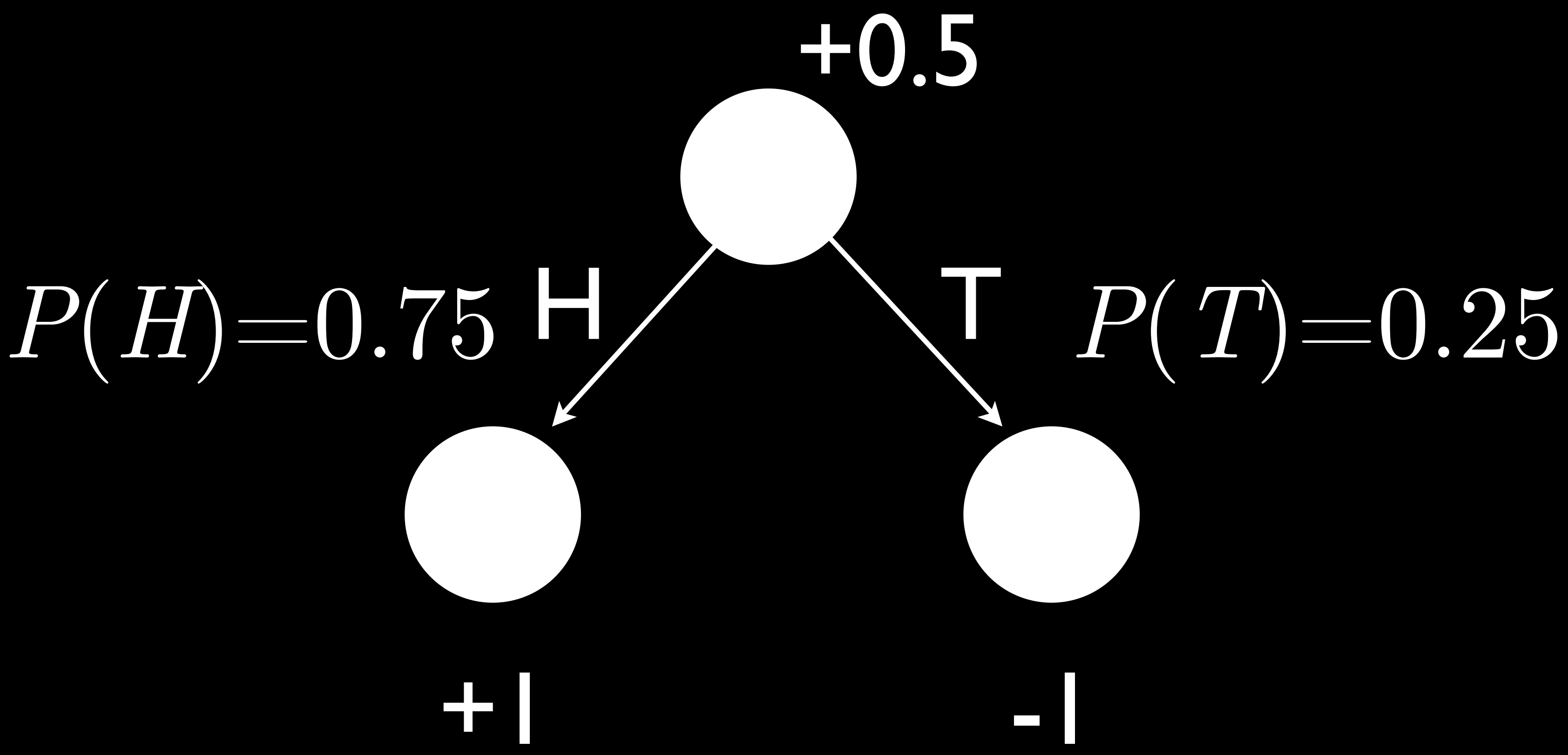
MAX
CHANCE
MIN
CHANCE
MAX
TERMINAL
1/36 1,1
1/18 1,2
1/18 6,5
1/36 6,6
C
1/36 1,1
1/18 1,2
1/18 6,5
1/36 6,6
2
–1
1
–1
1


?
H
T
$P(H)=0.5$
$P(T)=0.5$
+1
-1


?
H
T
$P(H) = 0.5$
$P(T) = 0.5$
+1
-1


0
H
T
$P(H)=0.5$
$P(T)=0.5$
+1
-1


?
H
T
$P(H)=0.75$
$P(T)=0.25$
+1
-1



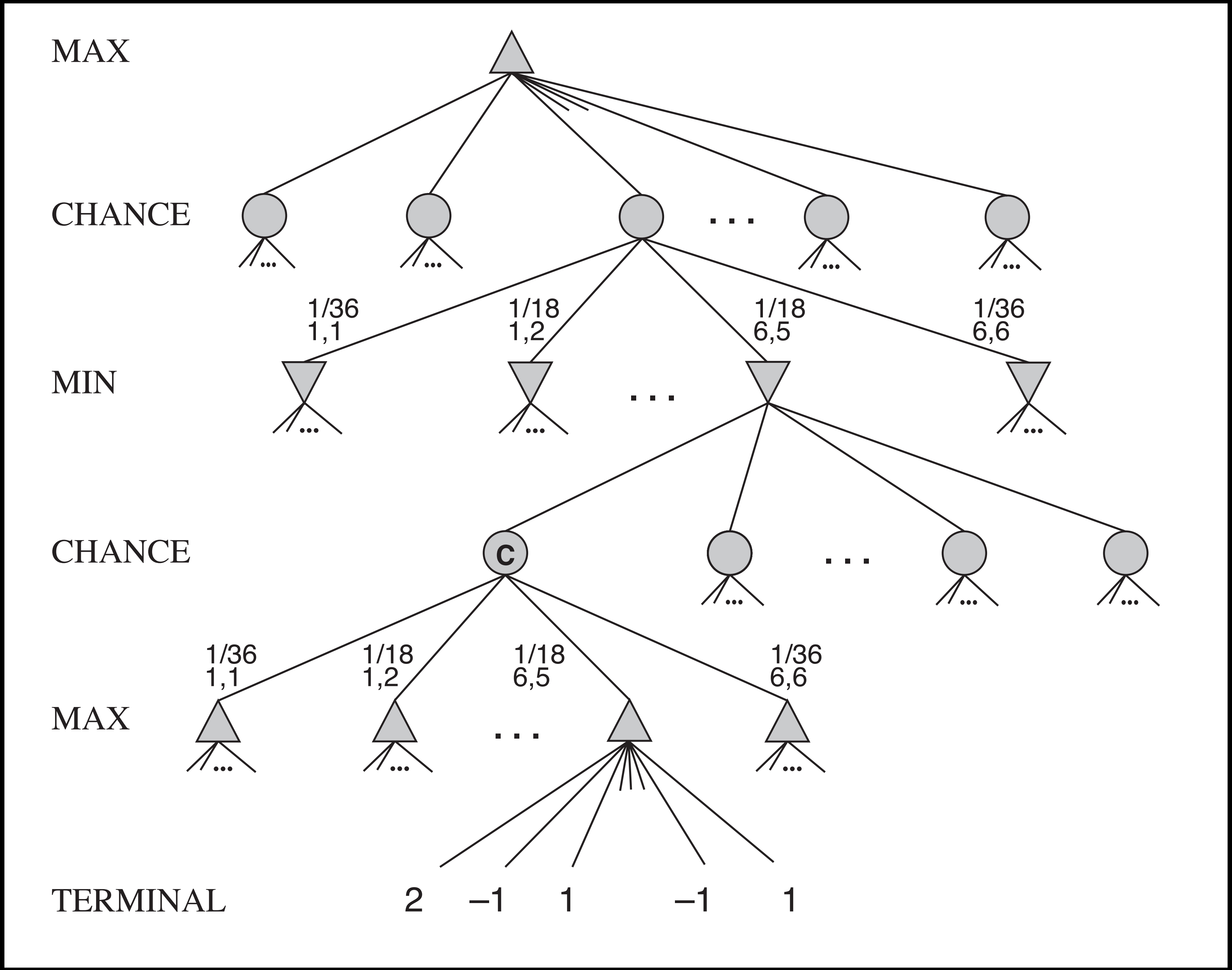
+0.5
H
T
$P(H)=0.75$
$P(T)=0.25$
+1
-1

Expectation

- Weighted average of possibilities
- Sum of the possible outcomes weighted by the likelihood of their occurrence
- What you would expect to win in the long run

Expecti-Minimax

- Same as MINIMAX for MIN and MAX nodes
- Same backing up utilities from terminal nodes
- Take expectation over chance nodes
    - Weighted average of possible outcomes


MAX
CHANCE
MIN
CHANCE
MAX
TERMINAL
1/36
1,1
1/18
1,2
1/18
6,5
1/36
6,6
C
1/36
1,1
1/18
1,2
1/18
6,5
1/36
6,6
2
–1
1
–1
1

Expecti-Minimax

$$\textsc{EMinimax}(s) =$$

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

Partial Observability

- Some of the state of the world is hidden (unobservable)
- There is some uncertainty about the state of the world

Partially-Observable Games

- Some of the state of the game is hidden from the player(s)
- Interesting because:
  - Valuable real-world games (e.g., poker)
  - Partial observability arises all the time in real-world problems

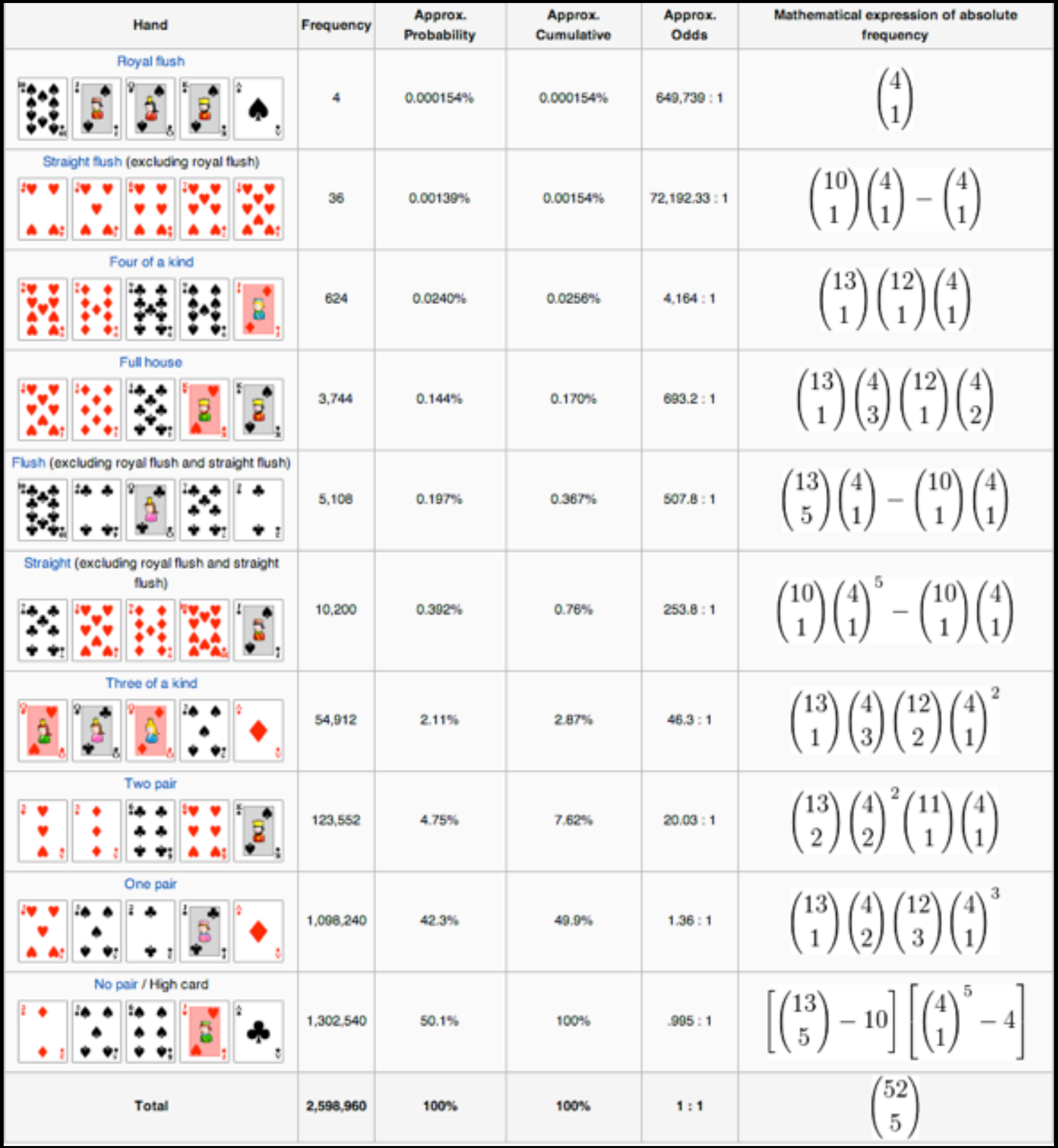
Partially-Observable Games

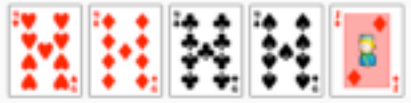
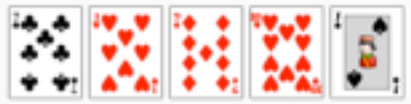
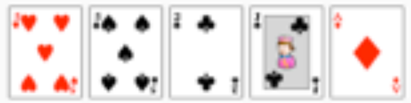
- Deterministic partial observability
  - Opponent has hidden state
  - Battleship, Stratego, Kriegspiel

Partially-Observable Games

- Deterministic partial observability
  - Opponent has hidden state
  - Battleship, Stratego, Kriegspiel
- Stochastic partial observability
  - Missing/hidden information is random
  - Card games: bridge, hearts, poker (most)

Stochastic Partially Observable Games



Hand	Frequency	Approx. Probability	Approx. Cumulative	Approx. Odds	Mathematical expression of absolute frequency
Royal flush	4	0.000154%	0.000154%	649,739 : 1	$\binom{4}{1}$
Straight flush (excluding royal flush)	36	0.00139%	0.00154%	72,192.33 : 1	$\binom{10}{1}\binom{4}{1}-\binom{4}{1}$
Four of a kind 	624	0.0240%	0.0256%	4,164 : 1	$\binom{13}{1}\binom{12}{1}\binom{4}{1}$
Full house	3,744	0.144%	0.170%	693.2 : 1	$\binom{13}{1}\binom{4}{3}\binom{12}{1}\binom{4}{2}$
Flush (excluding royal flush and straight flush)	5,108	0.197%	0.367%	507.8 : 1	$\binom{13}{5}\binom{4}{1}-\binom{10}{1}\binom{4}{1}$
Straight (excluding royal flush and straight flush) 	10,200	0.392%	0.76%	253.8 : 1	$\binom{10}{1}\binom{4}{1}^5-\binom{10}{1}\binom{4}{1}$
Three of a kind	54,912	2.11%	2.87%	46.3 : 1	$\binom{13}{1}\binom{4}{3}\binom{12}{2}\binom{4}{1}^2$
Two pair	123,552	4.75%	7.62%	20.03 : 1	$\binom{13}{2}\binom{4}{2}^2\binom{11}{1}\binom{4}{1}$
One pair 	1,098,240	42.3%	49.9%	1.36 : 1	$\binom{13}{1}\binom{4}{2}\binom{12}{3}\binom{4}{1}^3$
No pair / High card	1,302,540	50.1%	100%	.995 : 1	$\left[\binom{13}{5}-10\right]\left[\binom{4}{1}^5-4\right]$
Total	**2,598,960**	**100%**	**100%**	**1 : 1**	$\binom{52}{5}$

Weighted Minimax

- For each possible deal s:
  - Assume s is the actual situation
  - Compute Minimax or H-Minimax value of s
  - Weight value by probability of s
- Take move that yields highest expected value over all the possible deals

Weighted Minimax

$$\operatorname*{argmax}_a \sum_s P(s)\textsc{Minimax}(\textsc{Result}(s,a))$$

Weighted Minimax

$$\operatorname*{argmax}_a \sum_s P(s)\,\textsc{Minimax}(\textsc{Result}(s,a))$$

$$\binom{26}{13} = 10,400,600$$

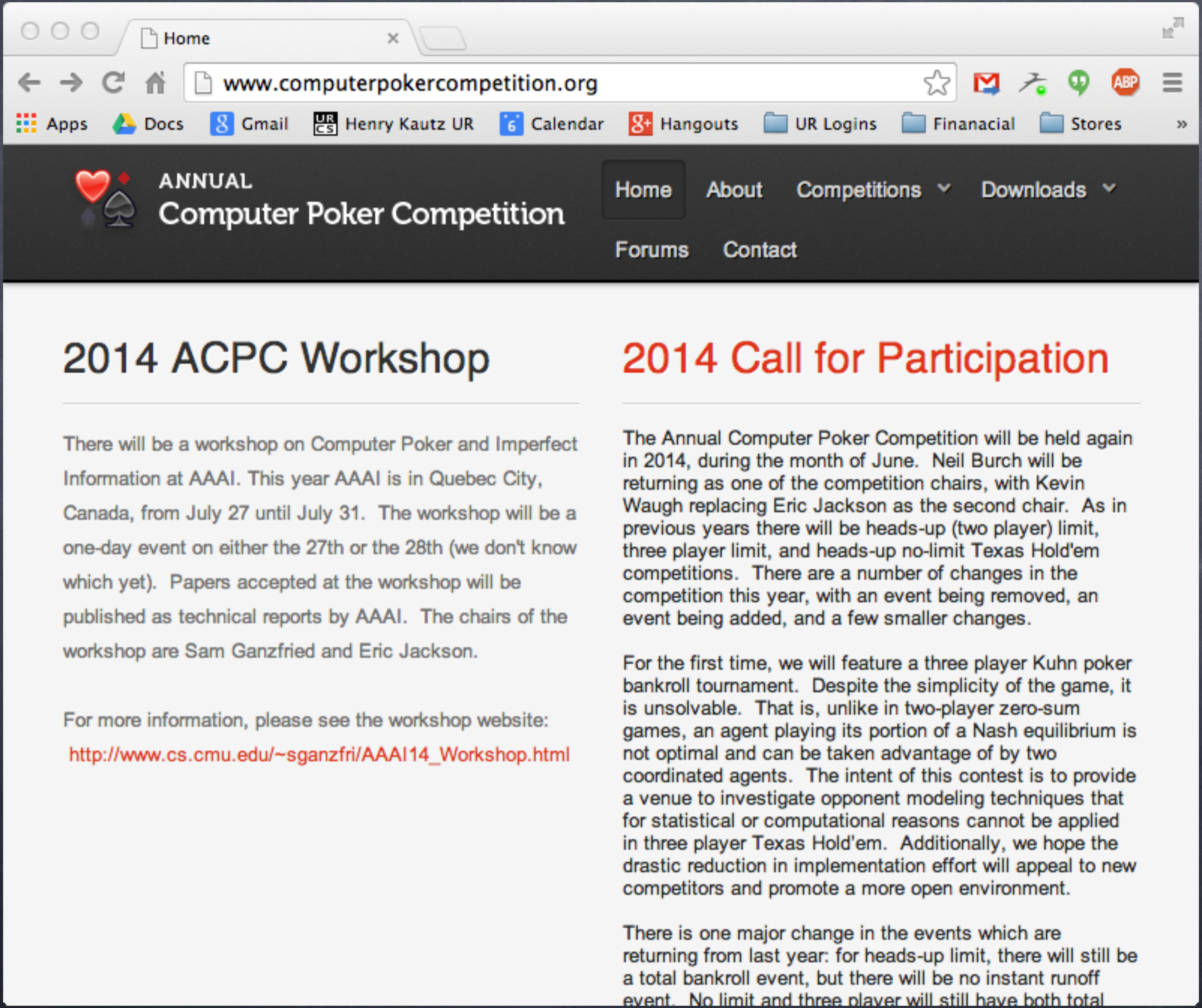
$$\binom{47}{25} = 1.48338977 \times 10^{13}$$

Monte Carlo Methods

- Use a “representative” sample to approximate a large, complex distribution

Monte Carlo Minimax

$$\operatorname*{argmax}_a \frac{1}{N} \sum_{i=1}^{N} \textsc{Minimax}(\textsc{Result}(s_i, a))$$



2014 ACPC Workshop

There will be a workshop on Computer Poker and Imperfect Information at AAAI. This year AAAI is in Quebec City, Canada, from July 27 until July 31. The workshop will be a one-day event on either the 27th or the 28th (we don't know which yet). Papers accepted at the workshop will be published as technical reports by AAAI. The chairs of the workshop are Sam Ganzfried and Eric Jackson.

For more information, please see the workshop website: http://www.cs.cmu.edu/~sganzfri/AAAI14_Workshop.html

2014 Call for Participation

The Annual Computer Poker Competition will be held again in 2014, during the month of June. Neil Burch will be returning as one of the competition chairs, with Kevin Waugh replacing Eric Jackson as the second chair. As in previous years there will be heads-up (two player) limit, three player limit, and heads-up no-limit Texas Hold'em competitions. There are a number of changes in the competition this year, with an event being removed, an event being added, and a few smaller changes.

For the first time, we will feature a three player Kuhn poker bankroll tournament. Despite the simplicity of the game, it is unsolvable. That is, unlike in two-player zero-sum games, an agent playing its portion of a Nash equilibrium is not optimal and can be taken advantage of by two coordinated agents. The intent of this contest is to provide a venue to investigate opponent modeling techniques that for statistical or computational reasons cannot be applied in three player Texas Hold'em. Additionally, we hope the drastic reduction in implementation effort will appeal to new competitors and promote a more open environment.

There is one major change in the events which are returning from last year: for heads-up limit, there will still be a total bankroll event, but there will be no instant runoff event. No limit and three player will still have both total

Summary

- Non-deterministic games
  - Expecti-MINIMAX: Compute expected MINIMAX value over chance nodes
- Partially observable games
  - Weighted MINIMAX: Compute expected value over possible hidden states
  - When tree becomes too large, sample branches rather than explore exhaustively

For Next Time:
AIMA 6.0-6.4