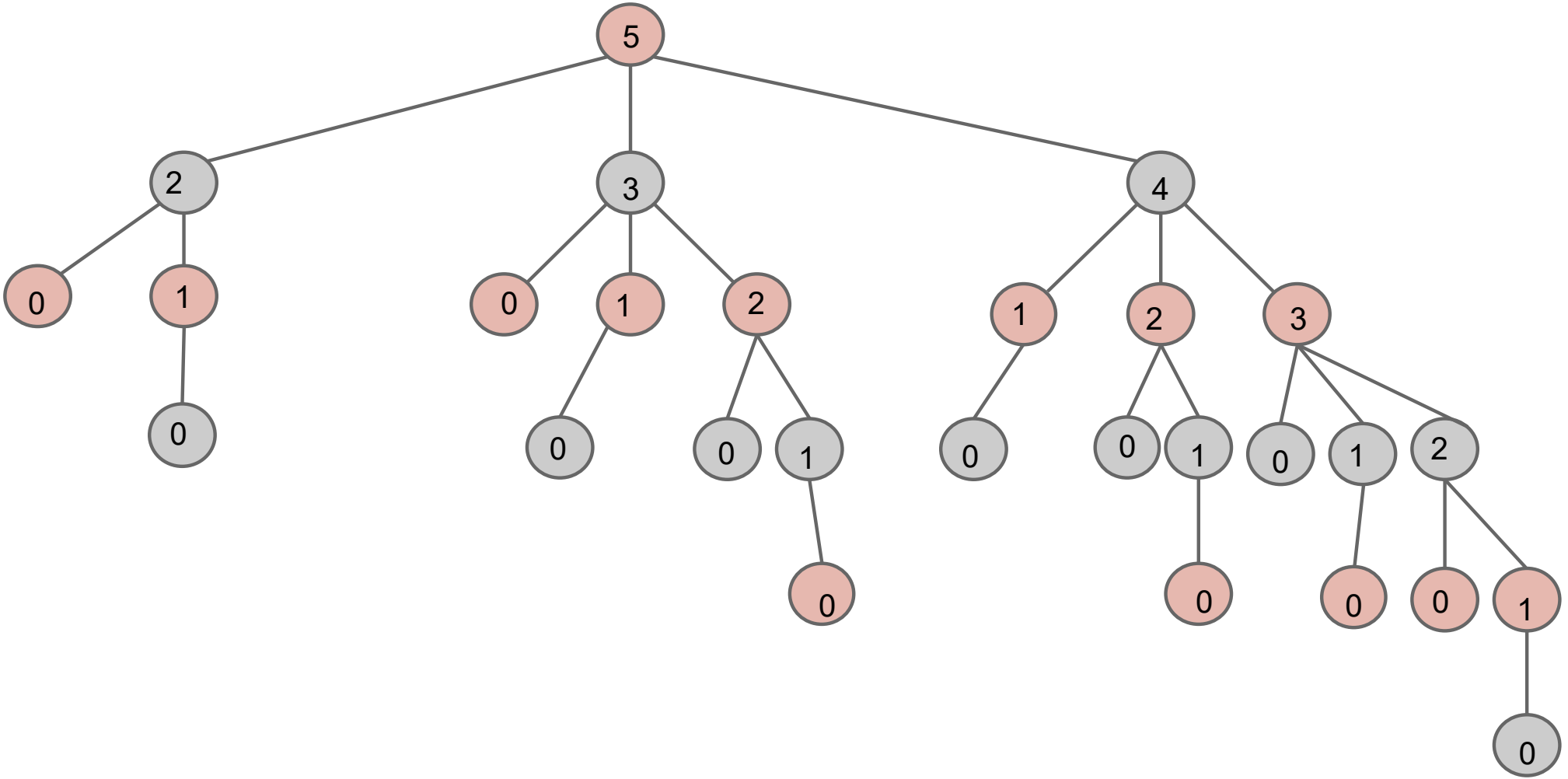
Minimax and Game Playing

Recap: Zero-Sum Games

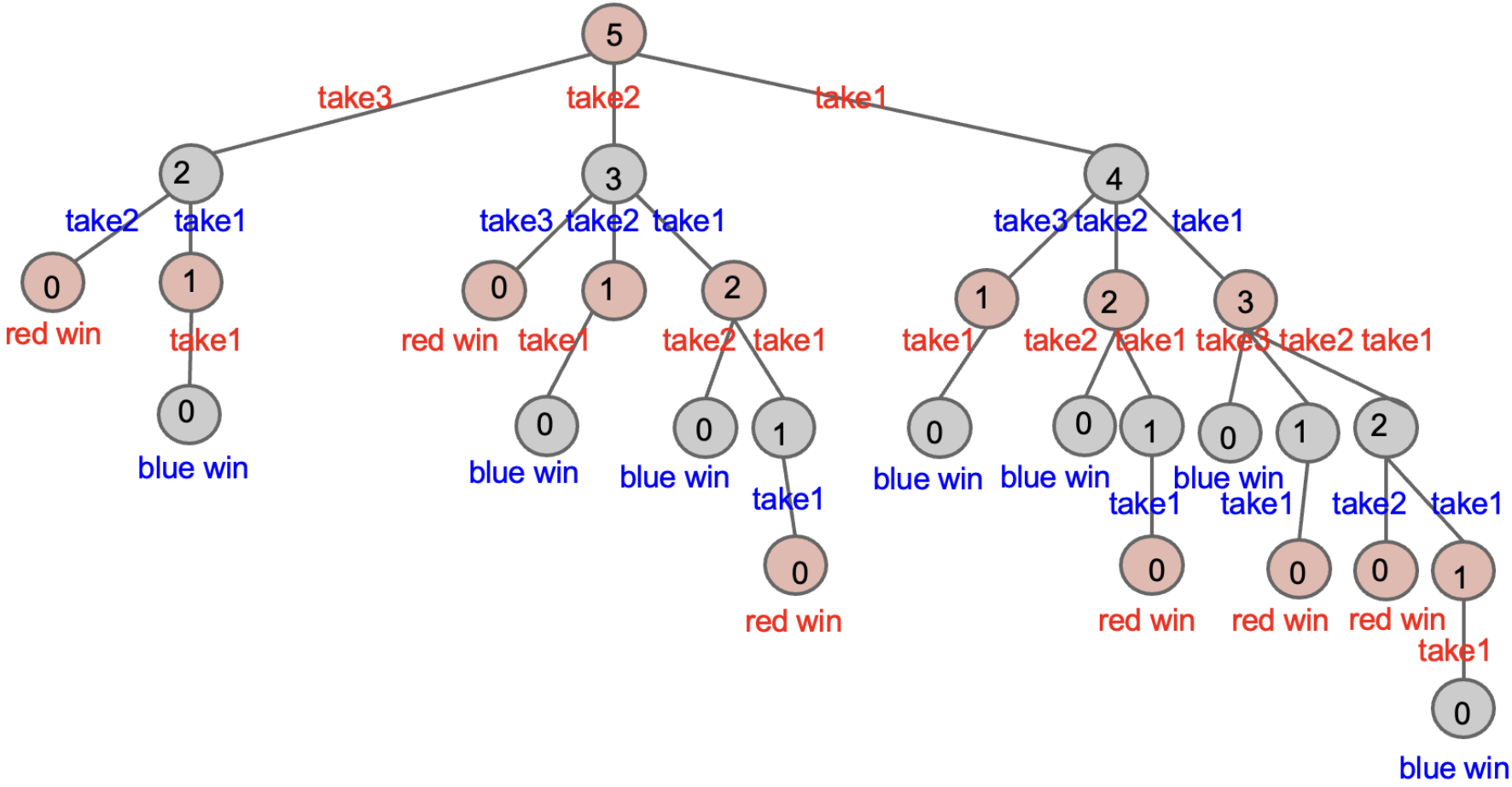
- Games where a positive reward for one player is a negative reward for the other
- Nim is an example of such a game
- The outcome of the game can be represented as a single number
- One player is trying to maximize this number
- The other is trying to minimize this number

Nim game tree for N=5



Recap: Nim (Other examples?)

Nim game tree for N=5



Setting up a game tree for search

- Players
  - MAX: the player who goes first
  - MIN: the player who goes second
- Values of terminal states
  - +1 = win for MAX
  - -1 = win for MIN
  - In some games it is possible to tie, which would be represented as 0

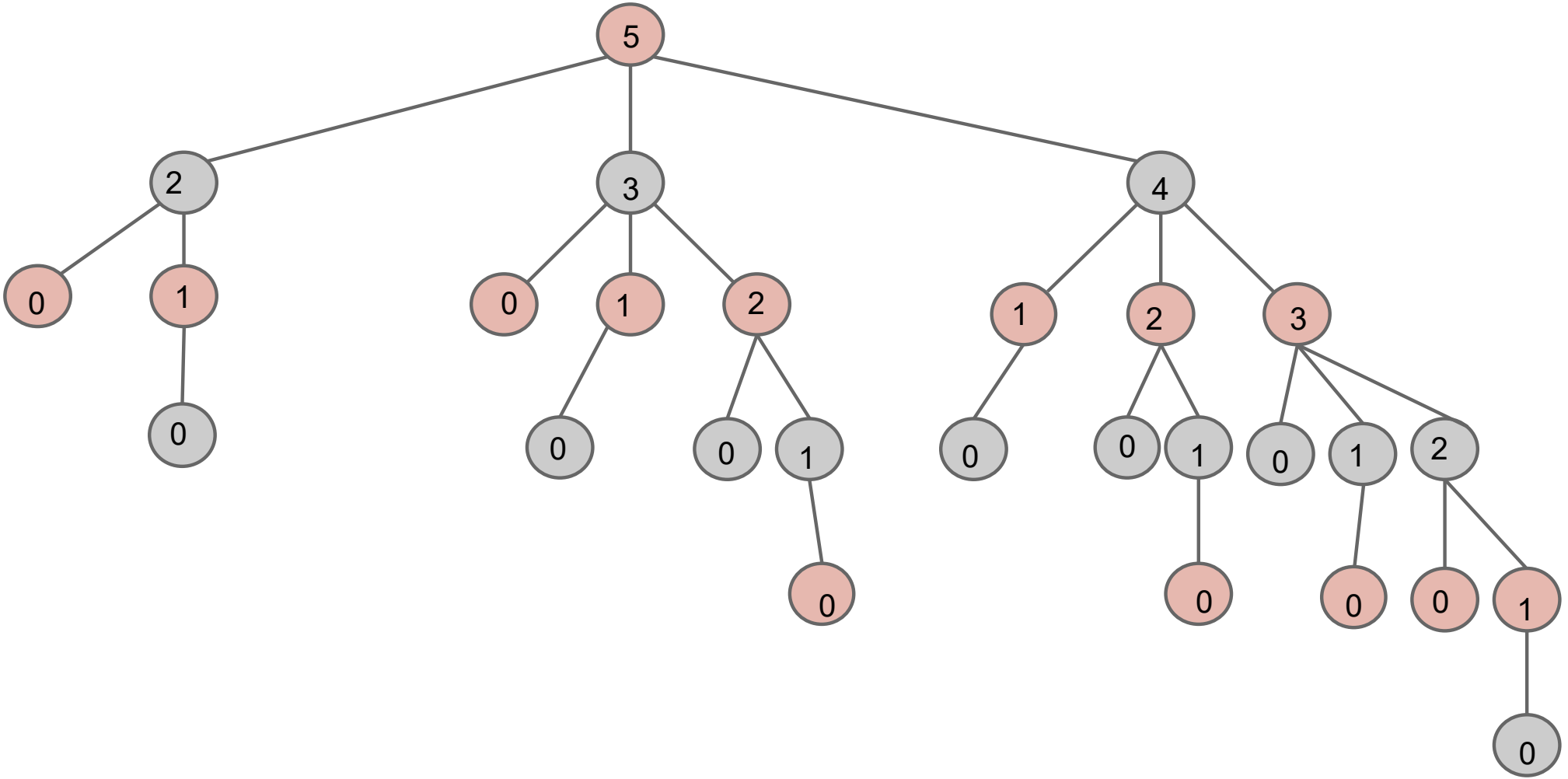
Setting up a game tree for search

Goal: Find the best move to make from the current state

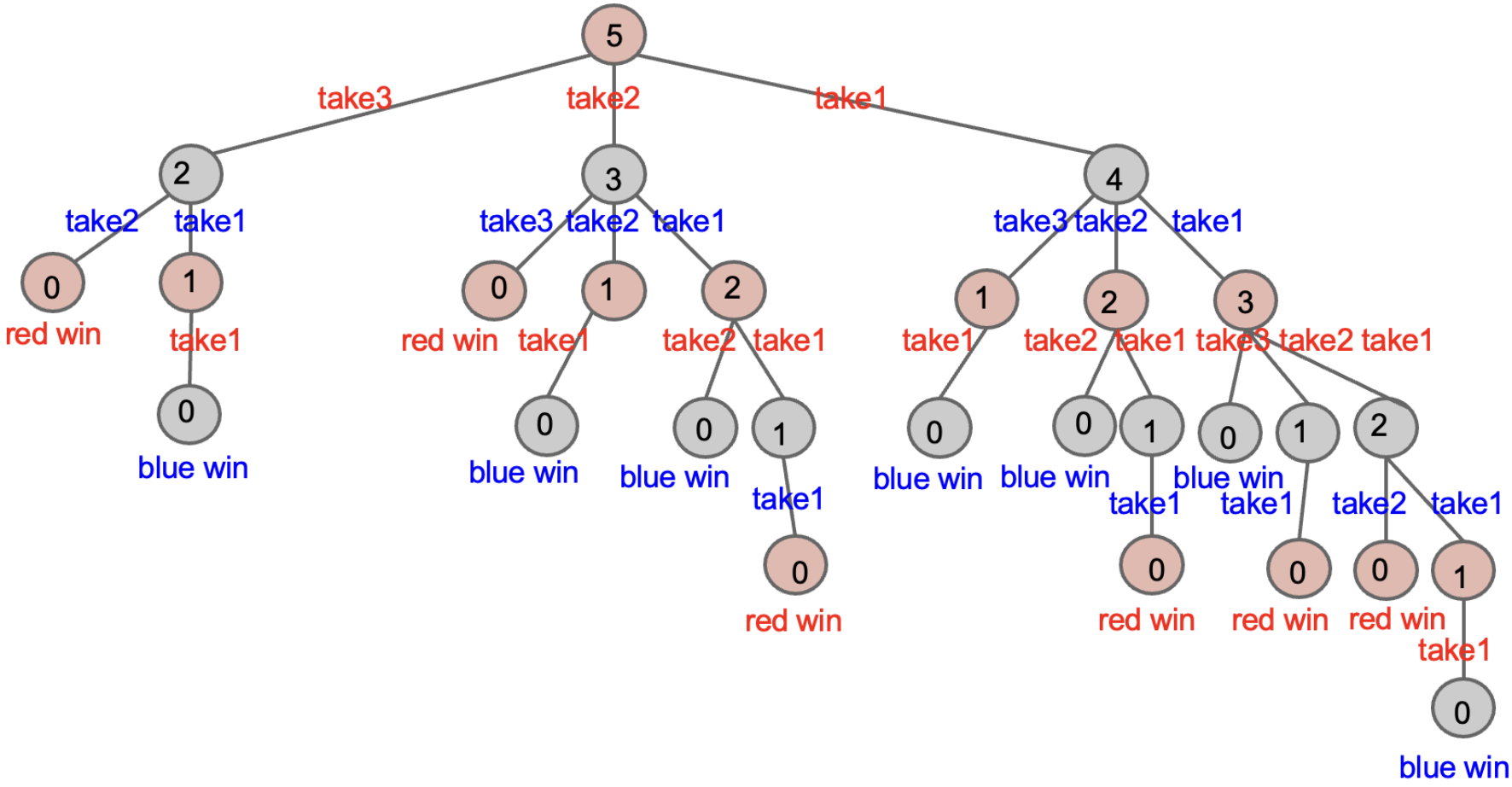
Assume: In the future each player will always make the best possible move (the one that will max/min the final value)

Minimax: Find the best move by propagating values up the tree from the terminal states!

Nim game tree for N=5



Bottom up propagation



Bottom up propagation

For finding the best *current* move:

- Is this approach time efficient?

- Is this approach space efficient?

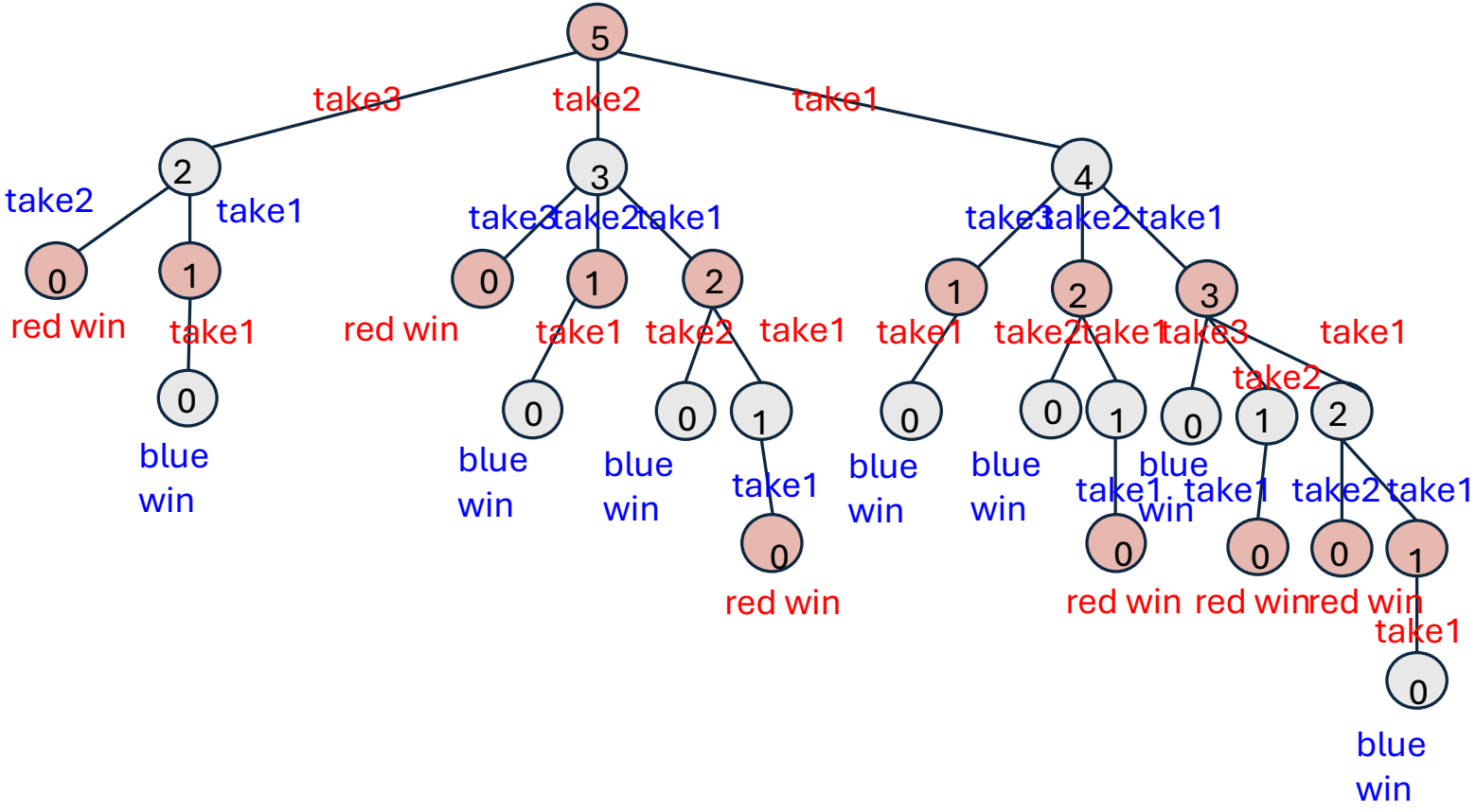
Bottom up propagation

For finding the best *current* move:

- Is this approach time efficient?
  - No, we have to explore the entire game tree!*
- Is this approach space efficient?
  - No, we have to store the entire game tree!*

**We don't necessarily need to construct the whole tree for the bottom up approach, but that's a story for another course*

Bottom up propagation



Recall: Depth first search

For a search tree: always explore the most recently discovered state

Advantage in state-space search?

Recall: Depth first search

For a search tree: always explore the most recently discovered state

Advantage in state-space search?

Only needed to store 1 path at a time – Space efficient!

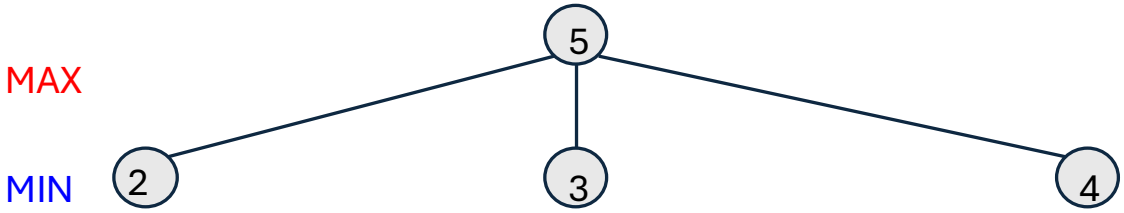
Let's try it with our Nim game tree!

Tracing through Minimax, for Nim N=5

Player1 is the maximizer

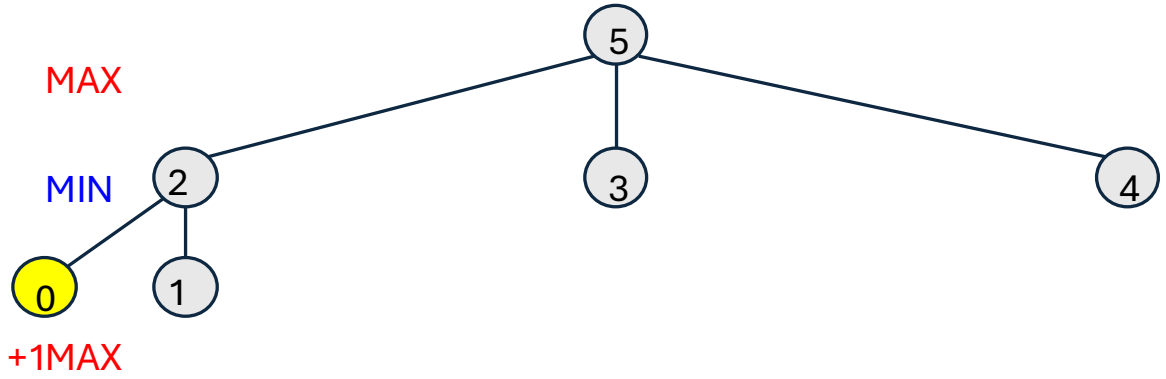
Tracing through Minimax, for Nim N=5

Initially, look at all possible moves from current game state



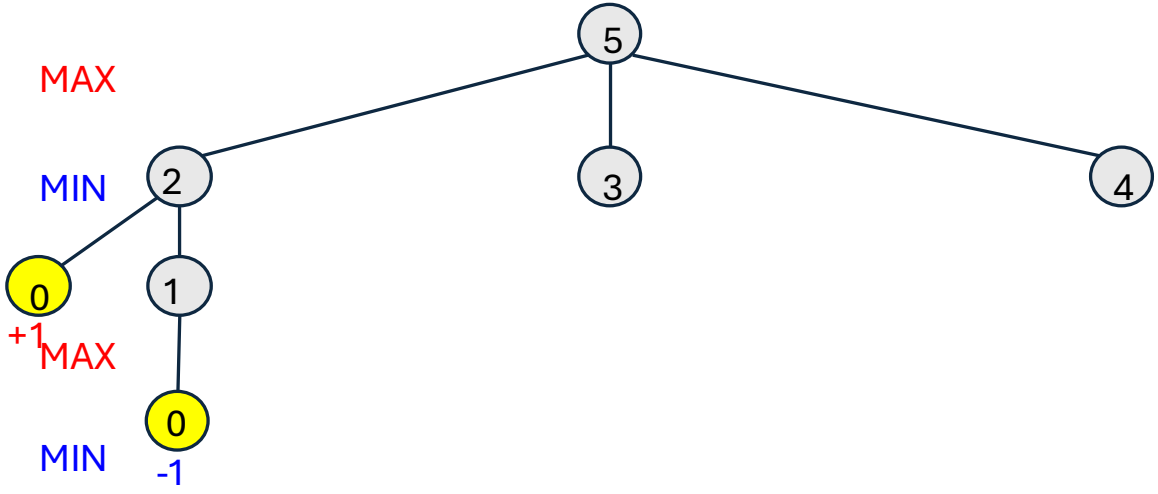
Tracing through Minimax, for Nim N=5

In a DFS way, explore deeper into the game tree



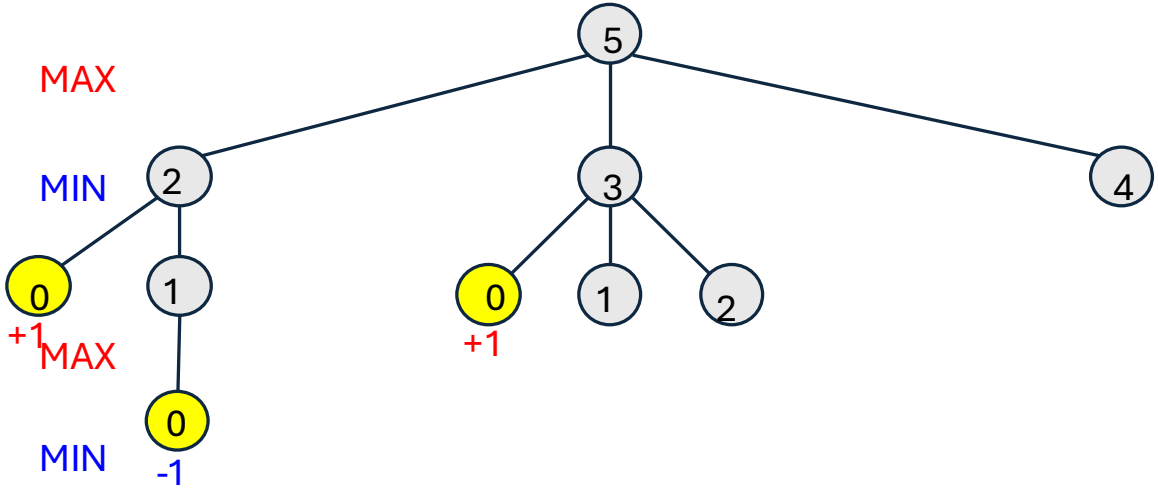
Tracing through Minimax, for Nim N=5

When terminal states are reached, backup the values from each player's perspective



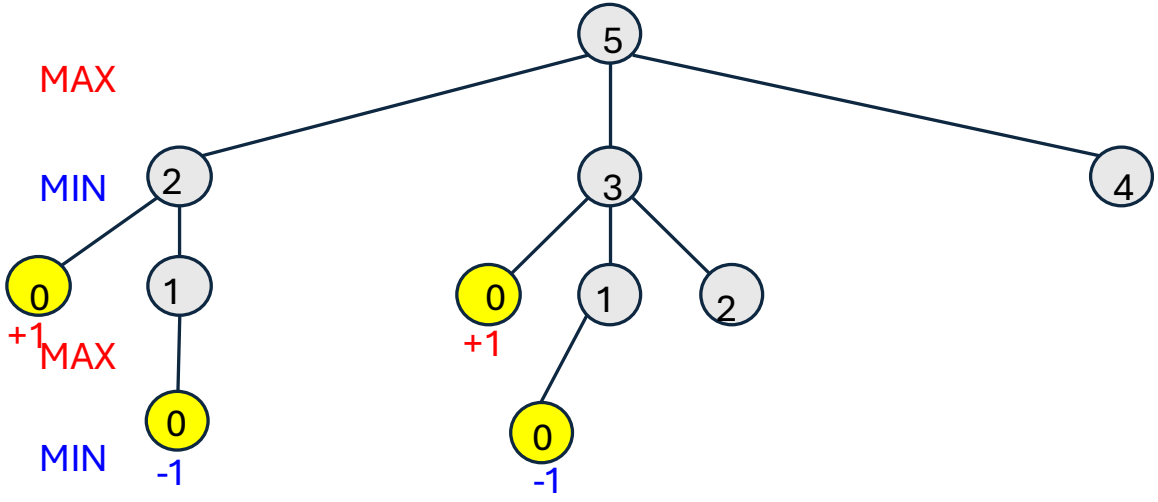
Tracing through Minimax, for Nim N=5

Continue exploring other moves in a DFS way



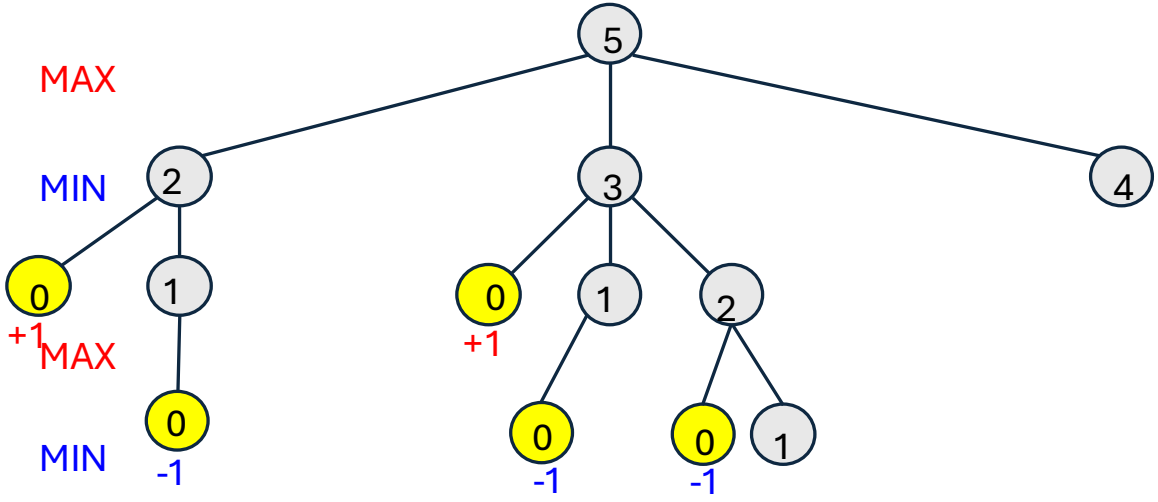
Tracing through Minimax, for Nim N=5

Continue exploring other moves in a DFS way



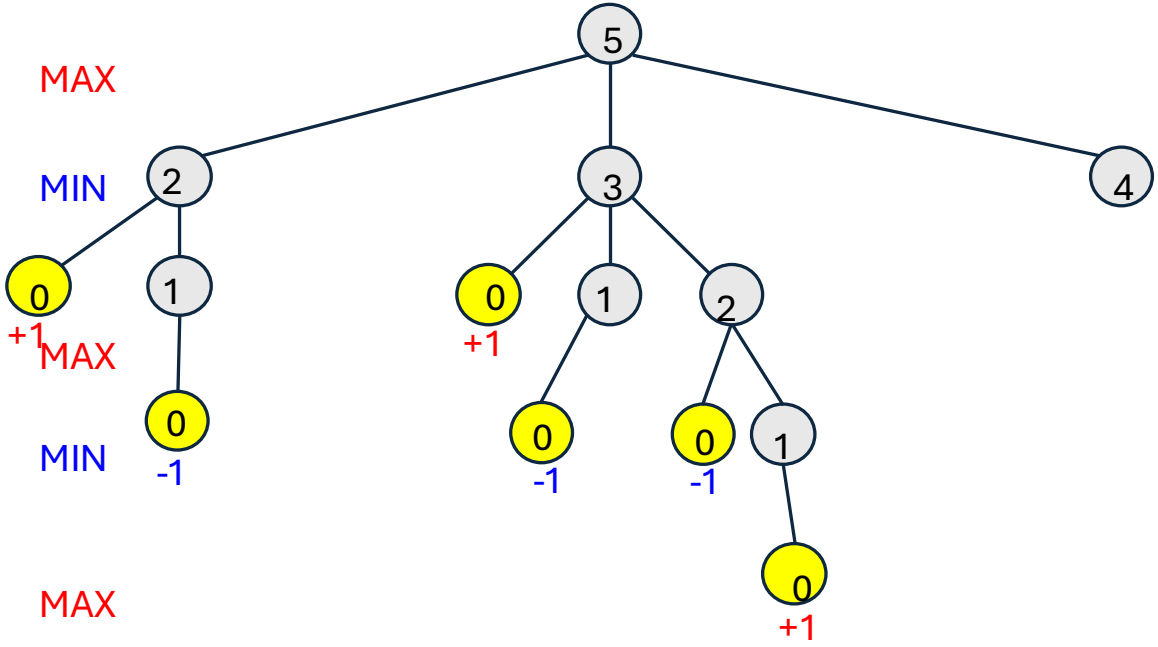
Tracing through Minimax, for Nim N=5

Continue exploring other moves in a DFS way



Tracing through Minimax, for Nim N=5

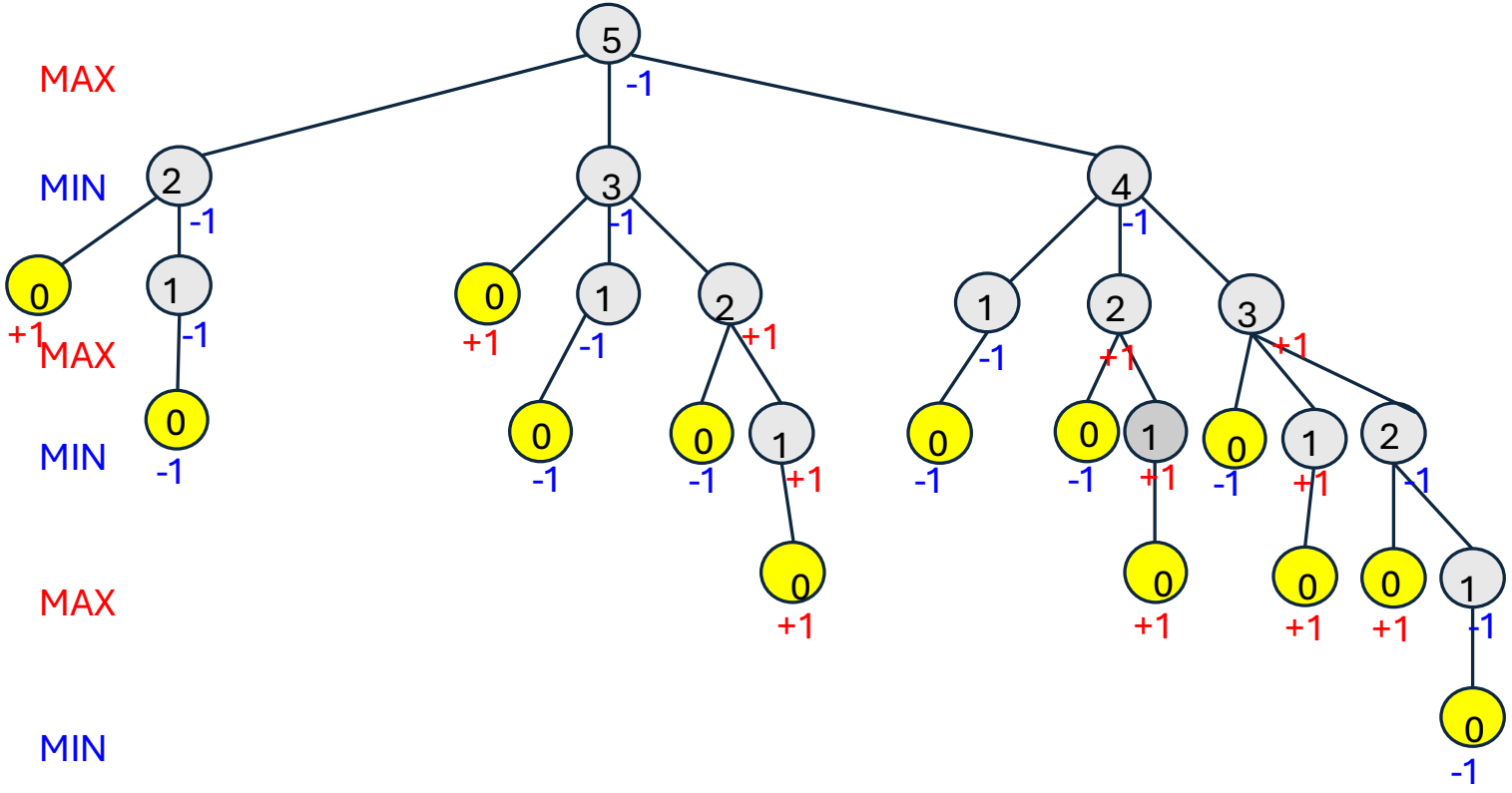
When terminal states are reached, backup the values from each player's perspective



Tracing through Minimax, for Nim N=5

Outcome of the entire search

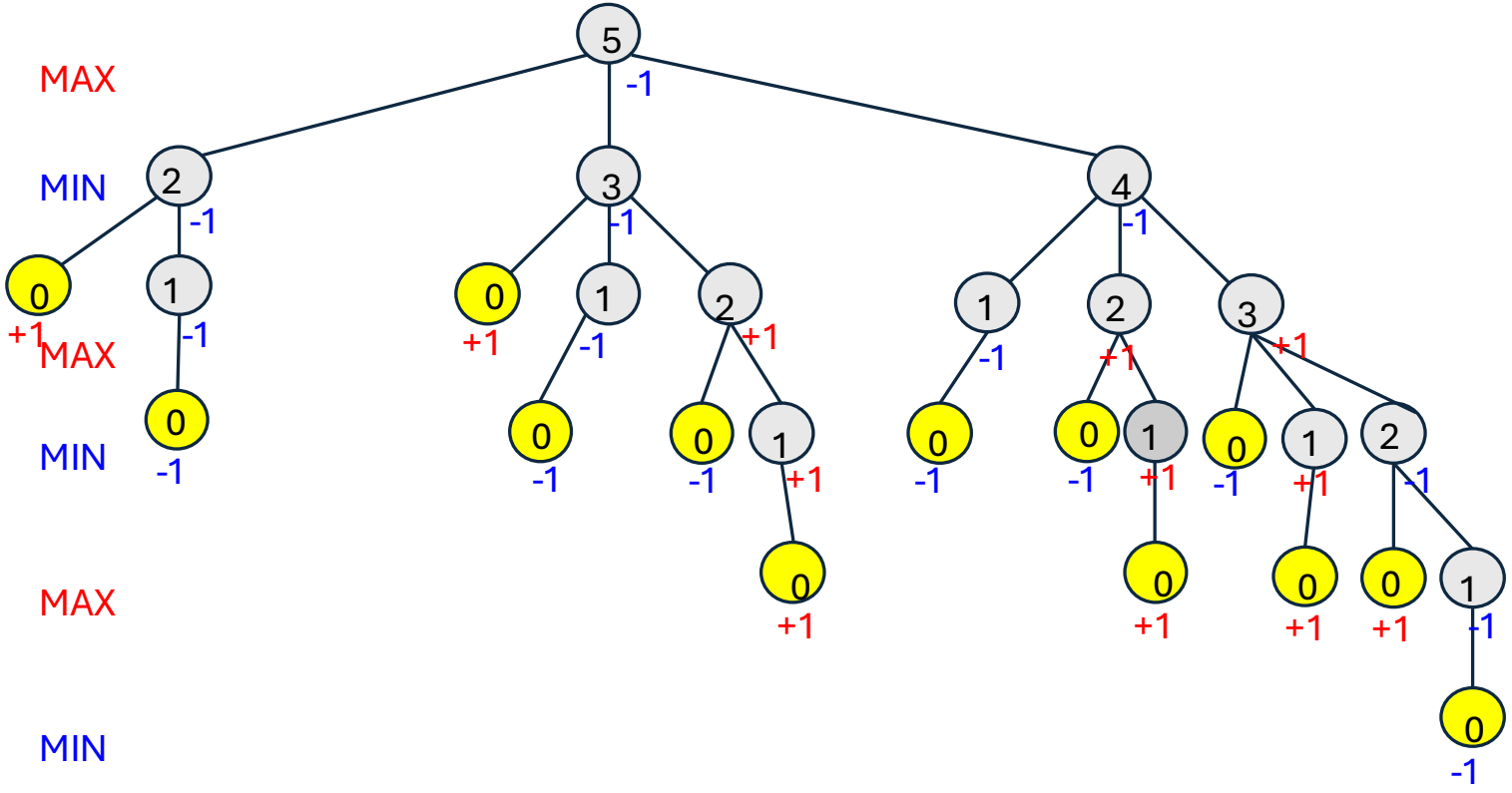
Notice that with optimal play, starting at N=5 is always a losing state



Tracing through Minimax, for Nim N=5

Outcome of the entire search

Notice that with optimal play, starting at N=5 is always a losing state



Recall: Search tree size

- Branching factor (*b*): Number of (new) states we can choose from any state
- Depth (*d*): Max number of moves to a terminal state

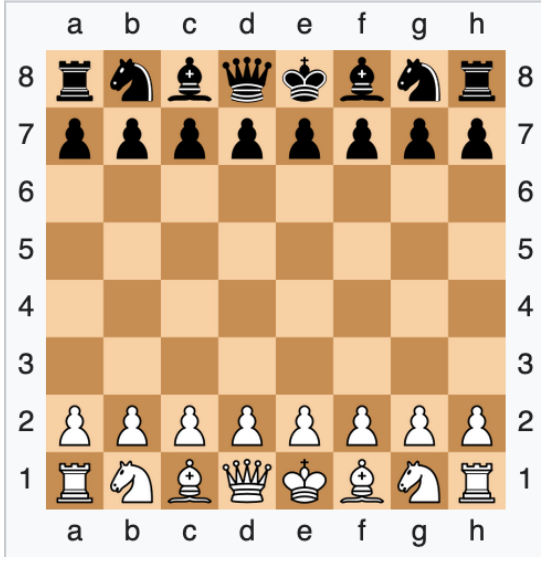
What is the size of a search tree? (*asymptotically in b and d*)

What are b and d for Nim?

Recall: Search tree size

- Branching factor (*b*): Number of (new) states we can choose from any state
- Depth (*d*): Max number of moves to a terminal state

What about chess?



Practicality of Minimax

- For most interesting games the game tree is too large to search to the end and to find optimal moves
- In chess, the branching factor is approximately 35 and games can last for 100 moves
- This creates a game tree of 35^{100} nodes which is approximately 10^{154}

Reading quiz

What is alpha-beta pruning primarily for?

A. Keeping a search tree balanced
B. Reducing the computational complexity of game tree search
C. Reducing computational complexity of local search
D. Estimating the true cost to a winning state

Would the minimax algorithm we just learned work for a card game, like poker, uno or go-fish?

A. Yes
B. No

Could we use local-search algorithms like hill-climbing or simulated annealing to search a game tree?

A. Yes
B. No

What about Chess AIs? (e.g. Stockfish)

Idea: Use depth-first search to a limited depth and try to approximate the value of states with a *static evaluation function*

Evaluation functions:

- Look at a game state without knowing any context and try to assign it a value
- Performance of a game playing program is highly dependent on this evaluation
- Using a good evaluation function allows us to make informed decisions about which current move is likely to lead to good situations later

- *Similar idea to the heuristic we used in A Star!*

Consider this tic-tac-toe board

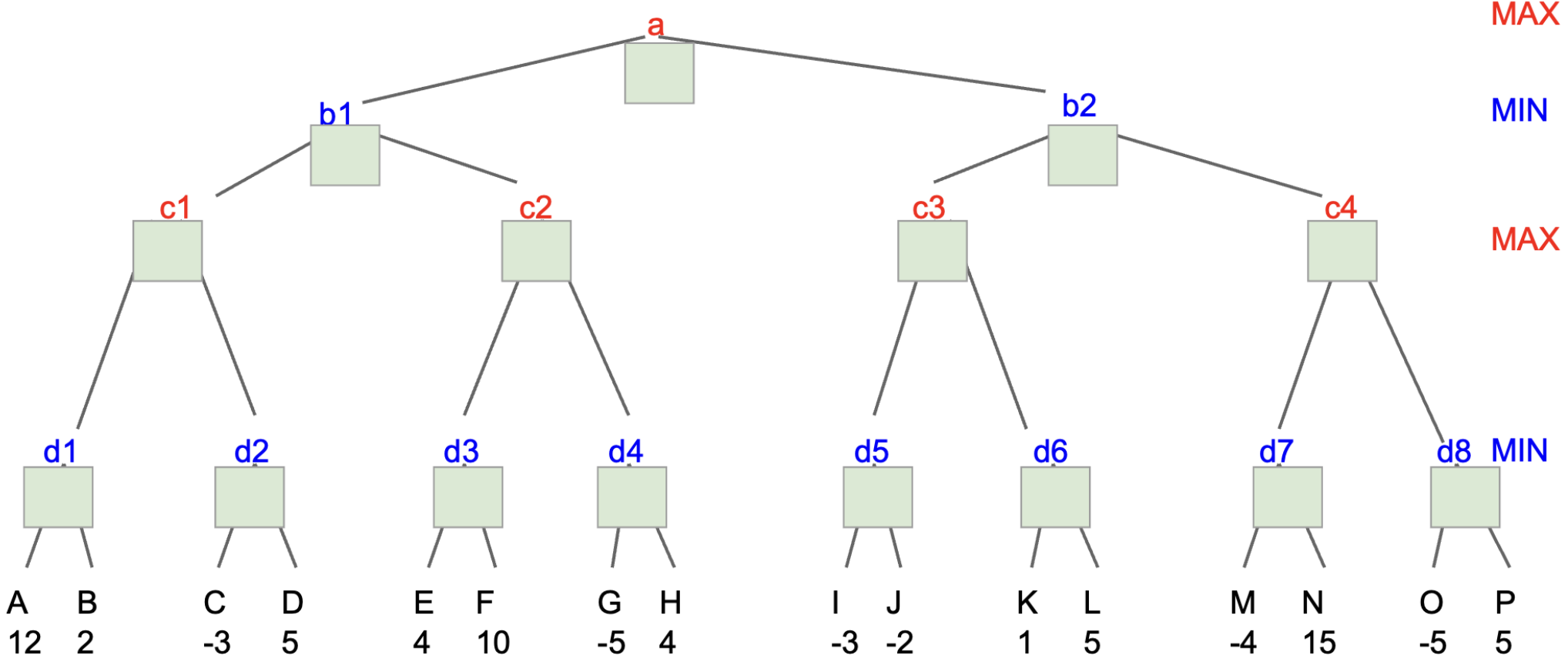
X		X
O	X	
		O

- X went first and is the maximizer
- O went second and is the minimizer
- It is O's turn
- A static evaluator would try to score this board positively if it is a better state for X or negatively if it is a better state for O

Features of a good evaluation function

- When a terminal state is reached, score it correctly
- Should be efficient to calculate since it will be called many, many times
- Should be strongly correlated with the actual chances of winning
- Exactness is less important than trying to get the relative values correct

Consider game tree where at depth limit of 4 static evaluation value is given
At each node track bestValue found by bounded minimax



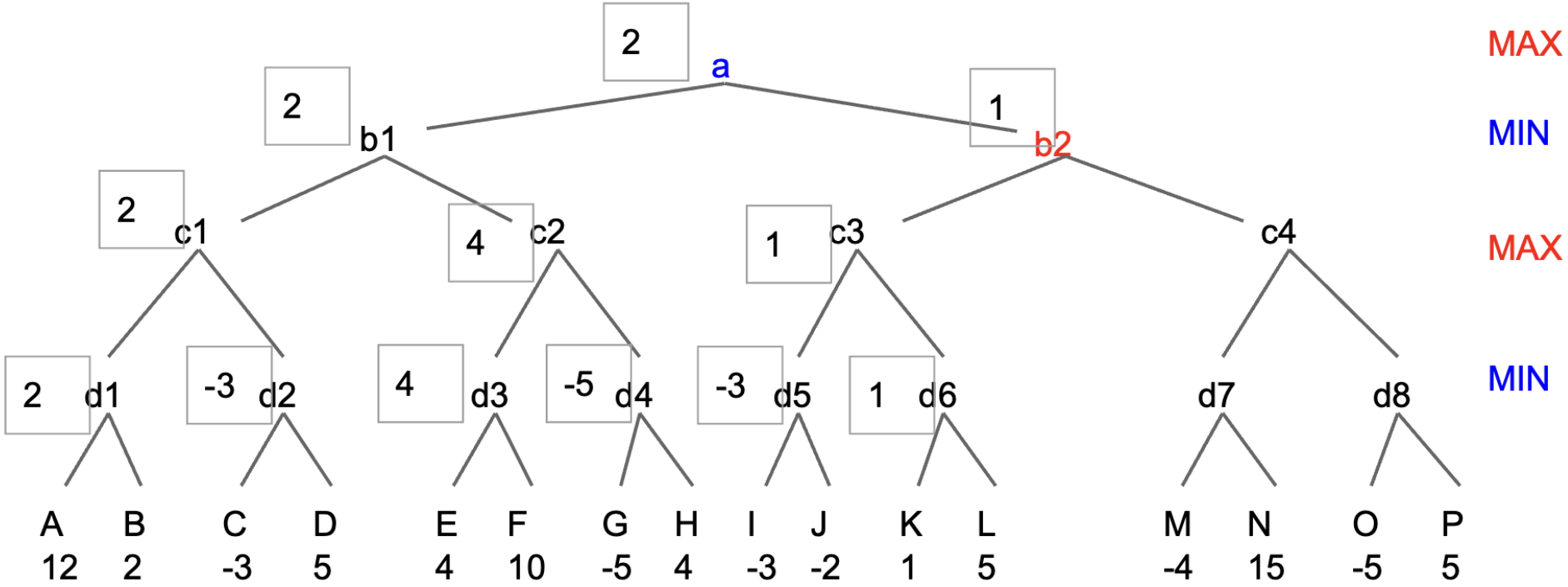
Pseudocode for Bounded Minimax

```
# Returns best value found, and as a side effect also updates best move found at root
bounded_min_max(state, depth)
   if depthLimit reached or state is terminal:
      return staticEval(state)
   # initialize the bestValue depending who's turn it is (+1 for max, -1 for min)
   bestValue = turn * -float("inf")
   for each move from state:
      determine the next_state
      # Recursive call
      value = bounded_min_max(next_state, depth+1)
      if player is maximizer
         if value > bestValue
            bestValue = value
            if depth is 0, update bestMove to current move
      else # player is minimizer
         if value < bestValue
            bestValue = value
            if depth is 0, update bestMove to current move
   return bestValue
```

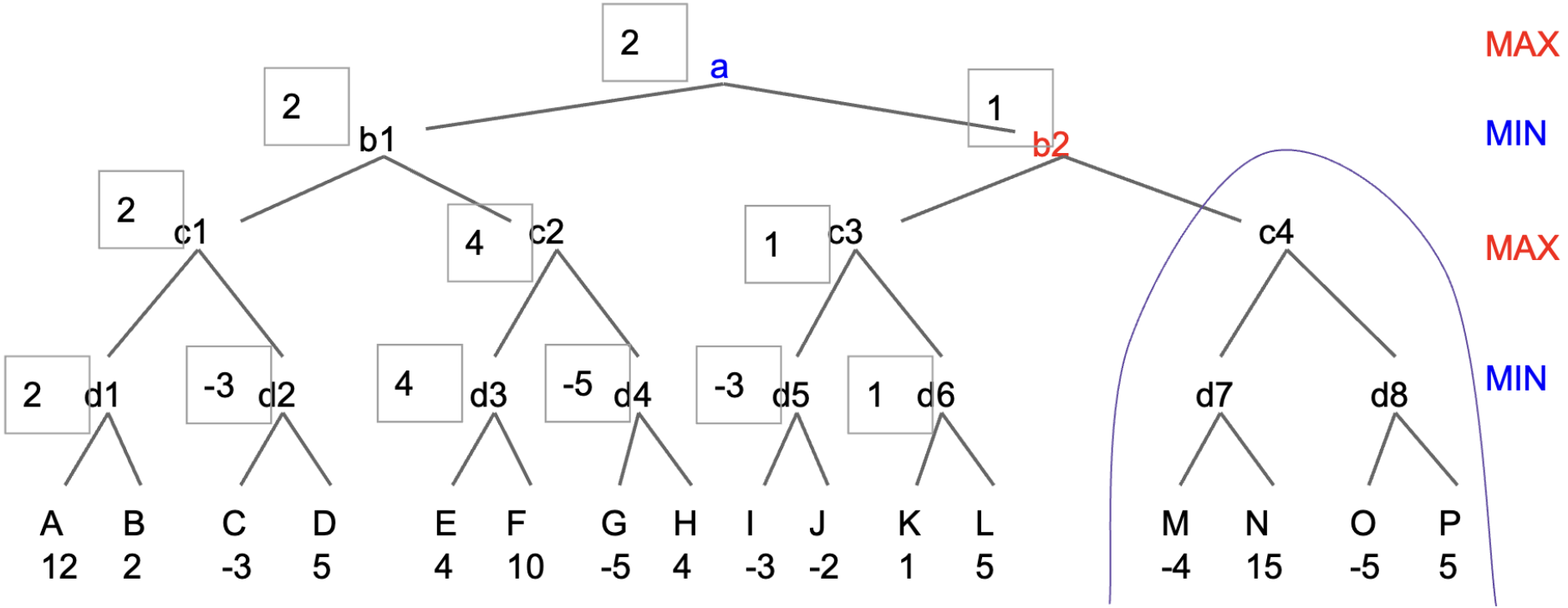
Alpha-Beta pruning

- A method for eliminating branches during the search that will not affect the outcome
- Minimax with Alpha-Beta pruning always returns the same scores as Minimax would when using the same depth limit
- However, it may return a different action in the cases of ties
- Potentially much faster than Minimax allowing us to search deeper in the same amount of clock time

Consider Minimax in mid search; what do you notice about the current best values at node a (maximizer) and node b2 (minimizer)?



Pruning is possible at this point



At node a, the bestValue so far is 2, and since a is a maximizer it will never accept anything lower.
At node b2, the bestValue so far is 1, and since b2 is a minimizer it will never accept anything higher.
Therefore we don't need to search the c4 branch! We know that this branch will not be chosen by Minimax.

How Alpha-Beta pruning works

- `alpha` is the current best score for MAX
- `beta` is the current best score for MIN
- Initialize `alpha` to `-inf`
- Initialize `beta` to `+inf`
- In recursive calls
  - MAX levels update `alpha`
  - MIN levels update `beta`
- Whenever `alpha >= beta`, cutoff search

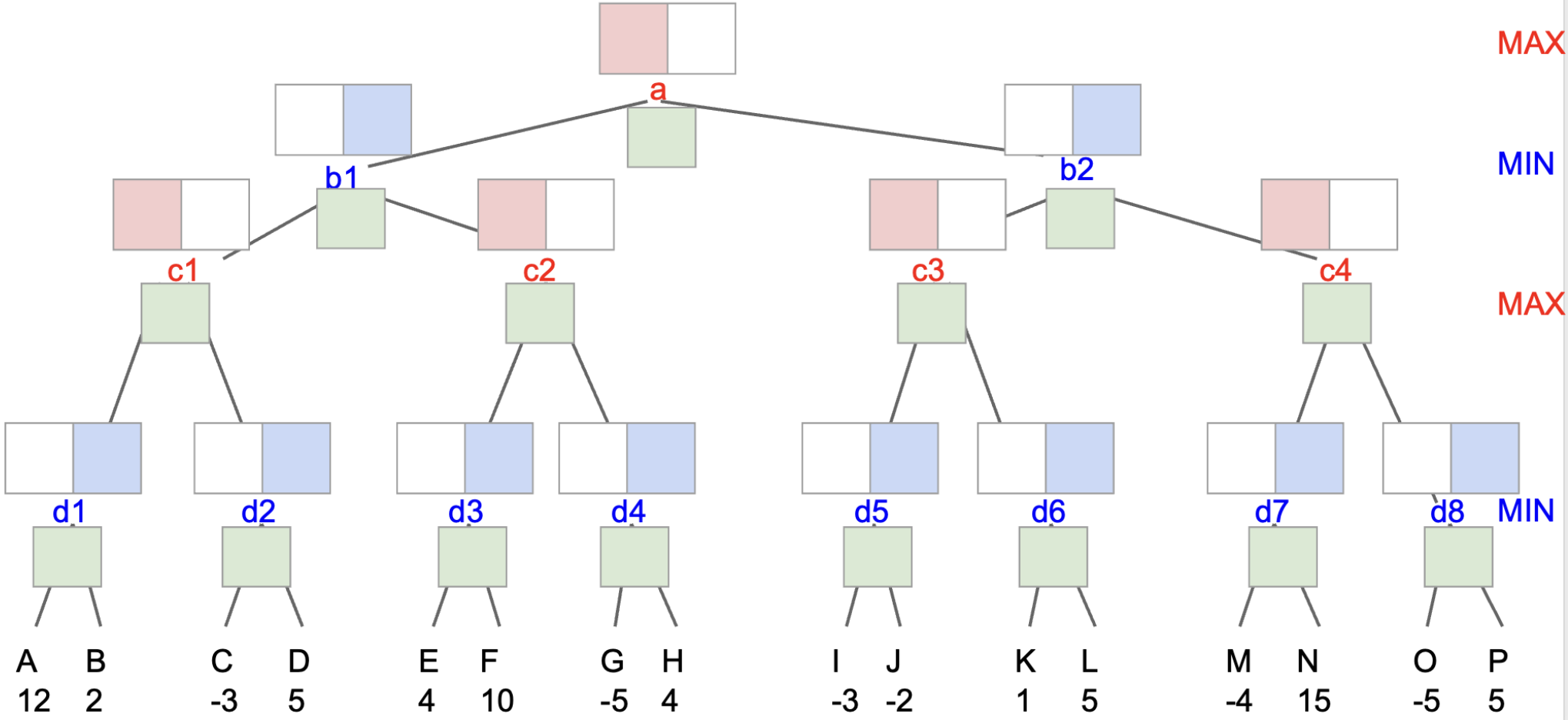
Pseudocode for Alpha Beta (very similar to Minimax)

```
pruning_search(state, depth, alpha, beta)# Initially alpha is -inf, beta is +inf
   if depthLimit reached or state is terminal:
      return staticEval(state)
   # initialize the bestValue depending who's turn it is
   bestValue = turn * -float("inf")
   for each move from state:
      determine the next_state
      # Recursive call
      value = pruning_search(next_state, depth+1, alpha, beta)
      if player is maximizer
         if value > bestValue
            bestValue = value
            if depth is 0, update bestMove to current move
         alpha = max(value, alpha)
      else # player is minimizer
         if value < bestValue
            bestValue = value
            if depth is 0, update bestMove to current move
         beta = min(value, beta)
      if alpha >= beta
         break from for loop # pruning remaining children
   return bestValue
```

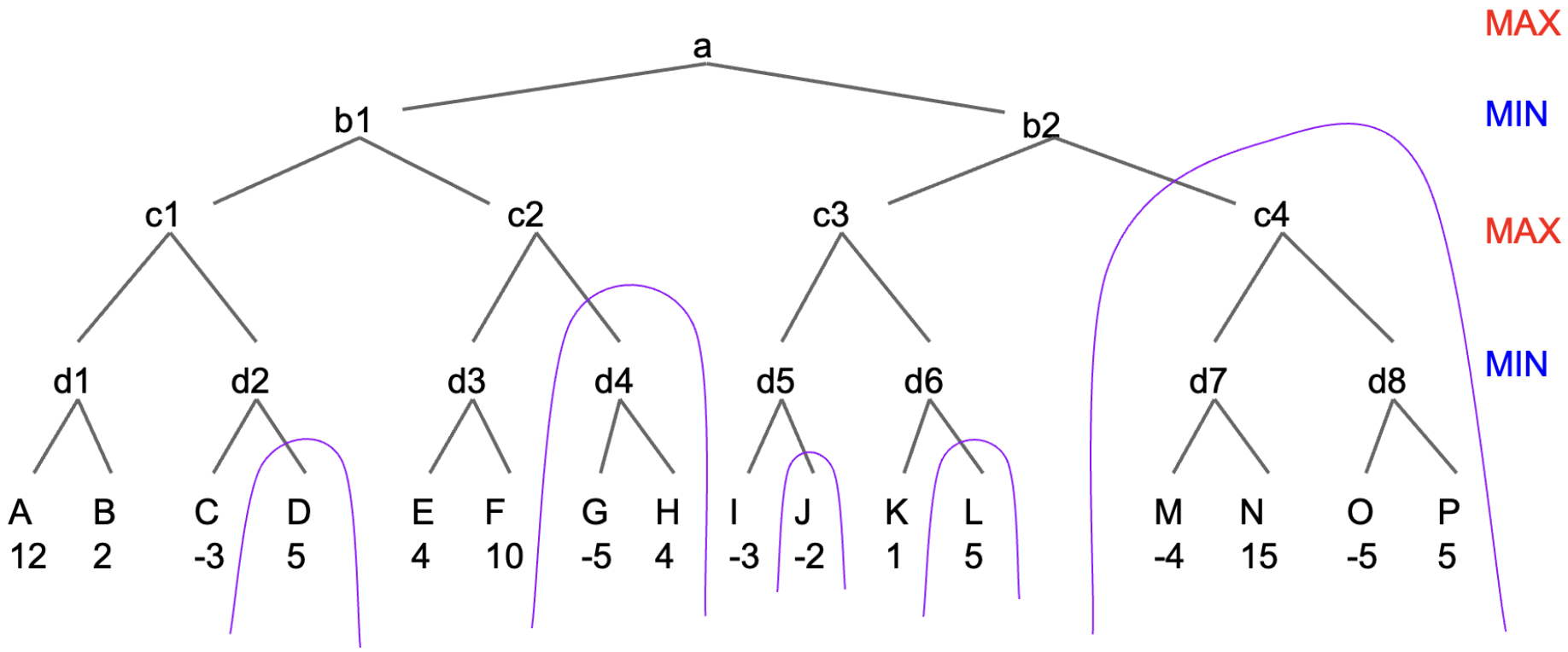
Trace through example

- Start with `alpha=-inf, beta=+inf` at the root
- Each recursive call gets passed down the current values of alpha and beta as parameters
- MAX nodes update `alpha` when a higher value is found
- MIN nodes update `beta` when a lower value is found
- if `alpha >= beta`, cut off search of remaining children for that node

At each node track bestValue, alpha, and beta
MAX updates alpha, MIN updates beta, if alpha >= beta, prune



Alpha-beta pruning results



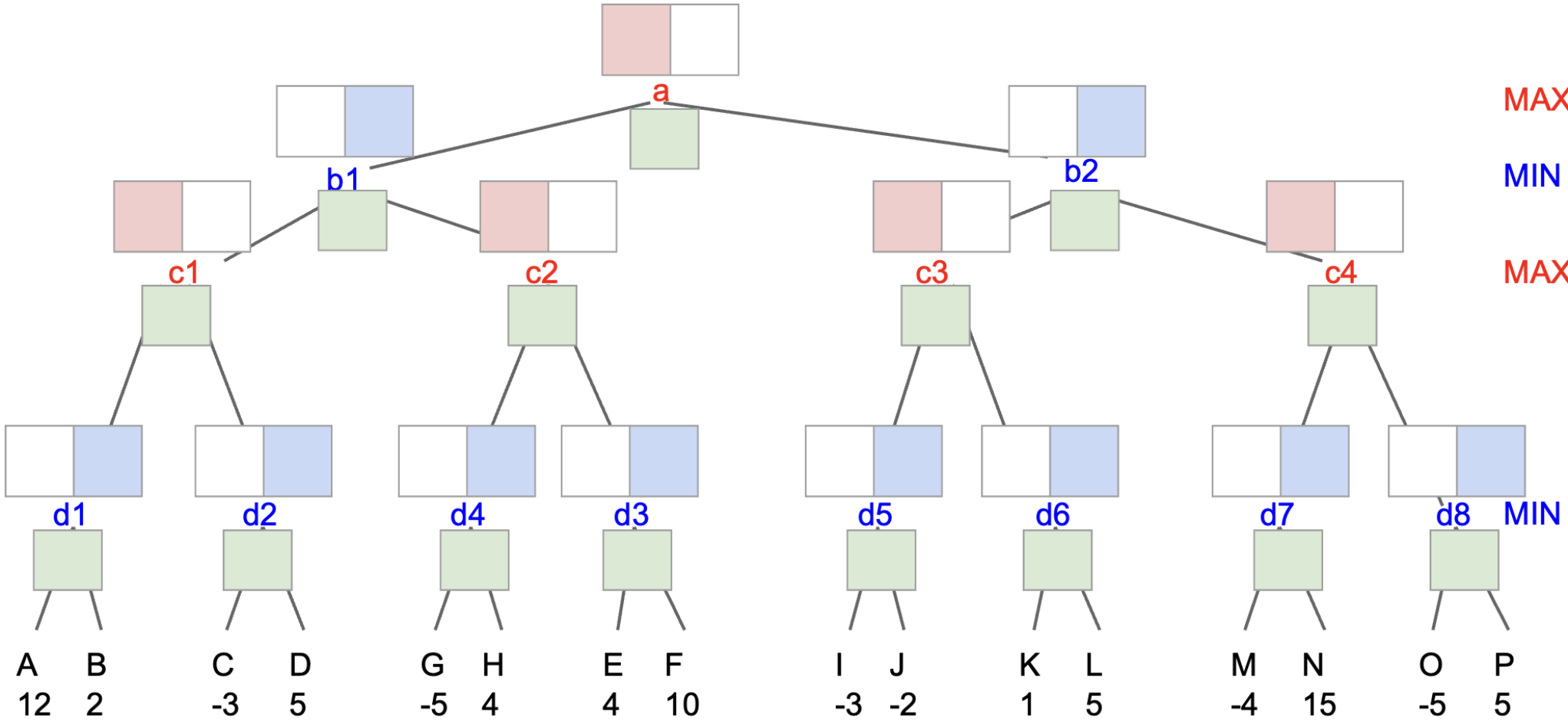
On the previous example the following branches will be pruned

- At d2, should cut off D when (α=2, β=-3)
- At c2, should cut off d4 when (α=4, β=2)
- At d5, should cut off J when (α=2, β=-3)
- At d6, should cut off L when (α=2, β=1)
- At b2, should cut off c4 when (α=2, β=1)

The order children are expanded matters

- For MIN nodes, seeing lower valued children first improves pruning
- For MAX nodes, seeing higher valued children first improves pruning
- Therefore, if you have some idea about which children might be better, you should order them to improve speed of search
- Try re-ordering the children in the previous example to see that this is the case

Re-ordered the children of c2, does this change what gets pruned?



Why do we need Alpha-Beta pruning?

- In adversarial search, the deeper you can search, the better the estimate of the best move to make
- Alpha-Beta pruning finds branches in the game tree that can be ignored
- By eliminating some branches, we may be able to search deeper in the same amount of time

Does minimax mimic the way humans play?

Similar	Different