Artificial Intelligence Programming

Adversarial Search

Cindi Thompson

Department of Computer Science

University of San Francisco

AC-3 Pseudocode

```
v = the variables in our problem.
d[v] is the list of values in the domain of each v

for vertex in v :
   neighbors = all vertices in v that share a constraint with vertex
   for n in neighbors :
      for value in d[vertex] :
         if there is no value in d[n] consistent with value:
            remove value from d[vertex]
            if d[vertex] is empty, return failure
repeat until d[vertex] does not change for any v
```

Overview

- Example games (board splitting, chess, Othello)
- Min/Max trees
- Alpha-Beta Pruning
- Evaluation Functions
- Stopping the Search
- Playing with chance

Games as Search

"Unpredictable" opponent $\rightarrow$ specify a move for every possible opponent reply

Time limits $\rightarrow$ unlikely to find goal, must approximate

Let's start with deterministic, 2-player games

Two player games

- Board-Splitting Game
  - Two players, V & H
  - V splits the board vertically, selects one half
  - H splits the board horizontally, selects one half
  - V tries to maximize the final value, H tries to minimize the final value

14	5	11	4
12	13	9	7
15	13	10	8
16	1	6	2

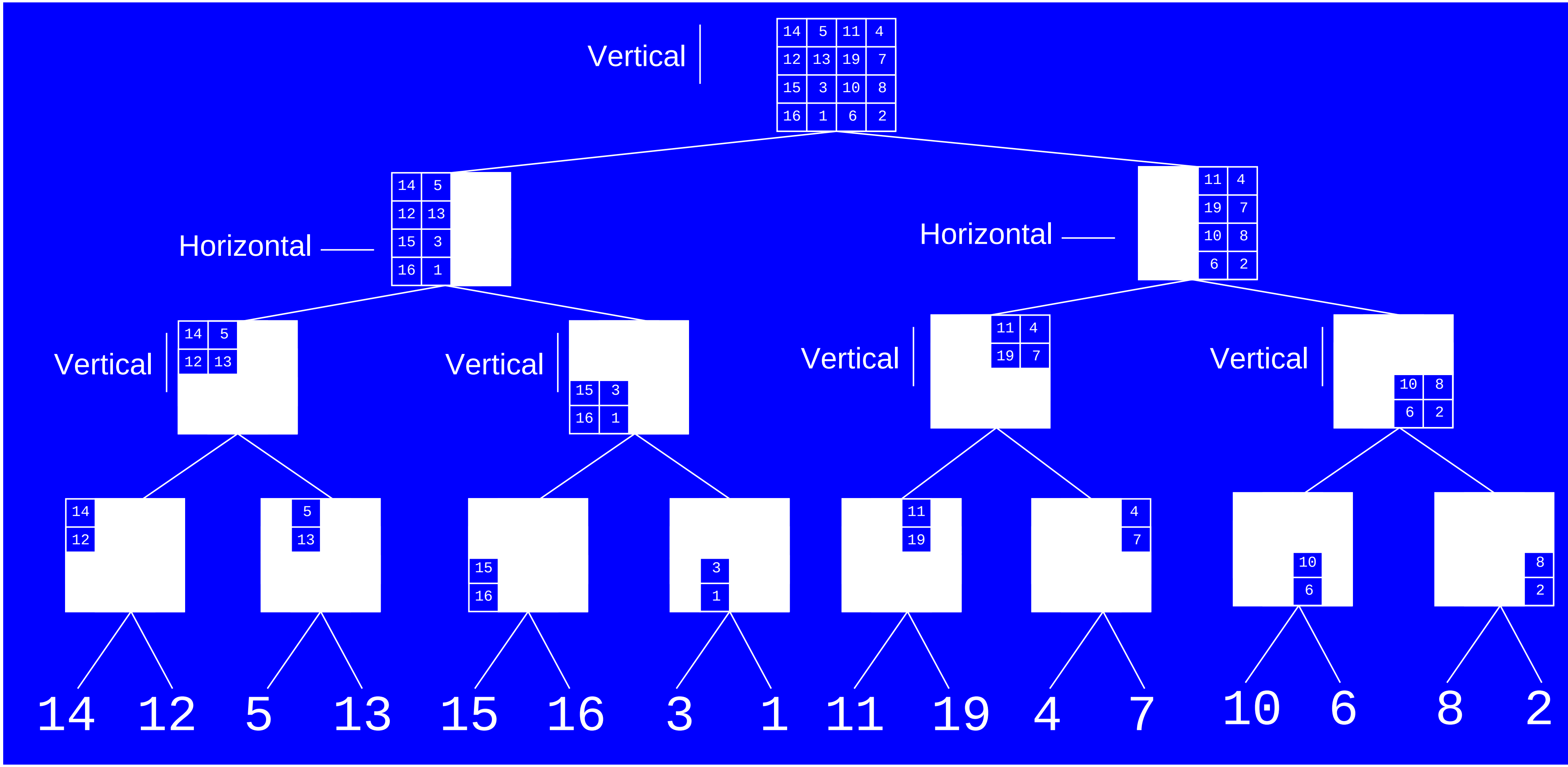
Two player games

- Board-Splitting Game
  - We assume that both players are rational (make the best possible move)
  - How can we determine who will win the game?
  - And, how can we determine the best move at each state?

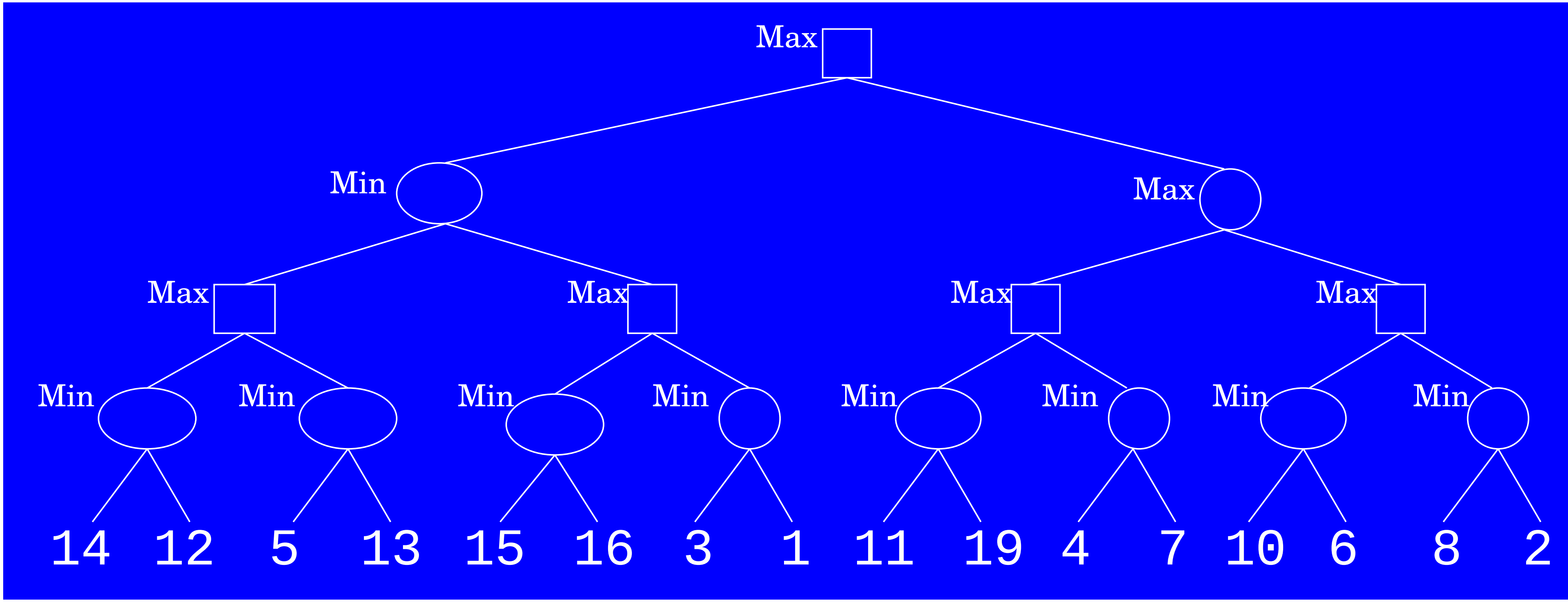
Two player games

- Board-Splitting Game
  - We assume that both players are rational (make the best possible move)
  - How can we determine who will win the game?
    - Examine all possible games!

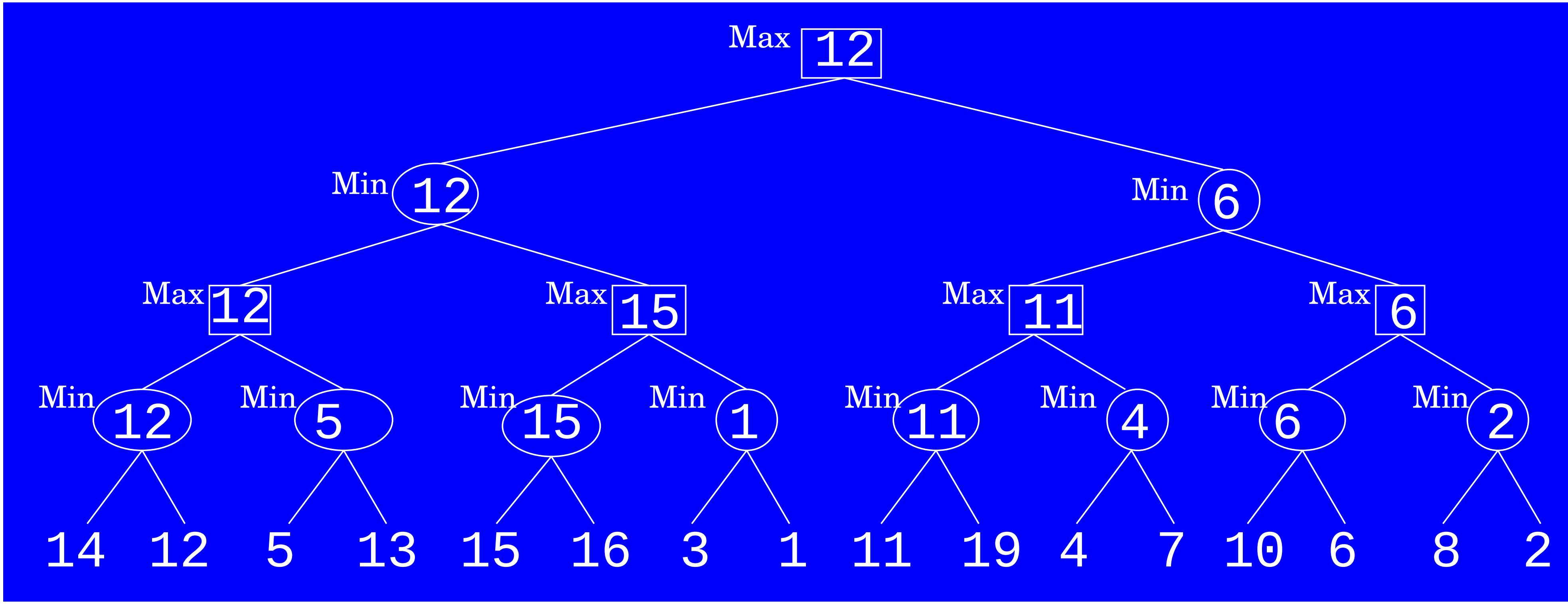
Two player games



Two player games



Two player games



Two player games

- Game playing agent can do this to figure out which move to make
  - Examine all possible moves
  - Examine all possible responses to each move
  - ... all the way to the last move
  - Caclulate the value of each move (assuming opponent plays perfectly)

Two-Player Games

- Initial state
- Successor Function
  - Just like other Searches
- Terminal Test
  - When is the game over?
- Utility Function
  - Only applies to terminal states
  - Chess: +1, 0, -1
  - Backgammon: 192 . . . -192

Minimax Algorithm

```
def Max-val(node):
  if terminal(node):
     return utility(node)
  maxVal = MIN_VALUE
  children = successors(node)
  for child in children:
     maxVal = max(maxVal, Min-val(child))
  return maxVal

def: Min-val(node)
  if terminal(node):
     return utility(node)
  minVal = MAX_VALUE
  children = successors(node)
  for child in children:
     minVal = min(minVal, Max-val(child))
  return minVal
```

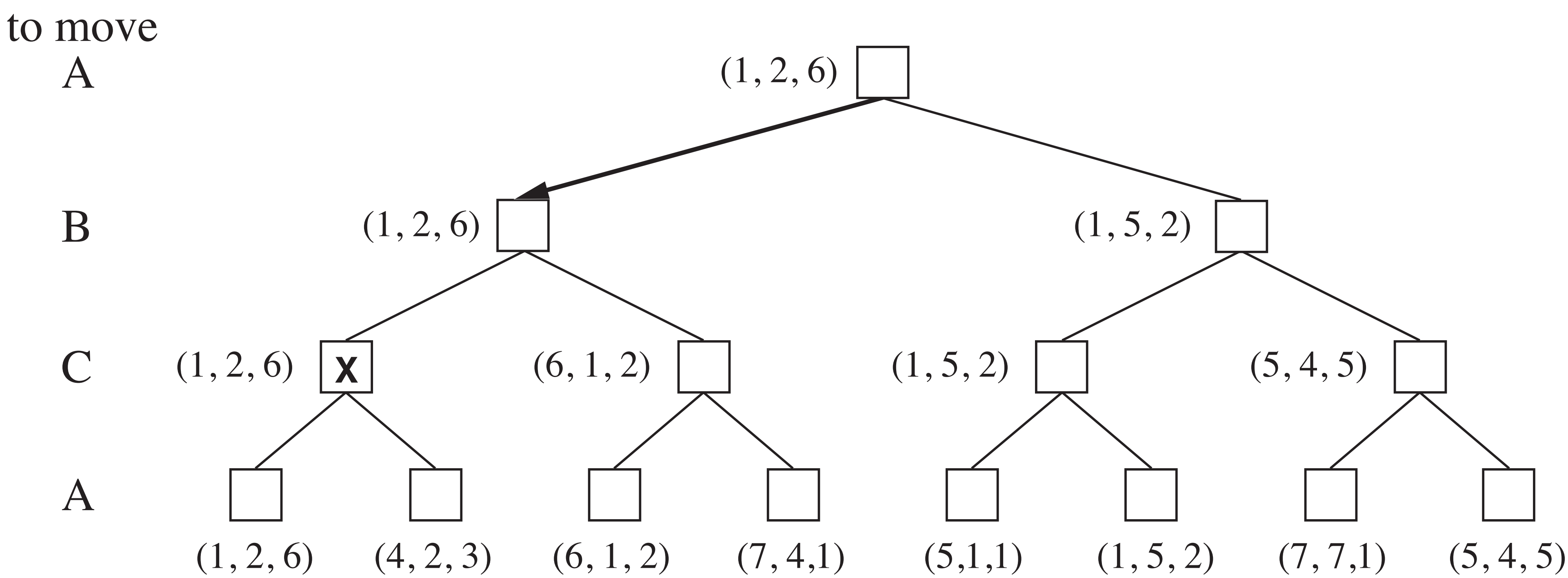
> 2 Player Games

- What if there are > 2 players?
- We can use the same search tree:
  - Alternate between several players
  - Need a different evaluation function

> 2 Player Games

- Functions return a vector of utilities
  - One value for each player
  - Each player tries to maximize their utility
  - May or may not be zero-sum

> 2 Player Games



Non zero-sum games

- Even 2-player games don't need to be zero-sum
  - Utility function returns a vector
  - Each player tries to maximize their utility
- If there is a state with maximal outcome for both players, rational players will cooperate to find it
- Minimax is rational, will find such a state

Minimax Algorithm

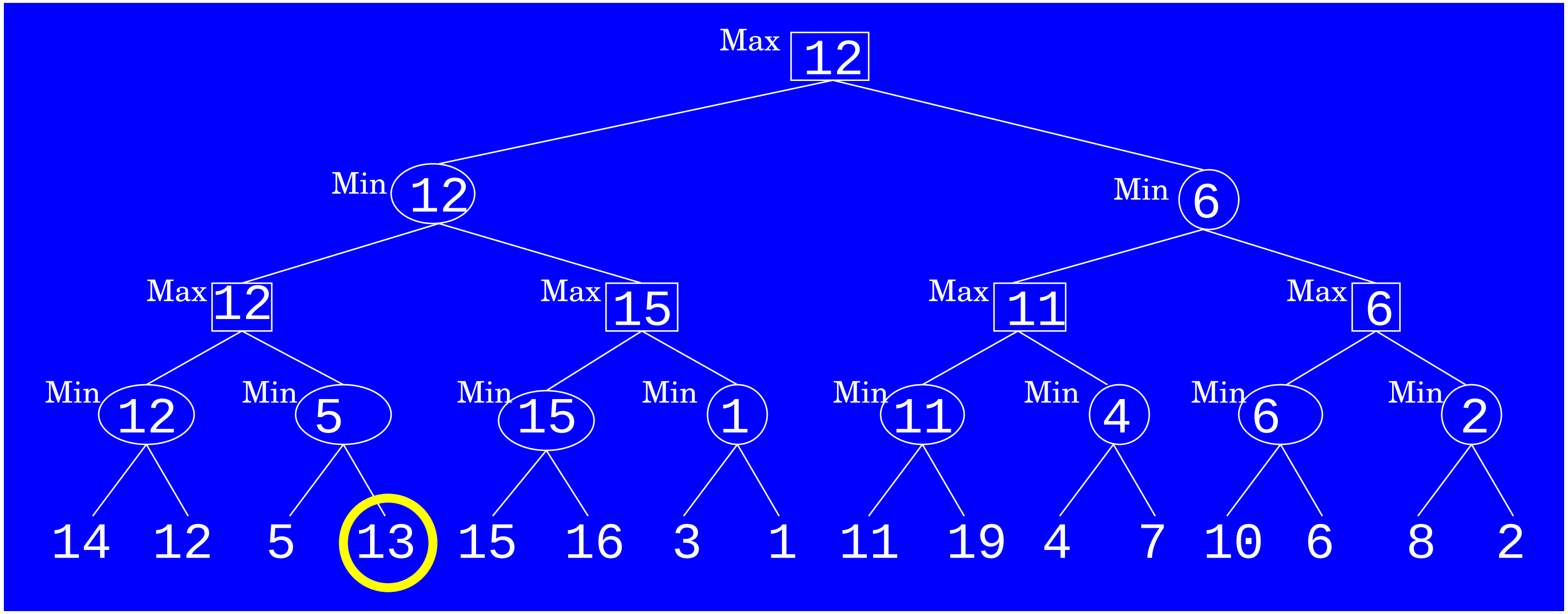
- Complete?
- Optimal?
- Branching factor of b, game length of d moves, what are the time and space requirements for Minimax?

Minimax Algorithm

- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent
- Branching factor of b, game length of d moves, what are the time and space requirements for Minimax?
  - Time: $O(b^d)$
  - Space: $O(d)$
- Not managable for any real games – chess has an average b of 35, can't search the entire tree
- Need to make this more managable

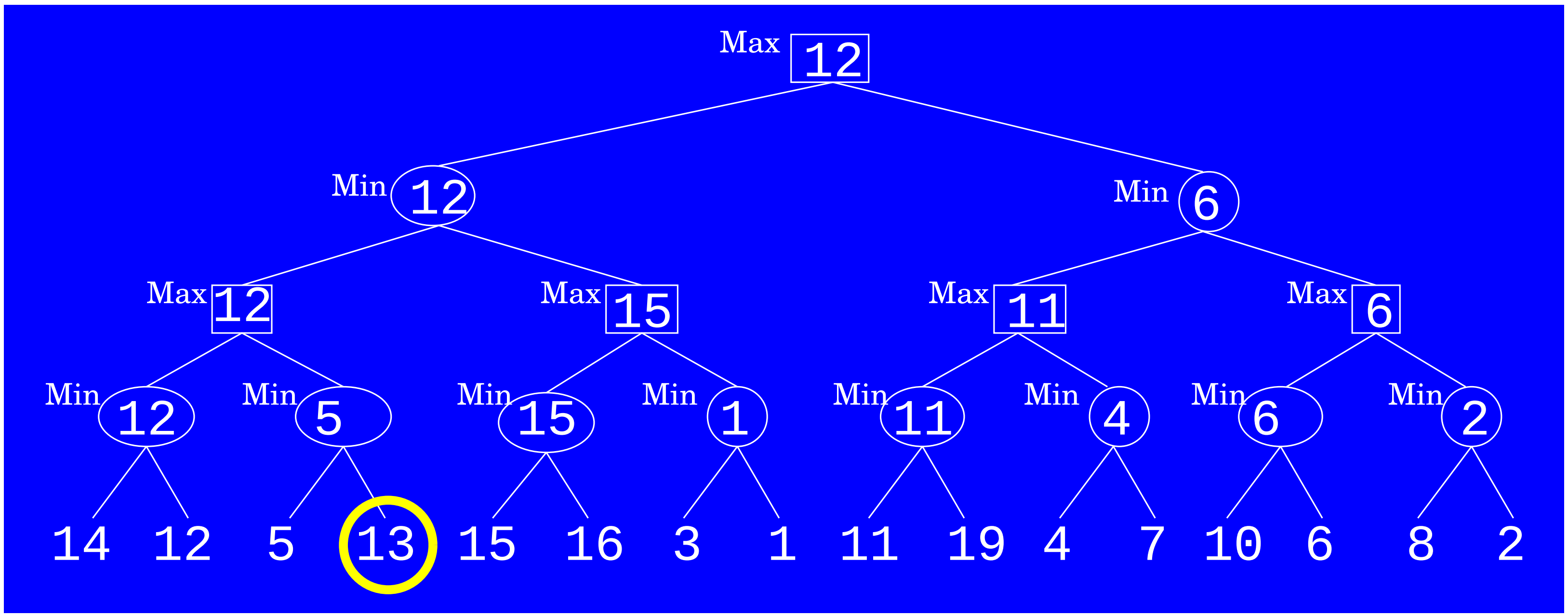
Alpha-Beta Pruning

- Does it matter what value is in the yellow circle?



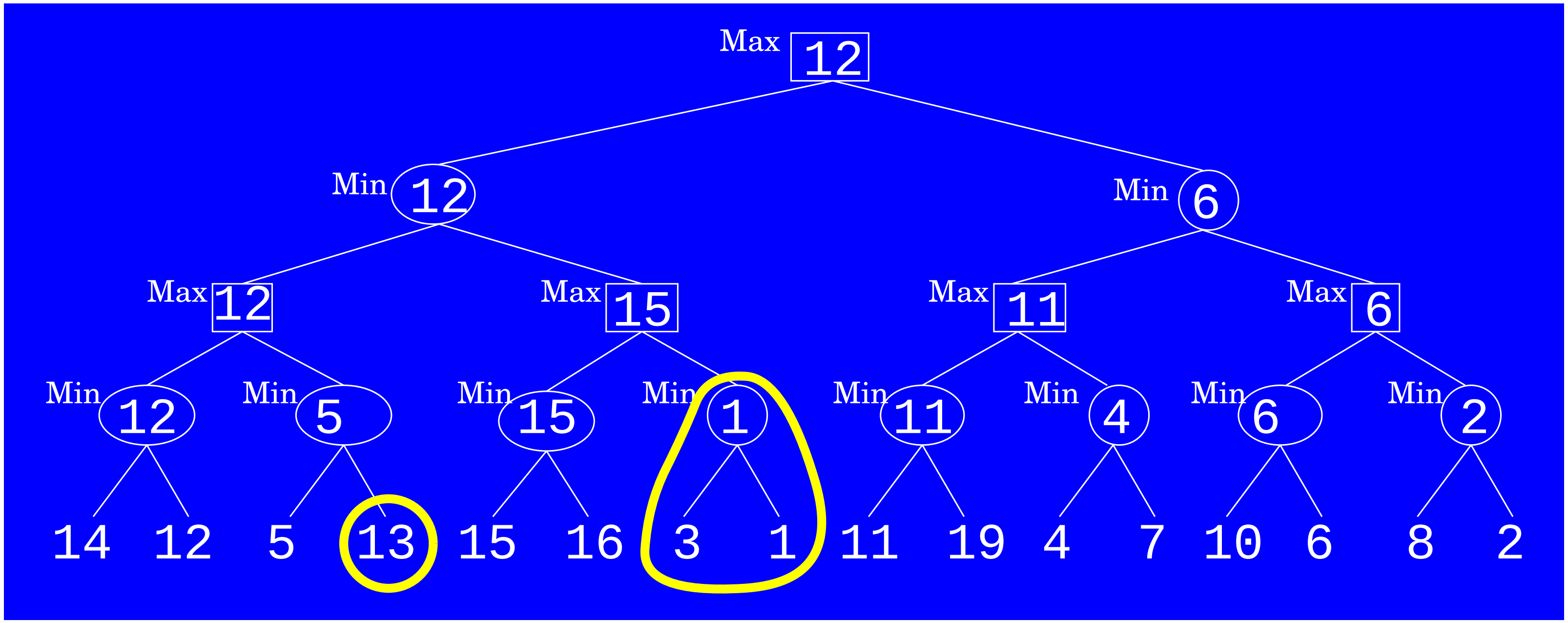
Alpha-Beta Pruning

- If the yellow leaf has a value > 5, parent won't pick it
- If the yellow leaf has a value < 12, grandparent won't pick it
- To affect the root, value must be < 5 **and** > 12



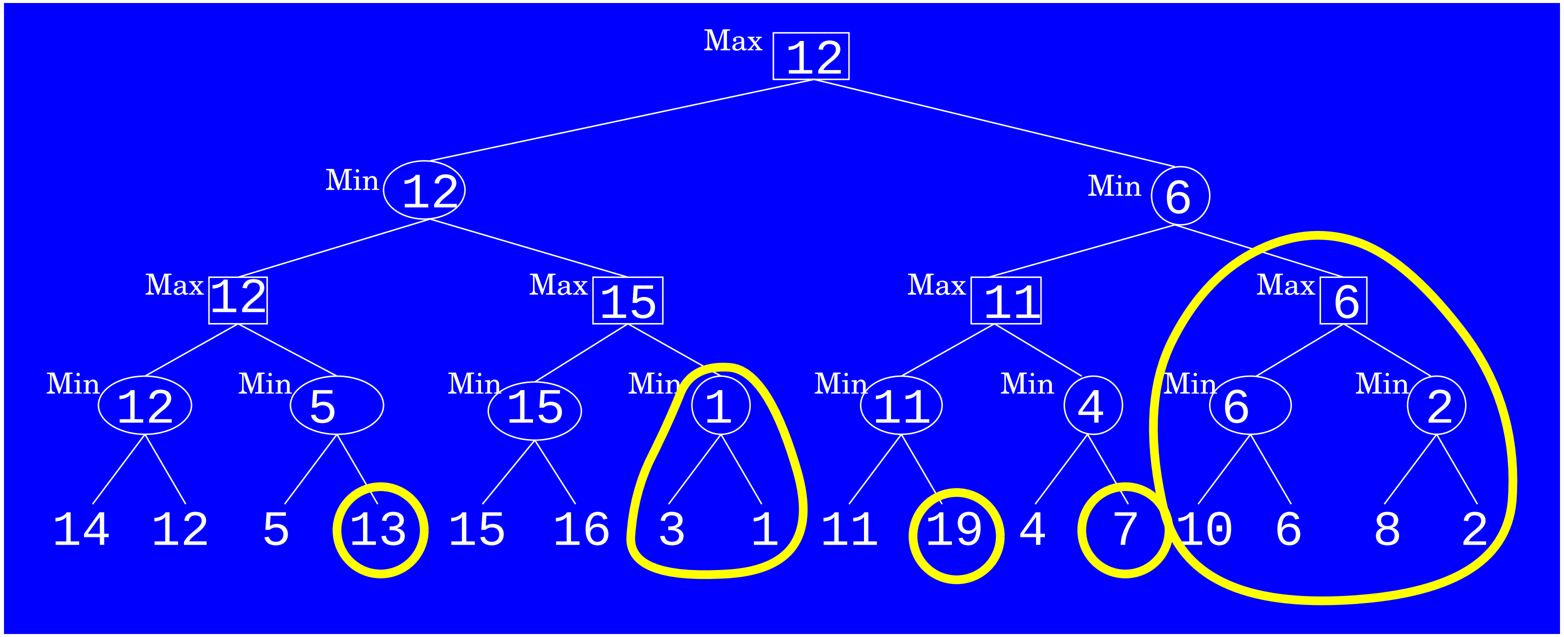
Alpha-Beta Pruning

- Value of nodes in neither yellow circle matter. Are there more?

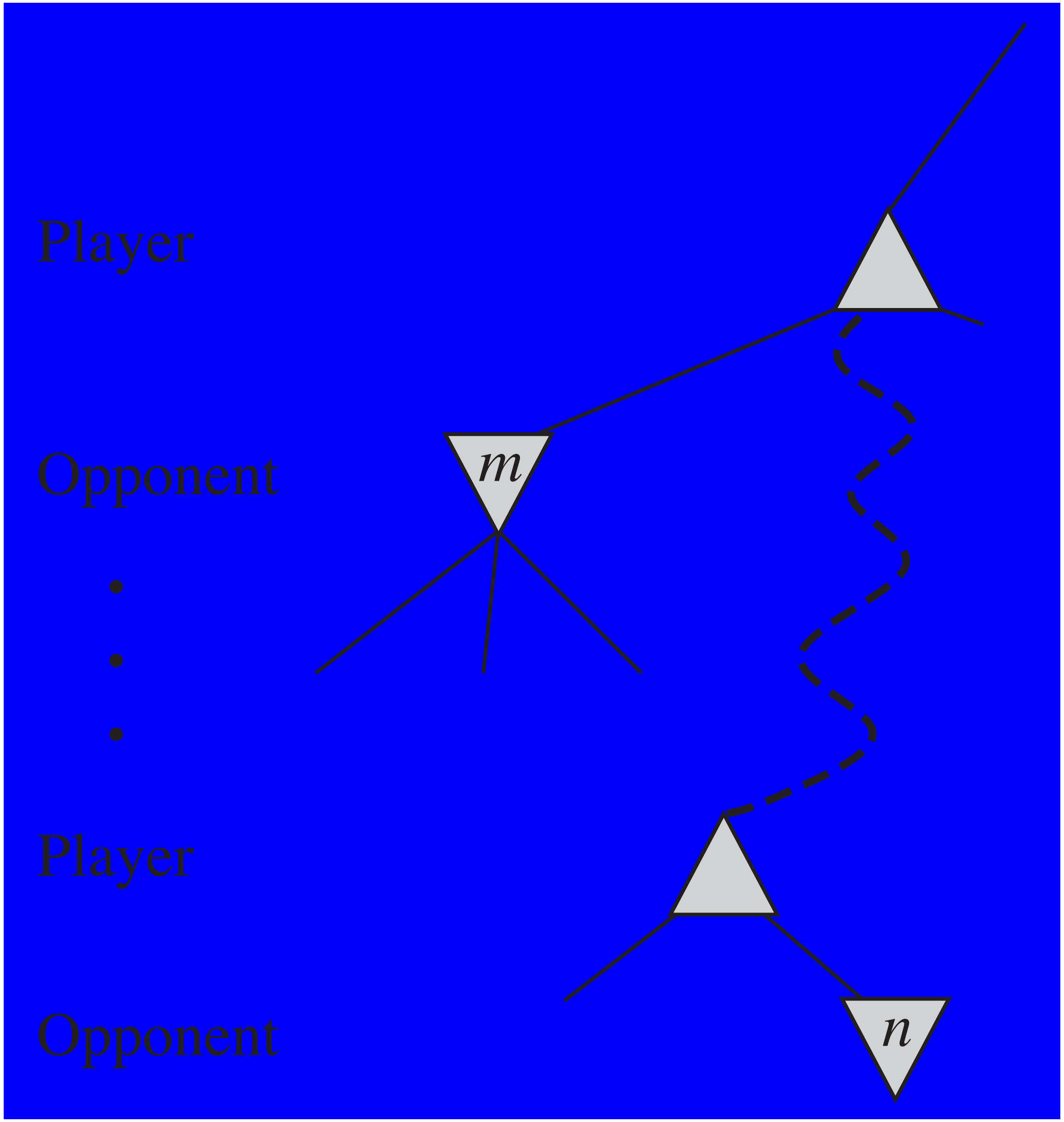


Alpha-Beta Pruning

- Value of nodes in none of the yellow circles matter.



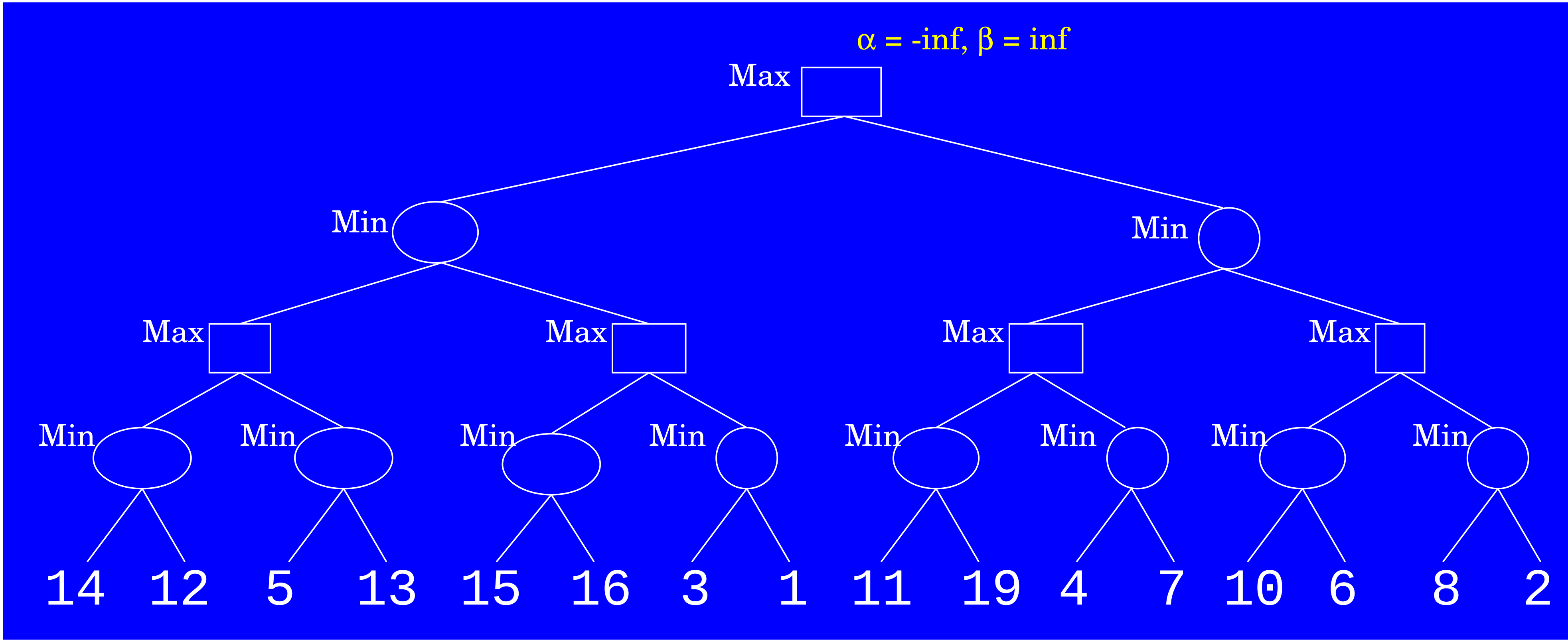
Alpha-Beta Pruning



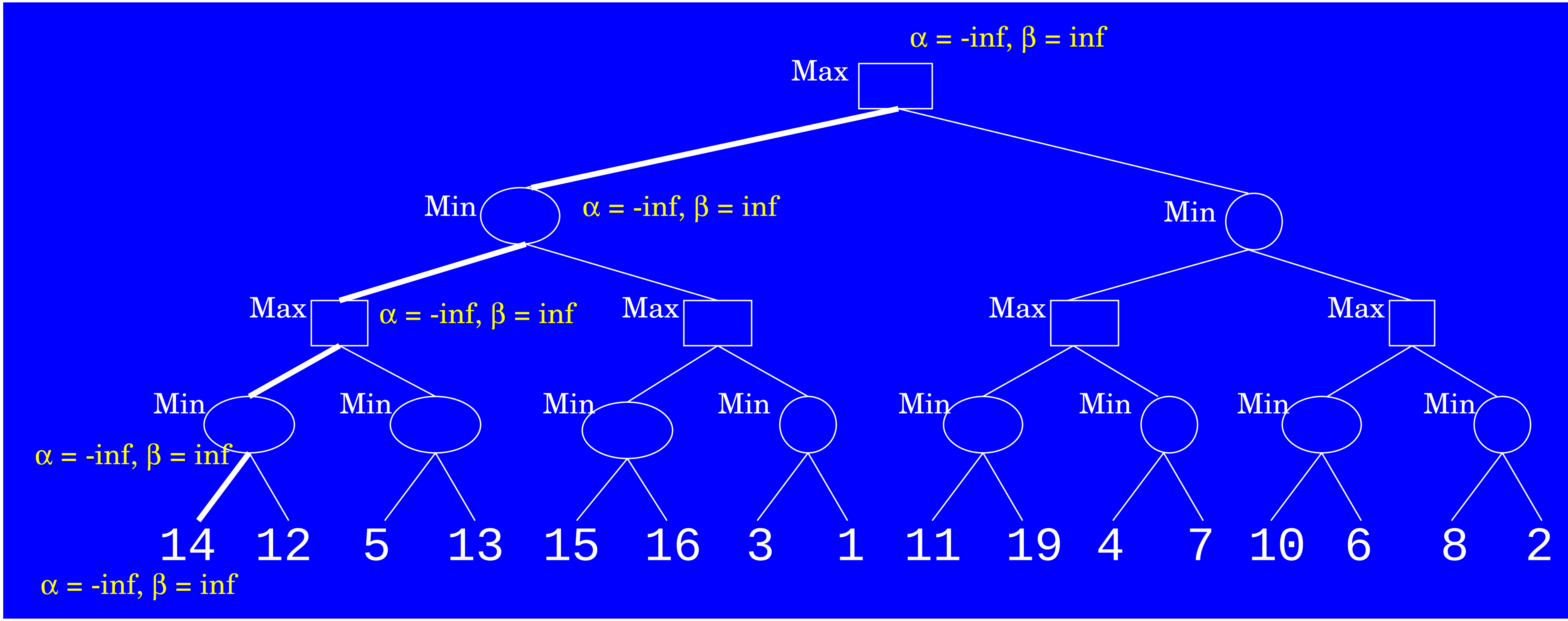
Alpha-Beta Pruning

- Maintain two bounds, lower bound α, and an upper bound β
  - Bounds represent the values the node must have to possibly affect the root
- As you search the tree, update the bounds
  - Max nodes increase α, min nodes decrease β
- If the bounds ever cross, this branch cannot affect the root, we can prune it.

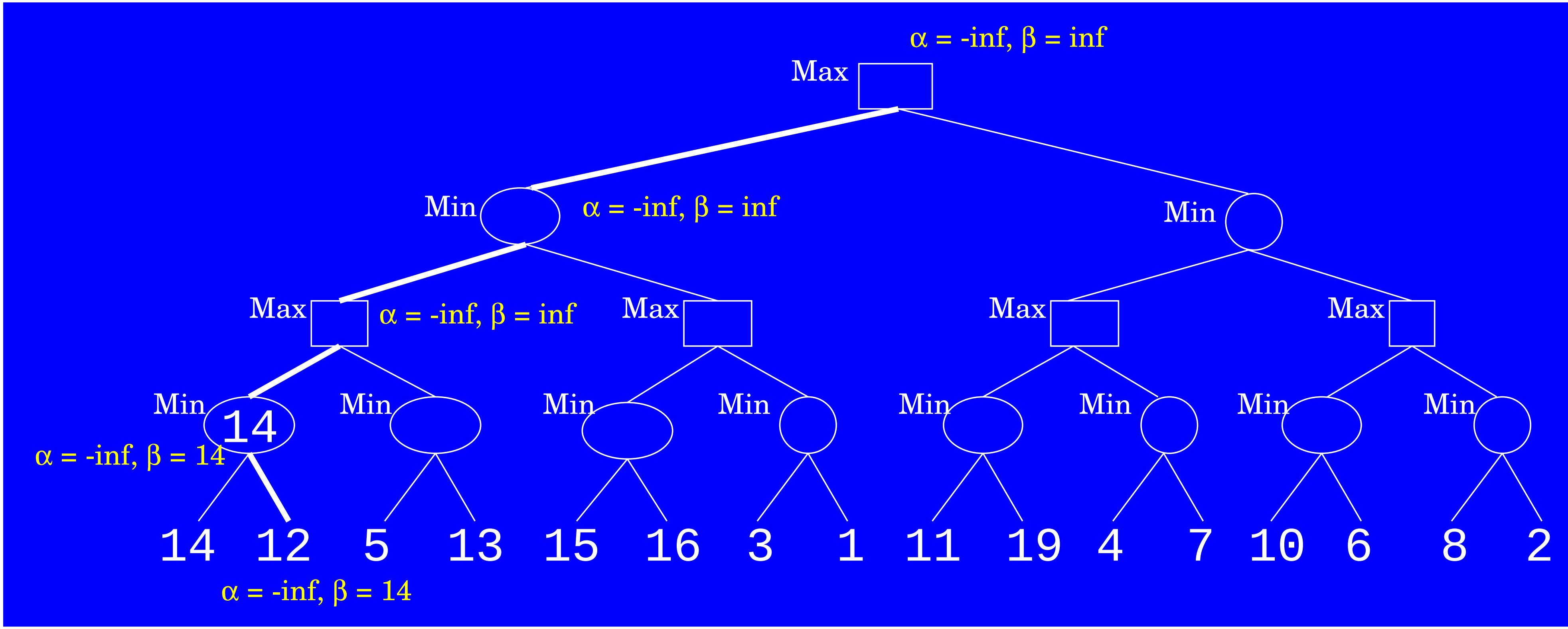
Alpha-Beta Pruning



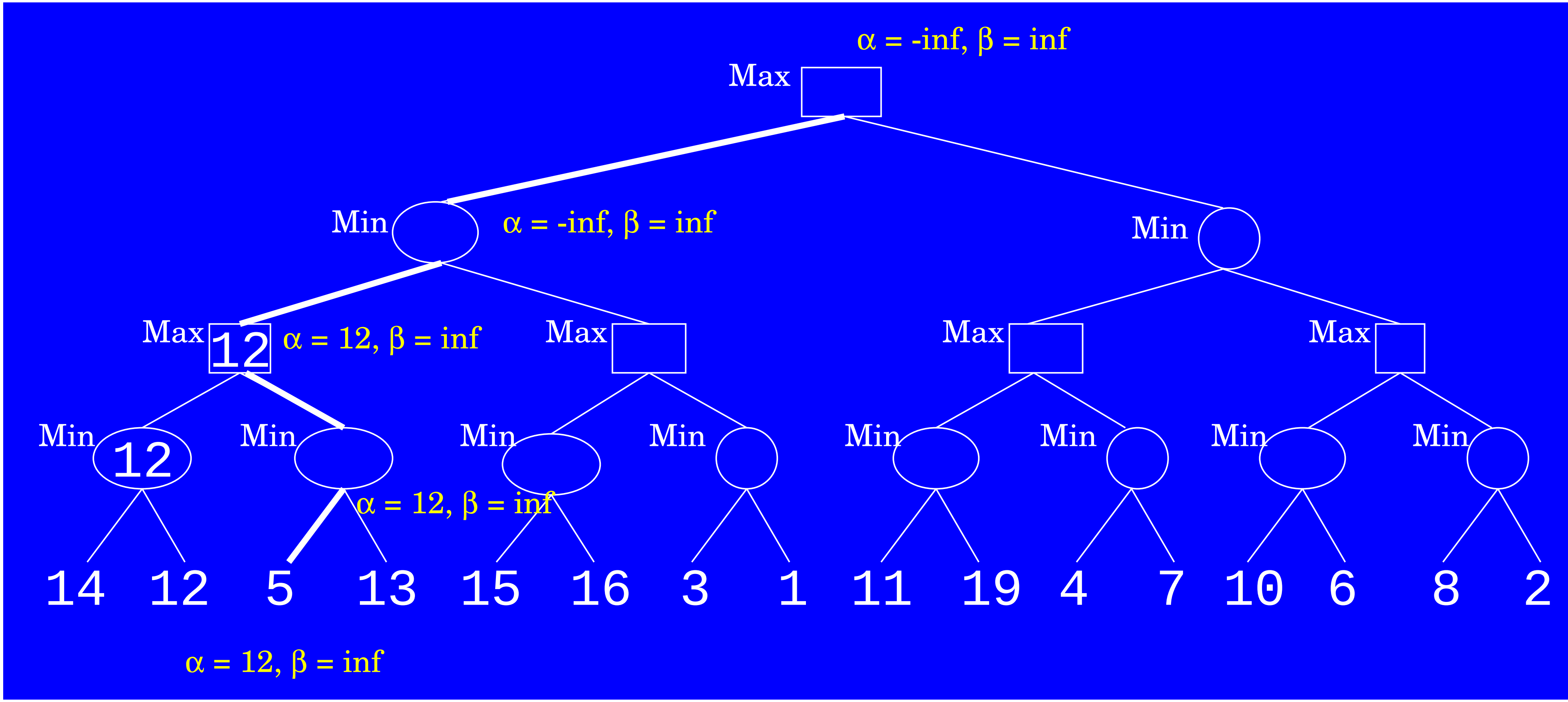
Alpha-Beta Pruning



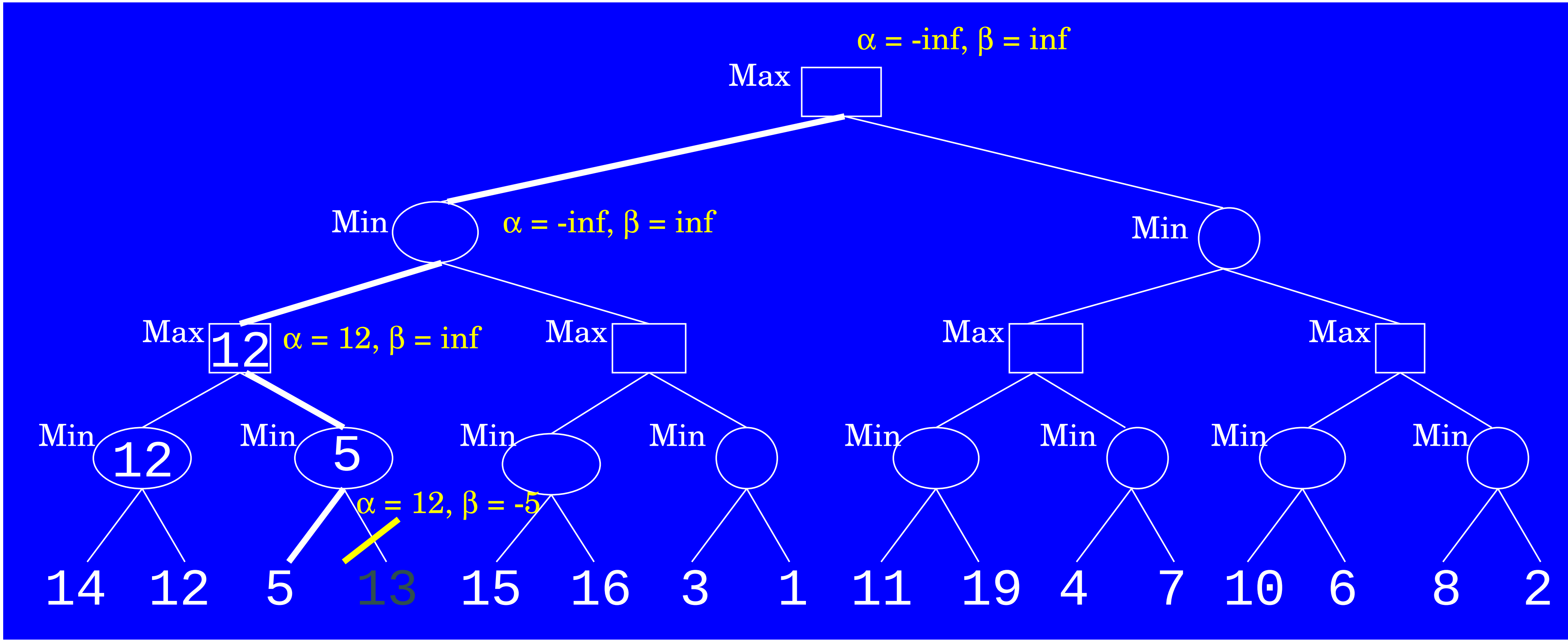
Alpha-Beta Pruning



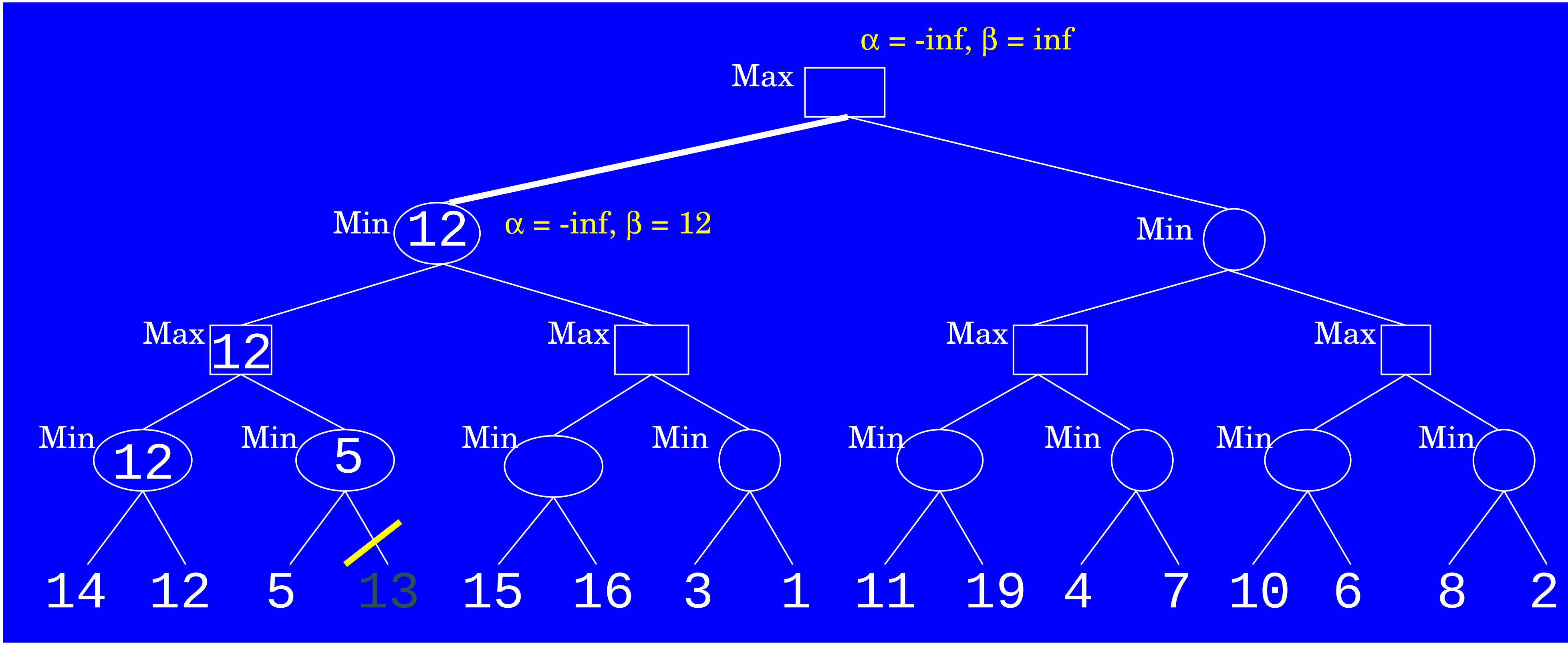
Alpha-Beta Pruning



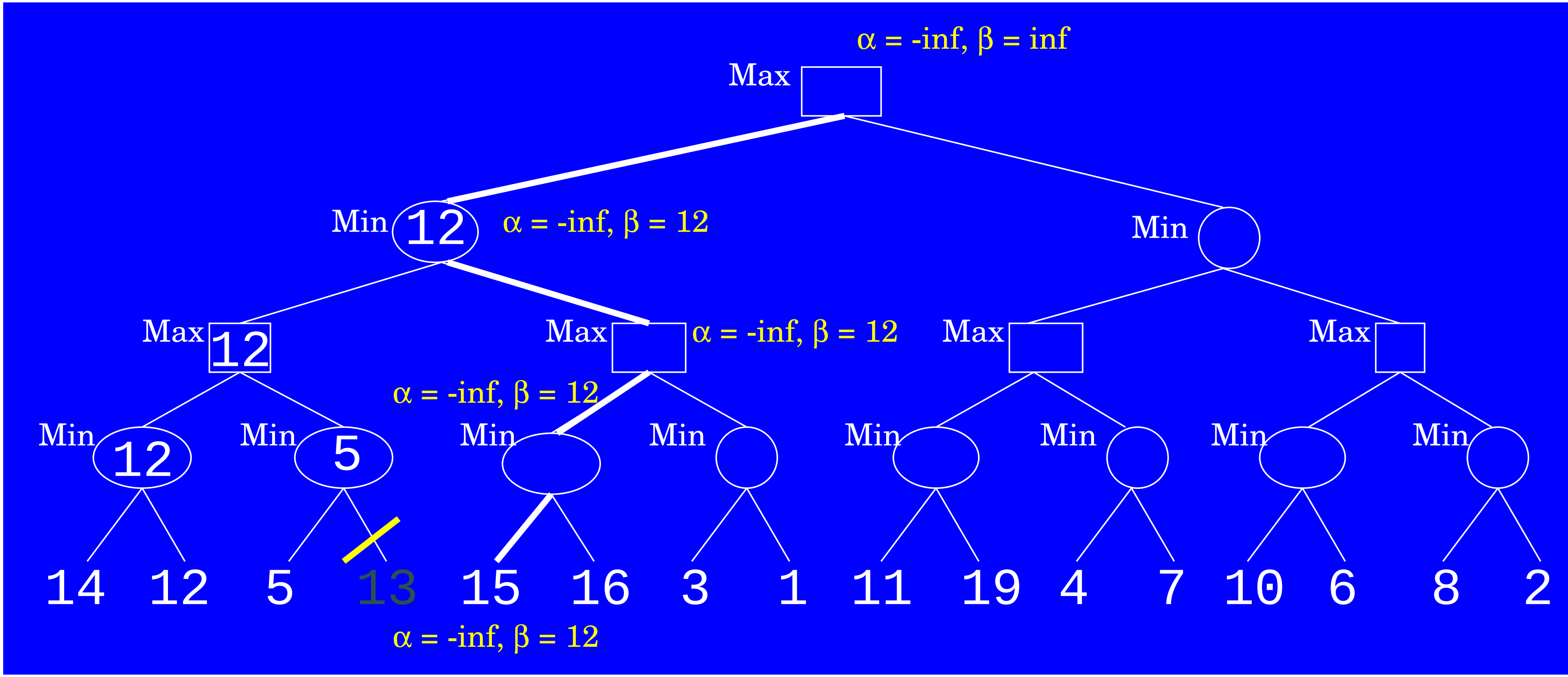
Alpha-Beta Pruning



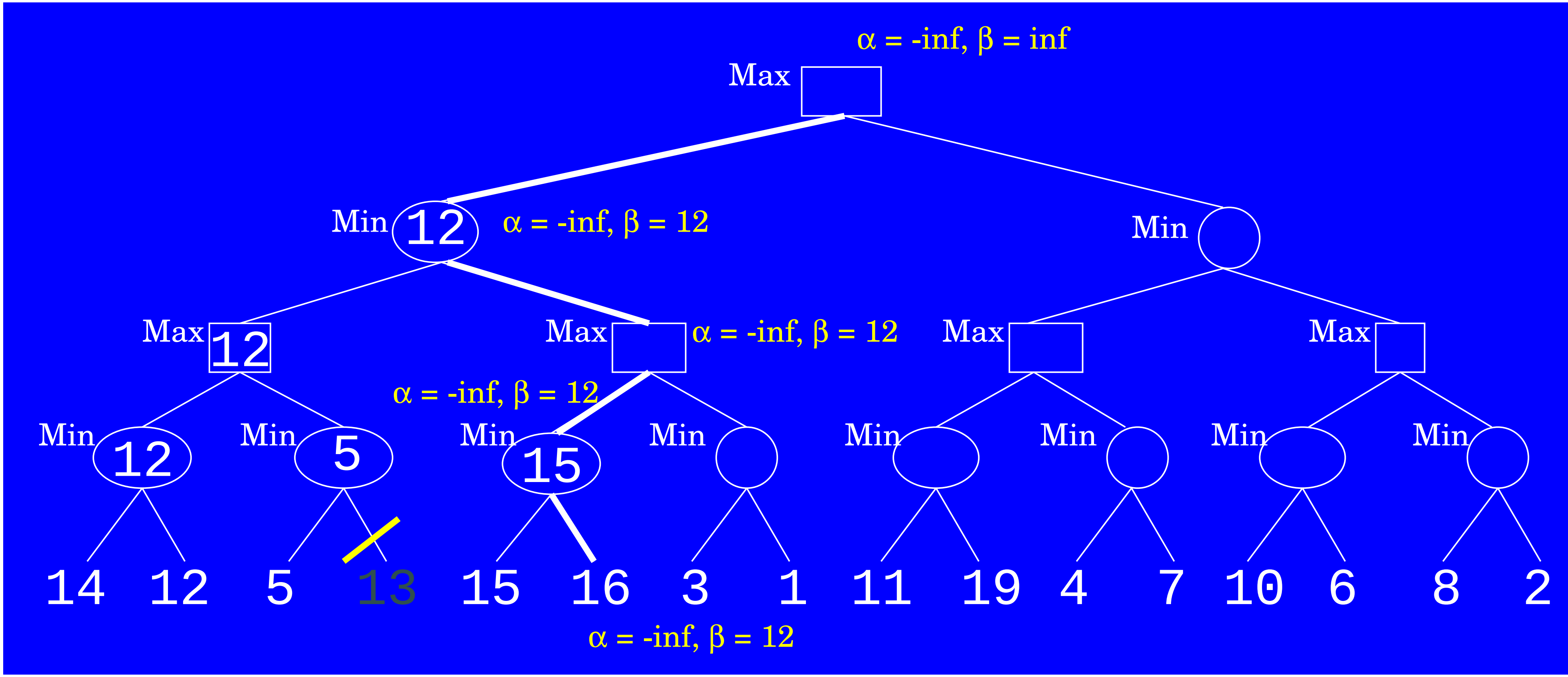
Alpha-Beta Pruning



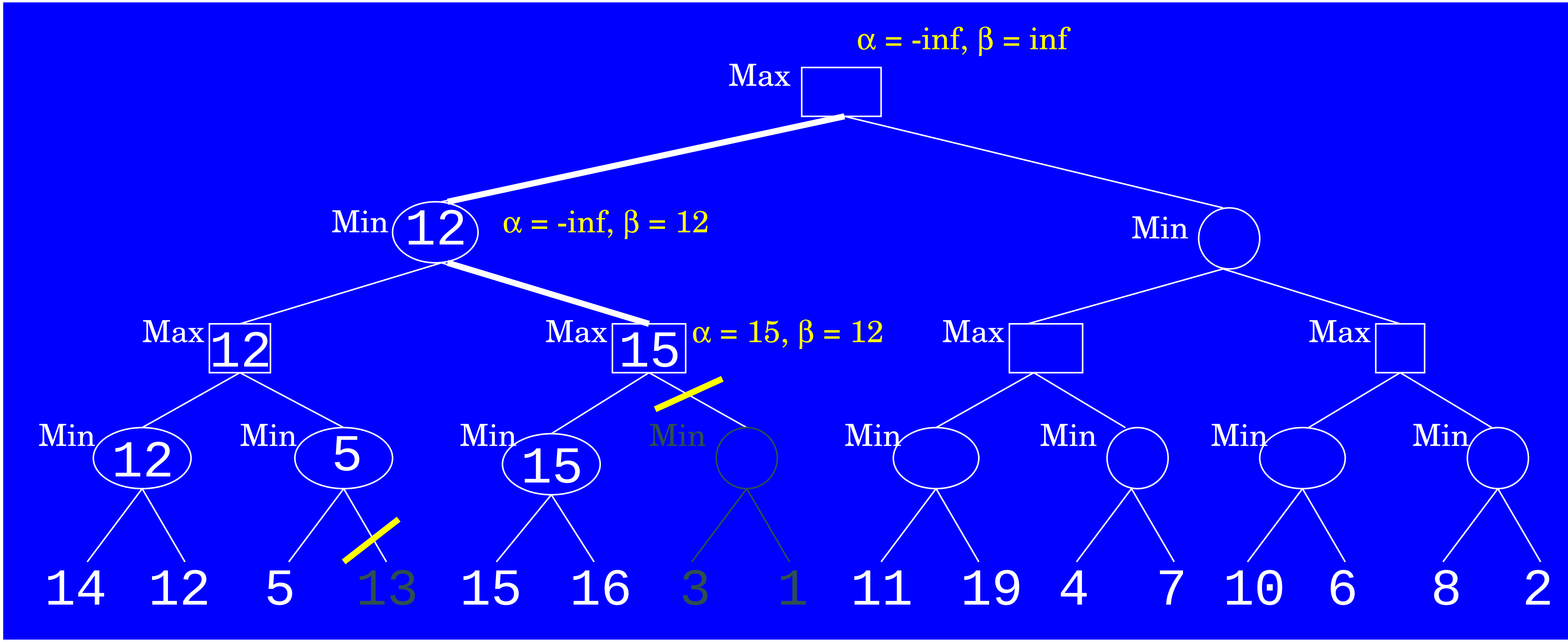
Alpha-Beta Pruning



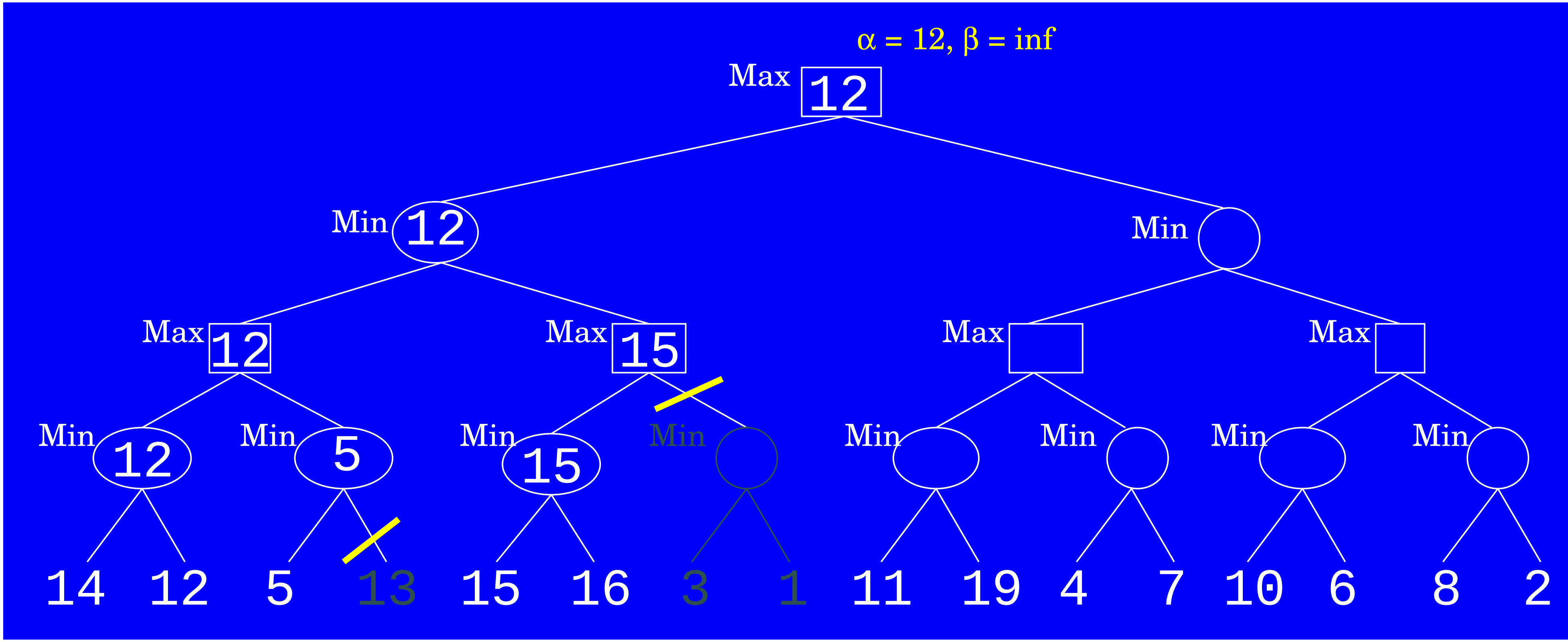
Alpha-Beta Pruning



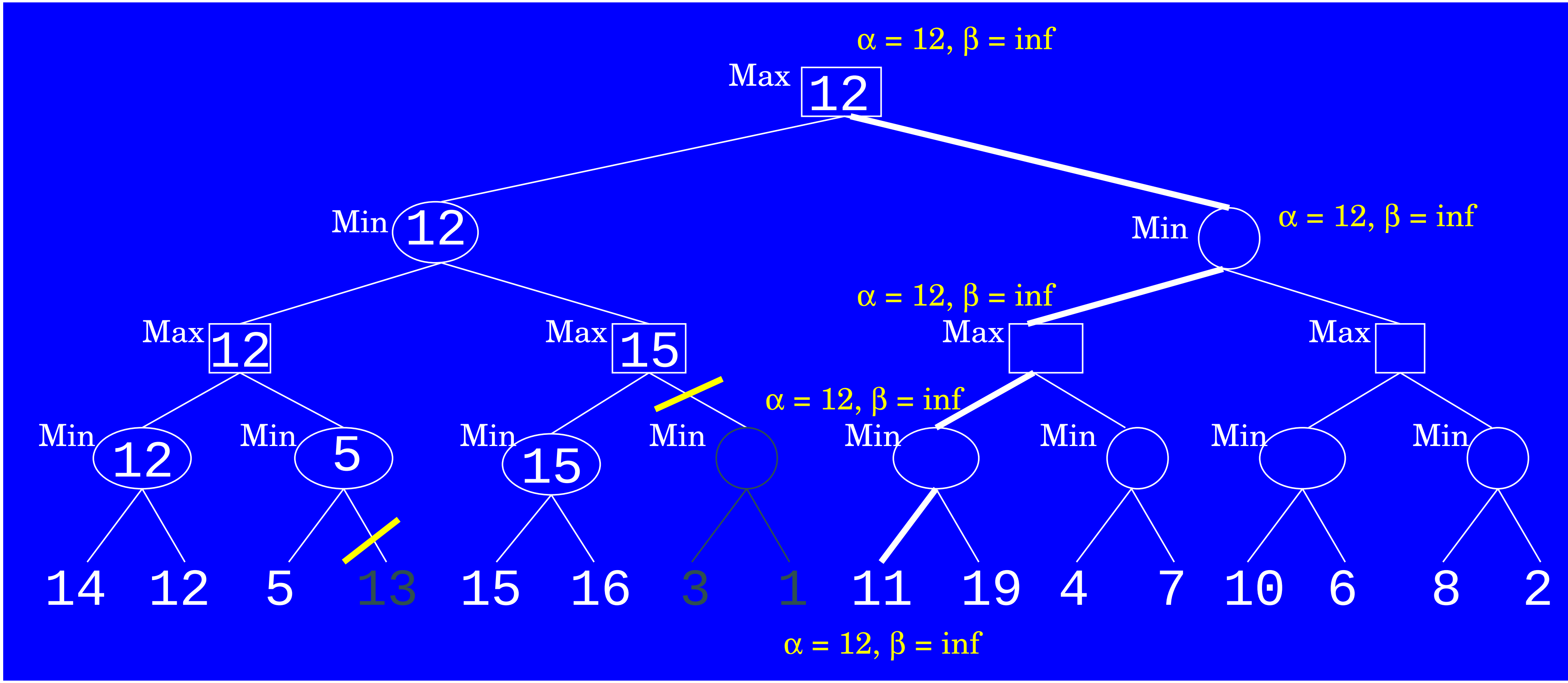
Alpha-Beta Pruning



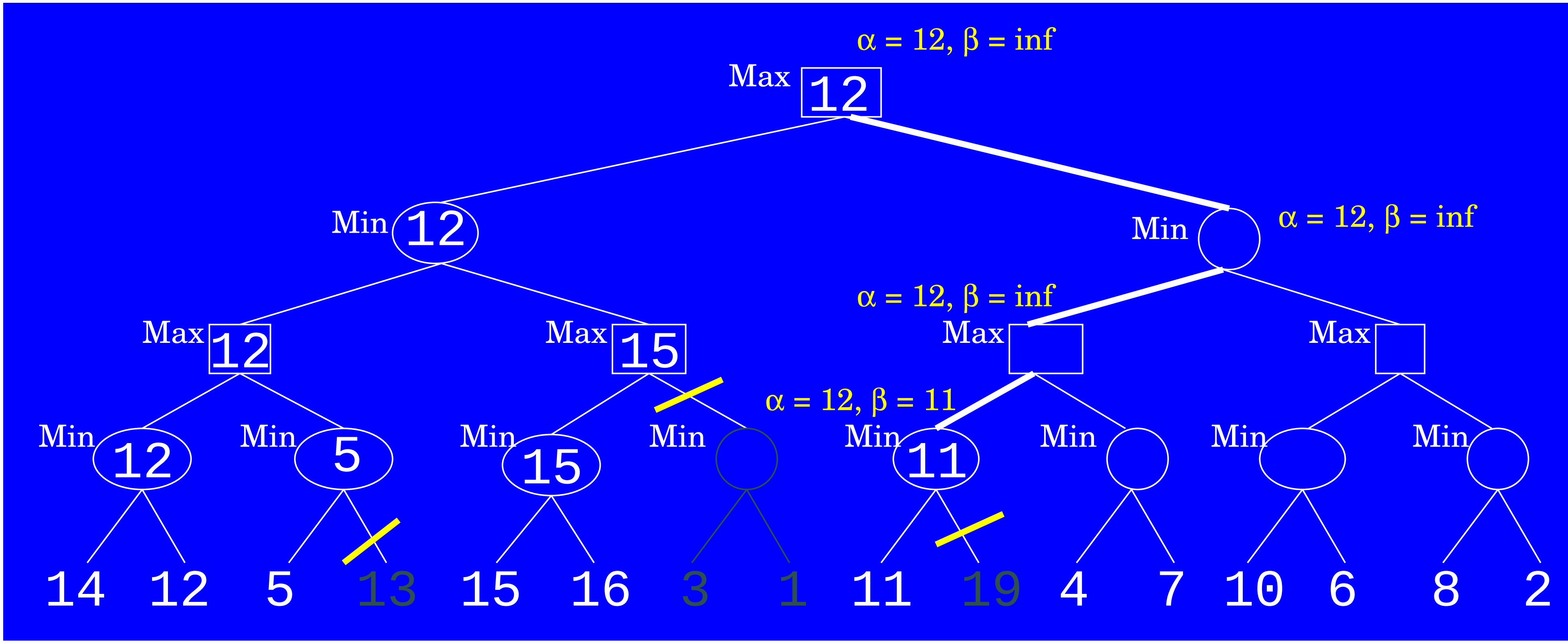
Alpha-Beta Pruning



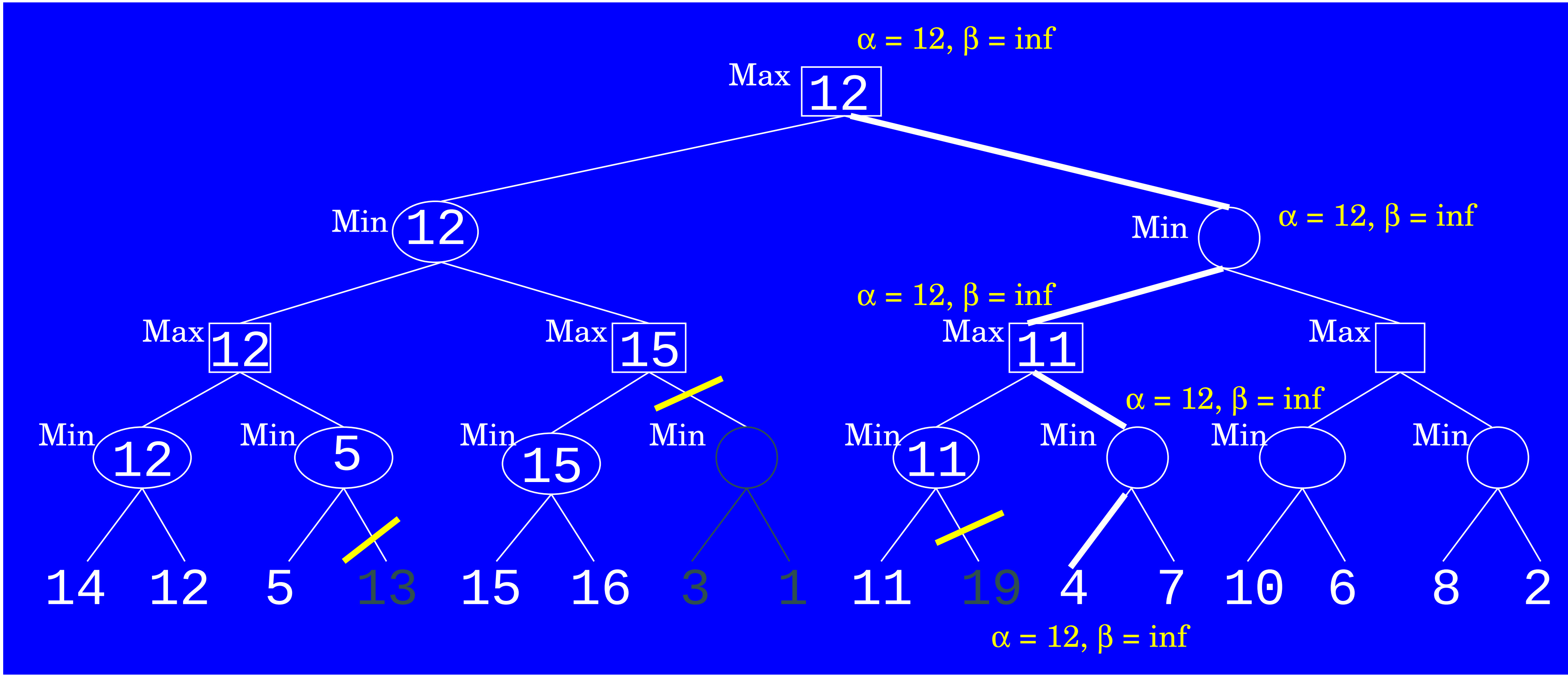
Alpha-Beta Pruning



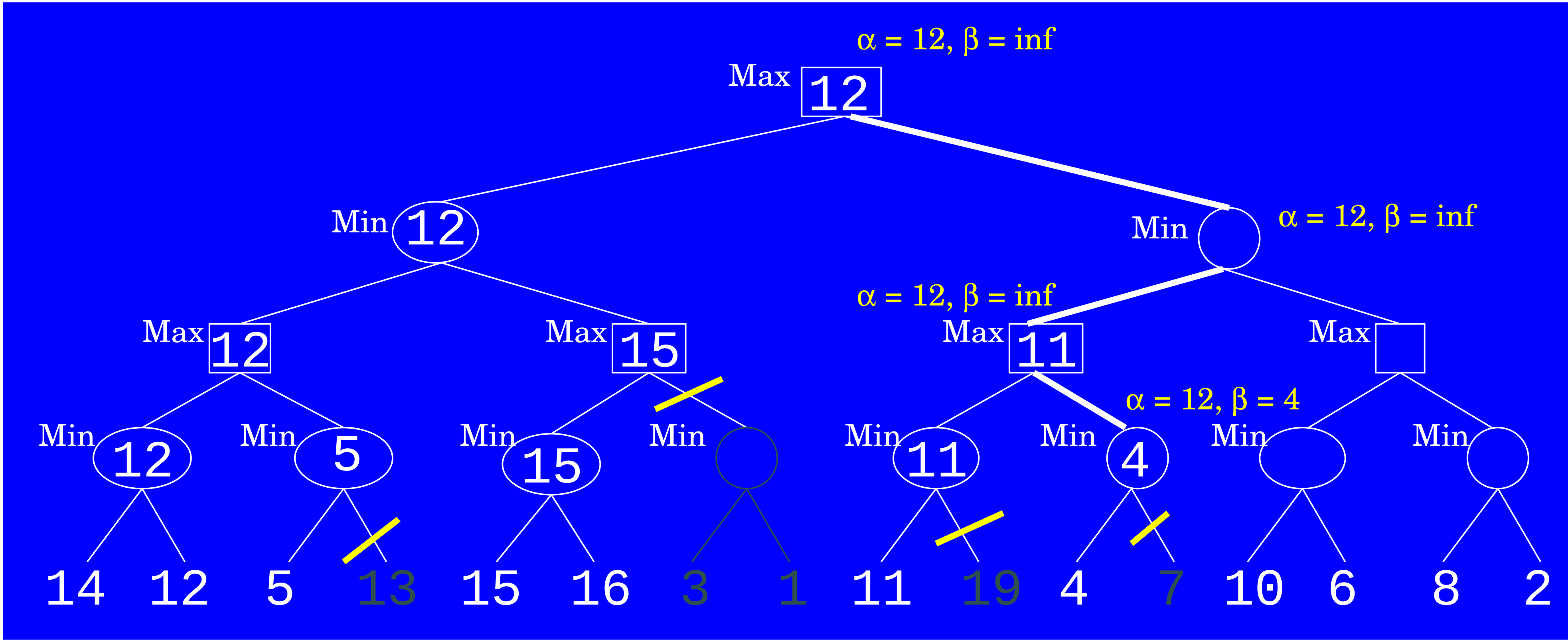
Alpha-Beta Pruning



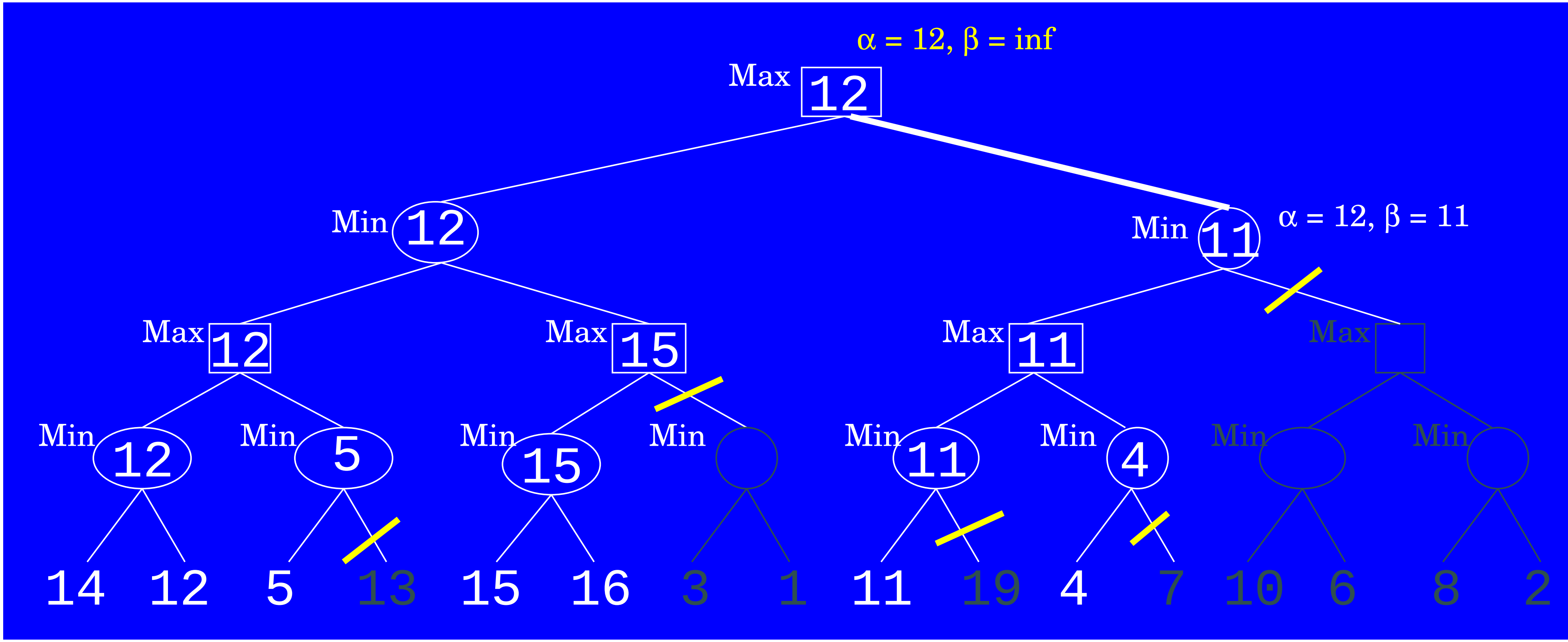
Alpha-Beta Pruning



Alpha-Beta Pruning



Alpha-Beta Pruning



Alpha-Beta Pruning

- We can cut large branches from the search tree *without* affecting final result
  - In the previous example, what would happen with similar values and a deeper tree?
- If we choose the order that we evaluate nodes (more on this in a minute…), we can dramatically cut down on how much we need to search

Evaluation Functions

- We can't search all the way to the bottom of the search tree
  - Trees are just too big
- Search a few levels down, use an evaluation function to see how good the board looks at the moment
- Back up the result of the evaluation function, as if it was the utility function for the end of the game

Evaluation Functions

- Chess:
  - Material - value for each piece (pawn = 1, bishop = 3, etc)
    - Sum of my material - sum of your material
  - Positional advantages
    - King protected
    - Pawn structure
- Othello:
  - Material – each piece has unit value
  - Positional advantages
    - Edges are good
    - Corners are better
    - “near” edges are bad

Evaluation Functions

- If we have an evaluation function that tells us how good a move is, why do we need to search at all?
    - Could just use the evaluation function
- If we are only using the evalution function, does search do us any good?

Evaluation Functions & α-β

- How can we use the evaluation function to maximize the pruning in alpha-beta pruning?

Evaluation Functions & α-β

- How can we use the evaluation function to maximize the pruning in alpha-beta pruning?
  - Order children of max nodes, from highest to lowest
  - Order children of min node, from lowest to highest
  - (Other than for ordering, eval function is not used for interior nodes)
- With perfect ordering, we need to search only $b^{d/2}$ (instead of b^d) to find the optimal move – can search up to twice as far

A simple example of the value of reasoning about which computations are relevant (a form of *meta-reasoning*)

Stopping the Search

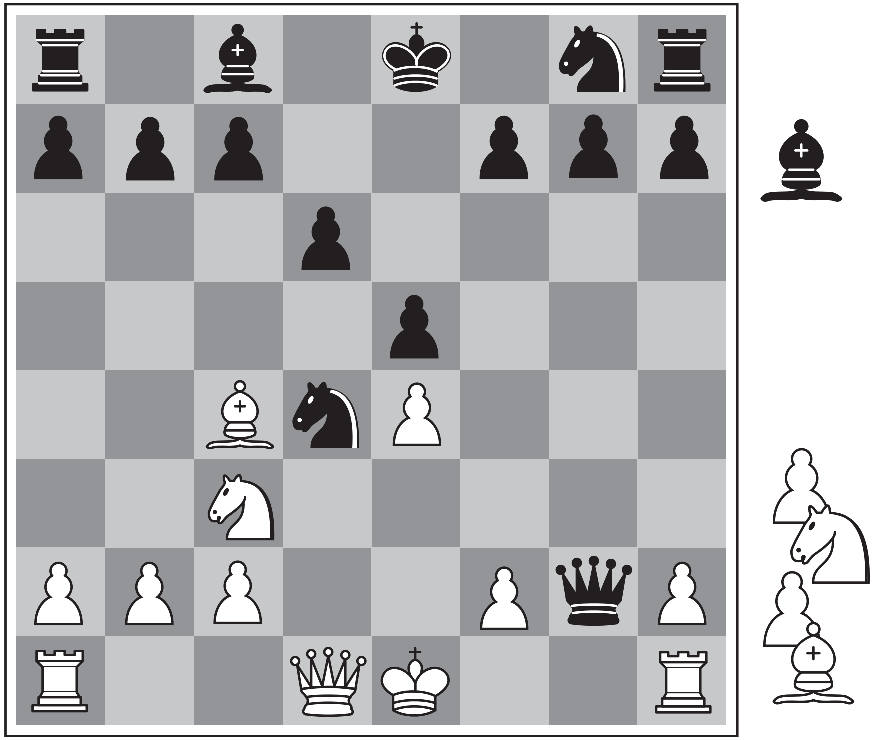
Still exponential!

- We can't search all the way to the endgame
  - Not enough time
- Search a set number of moves ahead
  - Problems?

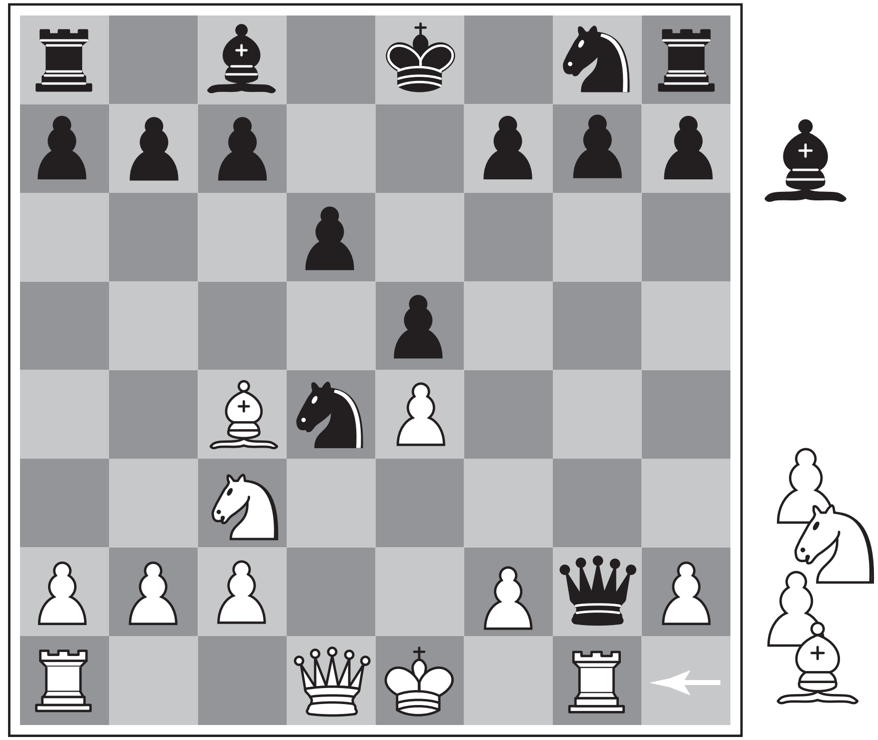
Stopping the Search

- We can’t search all the way to the endgame
  - Not enough time
- Search a set number of moves ahead
  - What if we are in the middle of a piece trade?
  - In general, what if our opponent is about to capture one of our pieces

Stopping the Search



(a) White to move



(b) White to move

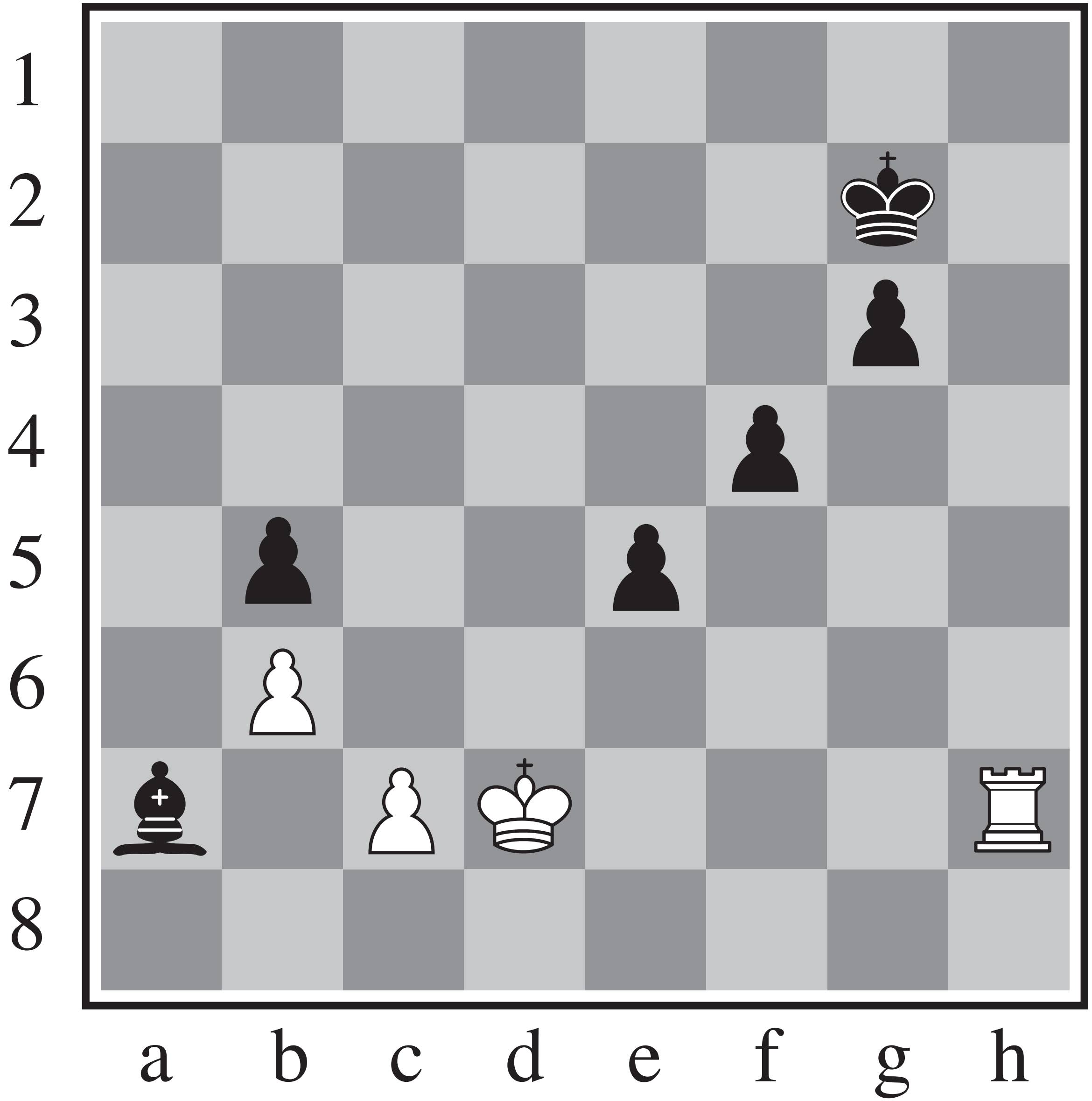
Stopping the Search

- Quiescence Search
  - Only apply the evaluation function to nodes that do not swing wildly in value
  - If the next move makes a large change to the evaluation function, look ahead a few more moves
  - Not increasing the search depth for the entire tree, just around where the action is
  - To prevent the search from going too deep, may restrict the kinds of moves (captures only, for instance)

Stopping the Search

- Horizon Problem
  - Sometimes, we can push a bad move past the horizon of our search
  - Not preventing the bad move, just delaying it
  - A position will look good, even though it is utlimately bad

Horizon Problem



Horizon Problem

- Singular Extensions
  - When we are going to stop, see if there is one move that is clearly better than all of the others.
  - If so, do a quick “search”, looking only at the best move for each player
  - Stop when there is no “clearly better” move
  - Helps with the horizon problem, for a series of forced moves
- Similar to quiescence search

Cutting off search

- Minimal change to alpha-beta search
- Does it work in practice?
  - $B^m = 10^6, b = 35 \rightarrow m = 4$
- 4-ply look ahead is a hopeless chess player
  - 4-ply $\approx$ human novice
  - 8-ply $\approx$typical PC, human master
  - 12-ply $\approx$ Deep Blue, Kasparov

State of the art

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a precomputed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.
- Chess: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

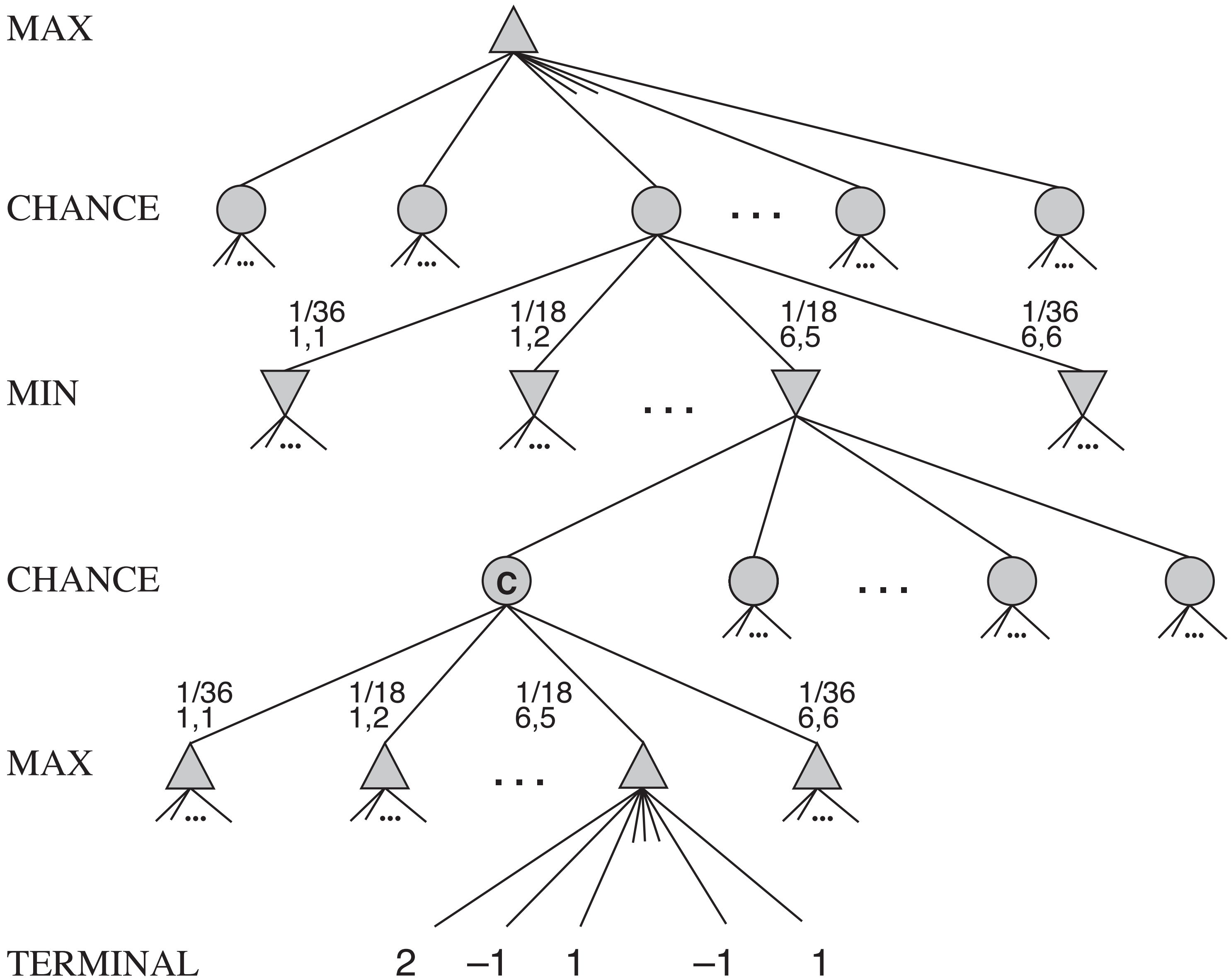
State of the art

- Othello: human champions refuse to compete against computers, who are too *good*.
- Go: human champions refuse to compete against computers, who are too *bad*. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

Adding Chance

- What about games that have an element of chance (backgammon, poker, etc)
- We can add chance nodes to our search tree
  - Consider “chance” to be another player
- How should we back up values from chance nodes?

Adding Chance



Adding Chance

- For Max nodes, we backed up the largest value:

$$\max_{s \in Sucessors(n)} Val(s)$$

- For Min nodes, we backed up the smallest

$$\max_{s \in Sucessors(n)} Val(s)$$

- For chance nodes, we back up the expected value of the node

$$\sum_{s \in Sucessors(n)} P(s)Val(s)$$

Adding Chance

- Adding chance dramatically increases the number of nodes to search
  - Braching factor b (ignoring die rolls)
  - n different dice outcomes per turn
  - Time to search to level m?

Adding Chance

- Adding chance dramatically increases the number of nodes to search
  - Braching factor b (ignoring die rolls)
  - n different dice outcomes per turn
  - Time to search to level m: $b^m n^m$

Summary

- Games are fun to work on!
- They illustrate several important points about AI
  - perfection is unattainable: must approximate
  - good idea to think about what to think about

Summary

- Min/Max trees
- Alpha-Beta Pruning
- Evaluation Functions
- Stopping the Search
- Playing with chance

Alpha-beta pseudocode, pt 1

```
def alpha-beta-search(state):
   v = max-val(state, -INF, INF)
   return action associated with v

def max-val(s, alpha, beta): #returns a value
   if end-state(s):
      return utility(s)
    v = -INF
    for s in successors(s):
       v = max(v, min-val(s, alpha, beta) )
       if v >= beta:
          return v
       alpha = max(alpha, v)
   return v
```

Alpha-beta pseudocode, pt 2

```
def min-val(s, alpha, beta): #returns a value
    if end-state(s):
        return utility(s)
    v = +INF
    for s in successors(s):
        v = min(v, max-val(s, alpha, beta) )
        if v <= alpha:
            return v
        beta = min(beta, v)
    return v
```