





## Sept 28: Games

Let's recap last time! How do the Genetic Algorithm and Simulated Annealing differ or relate?

1. There is no random new candidates in the genetic algorithm.
2. There is no random probability of acceptance in the genetic algorithm.
3. They're actually the same algorithm.
4. There is no parallel to "mutation" in annealing.

	$x_0$	$x_1$	$x_2$	$x_3$
0				
1				
2				
3				

# Announcements and To-Dos

Announcements:

1. HW 2 due tonight: lots of searching!

(Zach brief  
office hour:  
5:05 - 5:30)

Last time we learned:

1. An in-class notebook, playing with annealing.

Minute form commentary:

1. HW going ok?

# Genetic Algorithm Recap

Initialization:

1. Begin with a **population** of  $k$  randomly generated states.
2. Each individual must somehow be represented as a **string** (like DNA) from some finite alphabet.

Specimen	Fitness	P(Reproduce)	Breeding Pop	Crossover	Mutation
28439198					
87932185					
12197835					
90271049					

# Genetic Algorithm Recap

Then we loop over generations:

1. Each specimen can be evaluated: it has a fitness/survival level, and from that an estimated probability that it's "good enough" to reproduce.

*Handwritten notes: "prior" above Specimen, "seen" above Fitness, "from prior" above P(Reproduce)*

Specimen	Fitness	P(Reproduce)	Breeding Pop	Crossover	Mutation
28439198	10	18%			
87932185	15	27%			
12197835	25	45%			
90271049	6	10%			

*Handwritten notes:*  
 $\uparrow$   
 heuristic  
 $f(\text{state})$     $f(\text{specimen})$

# Genetic Algorithm Recap

Then we loop over generations:

1. Each specimen can be evaluated: it has a fitness/survival level, and from that an estimated probability that it's "good enough" to reproduce.
2. From the population eligible for reproduction, select pairs of parents, which may appear multiple times

Specimen	Fitness	P(Reproduce)	Breeding Pop	Crossover	Mutation
28439198	10	18%	12197835		
87932185	15	27%	87932185		
12197835	25	45%	12197835		
90271049	6	10%	28439198		

## Genetic Algorithm Recap

Then we loop over generations:

1. Each specimen can be evaluated: it has a fitness/survival level, and from that an estimated probability that it's "good enough" to reproduce.
2. From the population eligible for reproduction, select pairs of parents, which may appear multiple times
3. "Breed" those pairs, in some random way picking traits/values from each.

Specimen	Fitness	P(Reproduce)	Breeding Pop	Crossover	Mutation
28439198	10	18%	12197835	12197185	
87932185	15	27%	87932185	87932835	
12197835	25	45%	12197835	28437835	
90271049	6	10%	28439198	12199198	

## Genetic Algorithm Recap

Then we loop over generations:

1. Each specimen can be evaluated: it has a fitness/survival level, and from that an estimated probability that it's "good enough" to reproduce.
2. From the population eligible for reproduction, select pairs of parents, which may appear multiple times
3. "Breed" those pairs, in some random way picking traits/values from each.
4. Add some added randomness to open truly new states by **mutating** strings with some small, independent probability

from prior ↓      breeding

Specimen	Fitness	P(Reproduce)	Breeding Pop	Crossover	Mutation
28439198	10	18%	12197835	12197185	1 <u>3</u> 197185
87932185	15	27%	87932185	87932835	87932 <u>0</u> 35
12197835	25	45%	12197835	28437835	28437835
90271049	6	10%	28439198	12199198	<u>9</u> 2199198

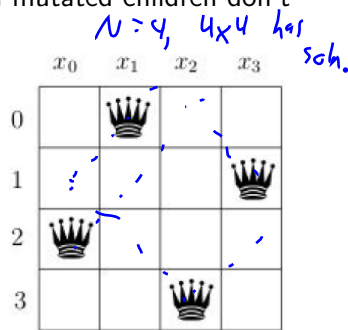
## Genetic Algorithm Addendum

One concern with the genetic algorithm: what if the crossover and mutated children don't even satisfy the problem to begin with?

Consider the “N-queens” problem: how many “Queen” pieces in chess can we add to a chess board? Queens can not share a column, row, or diagonal (45 degrees) with another queen.

To the right is a state that constitutes one of the solutions to the 4-Queens problem, that we might represent as

STATE = [2,0,3,1]. *repeat: share a row*











We could consider using the genetic algorithm: take the string 2031 and breed it to look for *more* solutions. Or take a string like 1234 and use it to look for a solution at all.



## Genetic Algorithm Addendum

Suppose our algorithm generated a mutated or child state of  $STATE = [1,0,3,1]$ . Since any string with a repeating value represents a shared row, we should immediately discard this state, or repair the “broken chromosome” at the end of it.

If we modify  $STATE = [1,0,3,1]$  to  $STATE = [1,0,3,2]$ , this is a technique known as constraint satisfaction. *replace w/ an unobserved row* *or [2,0,3,1]* 

	$x_0$	$x_1$	$x_2$	$x_3$
0				
1				
2				
3				

We may often embed these types of corrections into the objective function. Our objective function might be trying to minimize the overlap between queens, but we could also force it to only consider strings that are permutations of  $[0,1,2,3]$  with no redundant rows.

We could also just modify the reproduction function itself to somehow only allow “feasible” states: crossovers would involve swapping locations of characters rather than e.g. copying substrings.

We may start our discussion of games by classifying what we mean in terms of the different task environments.

In **searching**, we assumed an environment with no adversary.

1. The solution was a method for *finding* the goal state.
2. Heuristics and constraint satisfactions can help find optimal solutions (faster).
3. We had an evaluation function: the (estimated) cost from start to goal through any given node.
4. Examples: path planning, scheduling activities, etc.

In **games**, we have an adversary.

1. The solution is a *strategy*, which specifies moves for every possible opponent reply.
2. Time and memory limits often force approximate solutions
3. Examples: chess, checkers, backgammon (possibly Among Us?)

## Zero Sum Games

Our focus will be on deterministic, turn-taking, two-player games. These are **zero-sum** games with perfect information. The board state is always fully observable.

A **zero-sum** game is a game where one participant's gain (or loss) is exactly balanced by losses (or gains) of the other participant.

Zero Sum



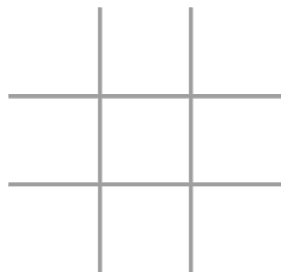
Non-Zero Sum



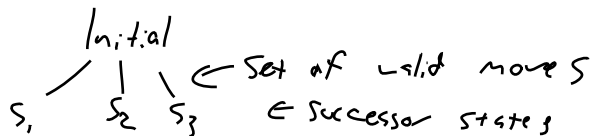
## Playing Games

To model a game we need to capture that each participant is trying to maximize their own gains. Put another way: as a player your goal is to **both** *maximize* your gains and **minimize** your opponent's. These lead to building a model:

1. We have two players, MAX and MIN
2. MAX goes first, and then they take turns until the game is over.
3. We could think of doing a search:
  - 3.1 Initial State is the board
  - 3.2 Define:  $s$ , current state
  - 3.3  $\text{def TO\_PLAY}(s)$  : whose move it is at  $s$ ?
  - 3.4  $\text{def ACTIONS}(s)$  : what legal moves are there?
  - 3.5  $\text{def RESULTS}(a,s)$  : what results from move  $a$ ?
  - 3.6  $\text{def GAME\_OVER}(s)$  : Is  $s$  an end state?
  - 3.7  $\text{def UTILITY}(s,p)$  : the payoff to player  $p$  if the game “ends” in state  $s$ .



# Playing Games

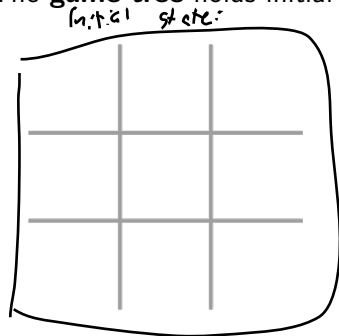


Navigation of game decisions can lead to a game decision tree. The **game tree** holds initial state, actions, and results.

1. Nodes are game states
2. Edges are actions/moves

We can work down a tree until we reach one or some **terminal states**

9 possible  
Successors



whose turn is it?

Player 1's turn

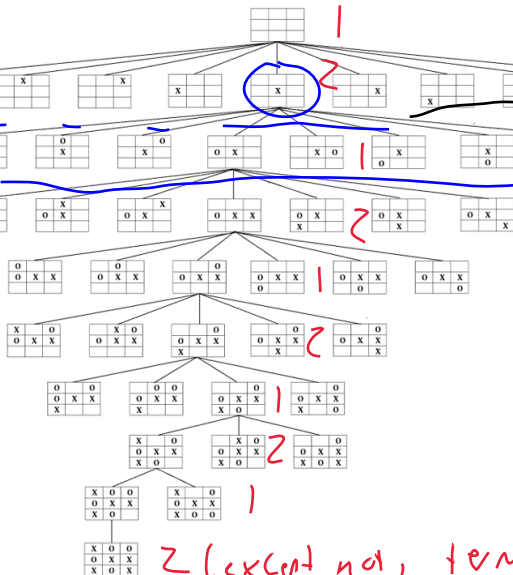
9 options

Player 1  
by road

Player 1's turn

Player 2's turn

Player 2's options



2 (except not, terminal).

## Playing Games

Navigation of game decisions can lead to a game decision tree. The **game tree** holds initial state, actions, and results.

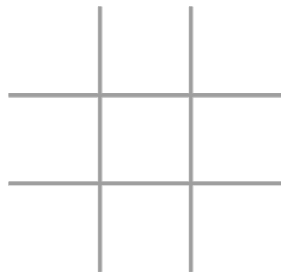
1. Nodes are game states
2. Edges are actions/moves

We can work down a tree until we reach one or some **terminal states**

The full tree is often **very expensive**

1. In tic-tac-toe, there are about 300,000 leaves.
2. In chess, there are about  $10^{120}$  leaves. For reference: there are estimate around  $10^{80}$  atoms in the universe, or  $10^{17}$  seconds since the big bang.

So we instead implement a *search tree*: finding a good move despite an intractable complete tree.

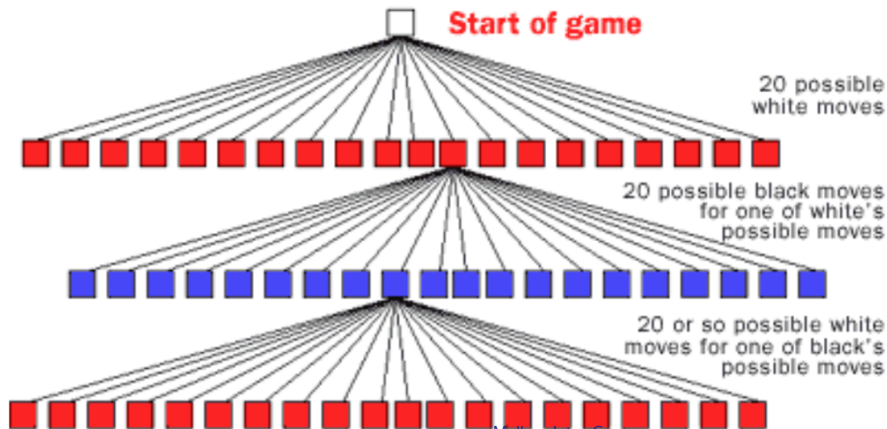


## Minimax

*need terminal state / fixed depth*

The **minimax** algorithm is a backtracking algorithm used in decision-making and game theory.

It works in turn-based games: Tic-Tac-Toe, Chess, Backgammon, Mancala, etc.



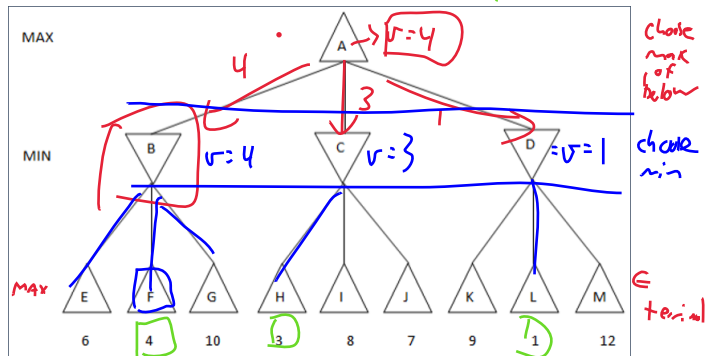


# Minimax

1. We have two players, MAX and MIN
2. Alternate making moves
3. Optimal strategy from state  $s$  is determined by the **minimax value** of that state.

The idea: we're MAX, trying to *maximize*. But that MIN character is trying to *minimize*, so we have to make sure they don't get in our way!

if MAX chooses B  
 min can choose F or G.  
 MIN will choose minimum, F:  $v=4$

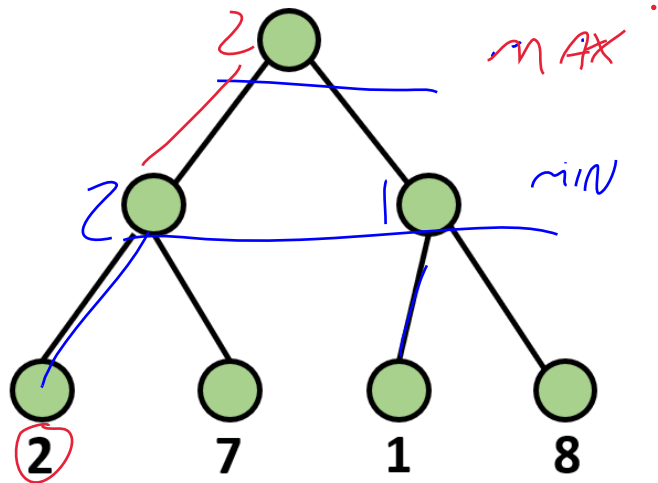


## Minimax Worked

Consider the Game Tree to the right.

Find the resulting value at the root node by following the minimax algorithm.

The value of this node comes from making a set of decisions: we can act to move left or right. After we do, the MIN player will choose the *minimum* value on the bottom row. We want to *maximize* their worst-case choice for us!



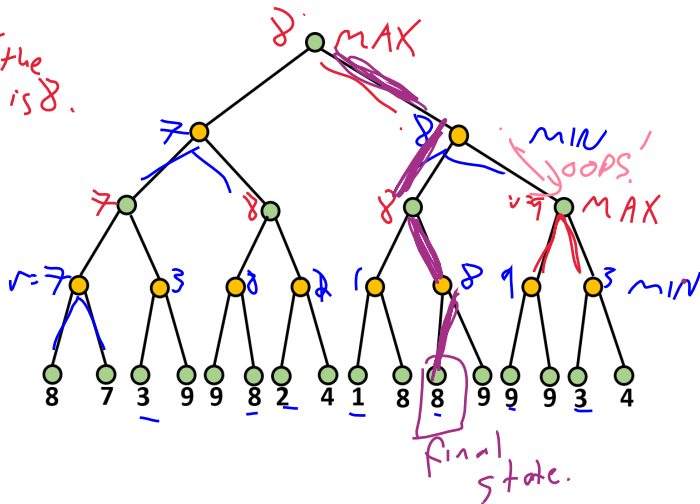
## Minimax Worked

: there is a sequence of moves, that mean the final (terminal) utility is 8.

Consider the Game Tree to the right.  
Find the resulting value at the root node by following the minimax algorithm.

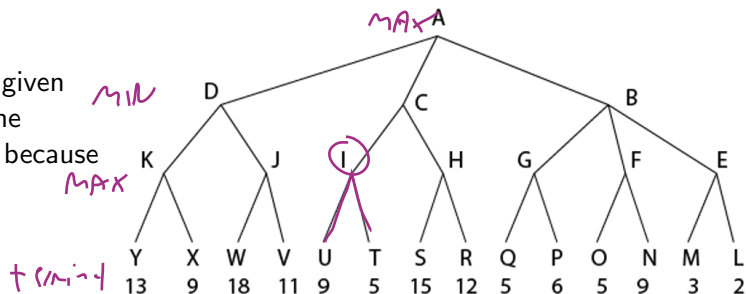
No way can  
MAX make that larger  
or MIN make it  
smaller

Solution



## Minimax More

The minimax decision in any given state  $s$  is the action that is the optimal choice for Max/Min, because it leads to the highest/lowest minimax value.



We define:

$$\text{minimax}(s) = \begin{cases} \text{utility}(s) & \text{if } s \text{ is terminal} \\ \max_{a \in \text{actions}(s)} & \text{if it is Max's turn} \\ \min_{a \in \text{actions}(s)} & \text{if it is Min's turn} \end{cases}$$

# Minimax Code

```

def minimax_decision(state):
    all_actions = what are the available actions?
    best_action = action that maximizes min_value(result(action, state))
    return best_action

def min_value(state):
    if terminal_state(state):
        return utility(state)
    value = infinity
    for action in all available actions:
        value = min(value, max_value(result(action, state)))
    return value

def max_value(state):
    if terminal_state(state):
        return utility(state)
    value = -infinity
    for action in all available actions:
        value = max(value, min_value(result(action, state)))
    return value

```

Handwritten annotations: "P2" next to the `min_value` function, and "P1" next to the `max_value` function. Blue circles highlight `return utility(state)` and `value = infinity` in `min_value`, and `return utility(state)` and `value = -infinity` in `max_value`. Blue boxes highlight `min` and `max_value` in the loop of `min_value`. Pink underlines highlight the `return` statements in `minimax_decision` and `max_value`.

# Minimax Code

```
function MINIMAX-DECISION(state) returns an action
  return  $\arg \max_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(\text{state}, a))$ 
```

---

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

---

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

**Figure 5.3** An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation  $\arg \max_{a \in S} f(a)$  computes the element  $a$  of set  $S$  that has the maximum value of  $f(a)$ .

## Minimax Complexity

The number of games states is exponential in the number of moves at any given moment: this set of actions will serve as a *branching factor* for the state tree.

So we **don't examine every node**.

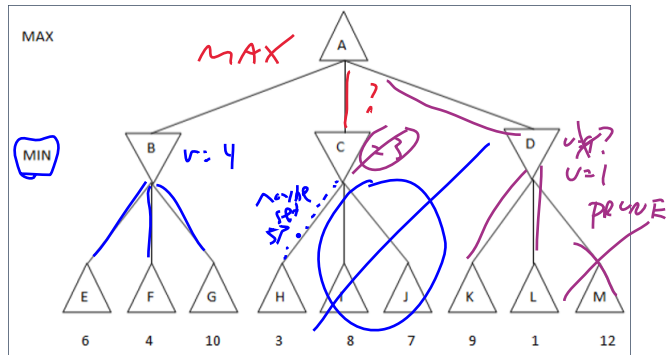
The **Alpha-Beta** pruning algorithm take similar ideas to e.g. Uniform-Cost expansion and applies it to the minimax problem.

1. Remove branches that do not influence our final decision.
2. Idea; you can bracket the best/worst (or highest/lowest) values at each node, even before all of its successors have been evaluated.

To avoid doing a full search - which is  $O(b^m)$  with maximum depth  $m$  and branching factor  $b$  - we know that under optimal play some branches can be discarded.

We call that *pruning* those branches.

2. On the tree to the right, the MIN value of  $B$  is 4. When we consider node  $C$ , we discover a candidate min of  $H = 3$  which makes  $C$  worse to move to (from  $A$ ) *regardless* of what's in  $I, J$ . We can prune those branches.



value of  $c \in \mathbb{Z}$



# Alpha-Beta Formalism

Using Alpha-Beta pruning in practice:

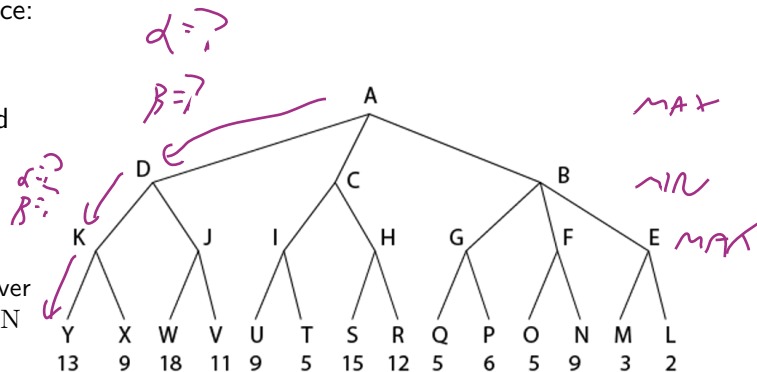
1. Start with  $\alpha = -\infty$ ,  $\beta = \infty$ .

These are the current best and worst case estimates for the problem as a whole.

2. Prune if:

2.1 While exploring a MIN,  $v \leq \alpha$ . This means we'd never choose to move to that MIN

2.2 While exploring a MAX,  $v \geq \beta$ . This means MIN would never choose to give us that option.



# Alpha-Beta Formalism

```
def alphabeta_search(state):
    alpha = -infinity
    beta = +infinity
    value = max_value(state, alpha, beta)
    best_action = action that has utility=value to Max
    return

def max_value(state, alpha, beta):
    if terminal_state(state):
        return utility(state)
    value = -infinity
    for action in all available actions:
        value = max(value, min_value(result(action, state), alpha, beta))
        if value >= beta: return value
        alpha = max(value, alpha)
    return value

def min_value(state, alpha, beta):
    if terminal_state(state):
        return utility(state)
    value = +infinity
    for action in all available actions:
        value = min(value, max_value(result(action, state), alpha, beta))
        if value <= alpha: return value
        beta = min(value, beta)
    return value
```

# Alpha-Beta Formalism

**function** ALPHA-BETA-SEARCH(*state*) **returns** an action  
 $v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$   
**return** the *action* in  $\text{ACTIONS}(\text{state})$  with value  $v$

---

**function** MAX-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** a utility value  
**if**  $\text{TERMINAL-TEST}(\text{state})$  **then return**  $\text{UTILITY}(\text{state})$   
 $v \leftarrow -\infty$   
**for each**  $a$  **in**  $\text{ACTIONS}(\text{state})$  **do**  
 $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$   
**if**  $v \geq \beta$  **then return**  $v$   
 $\alpha \leftarrow \text{MAX}(\alpha, v)$   
**return**  $v$

---

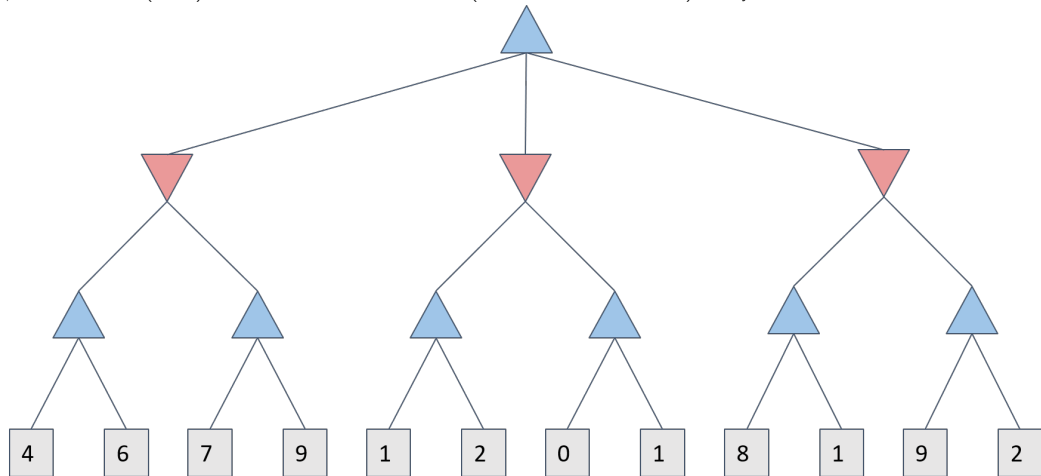
**function** MIN-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** a utility value  
**if**  $\text{TERMINAL-TEST}(\text{state})$  **then return**  $\text{UTILITY}(\text{state})$   
 $v \leftarrow +\infty$   
**for each**  $a$  **in**  $\text{ACTIONS}(\text{state})$  **do**  
 $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$   
**if**  $v \leq \alpha$  **then return**  $v$   
 $\beta \leftarrow \text{MIN}(\beta, v)$   
**return**  $v$

---

**Figure 5.7** The alpha-beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain  $\alpha$  and  $\beta$  (and the bookkeeping to pass these parameters along).

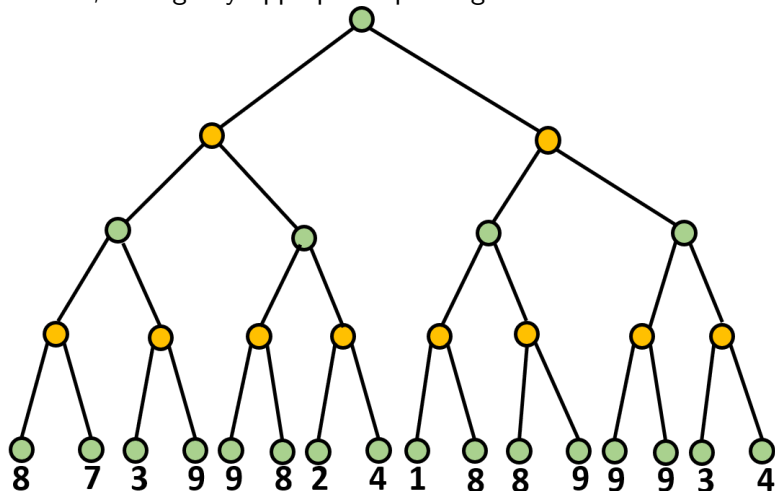
$\alpha$  := value of best (highest) choice for MAX we have found so far (over all actions from the root). Initially  $-\infty$ .

$\beta$  := value of best (lowest) choice for MIN we have found so far (over all actions from the root) Initially  $\infty$ .



## Another Example

Consider the game tree below. Find the resulting value of the root node by following the minimax, noting any appropriate pruning actions.



## AB Complexity

What's the payoff of the Alpha-beta pruning?

1. Generally doesn't affect final results.
2. Entire subtrees can be pruned early on: not just leaves.
3. On typical problems, can look twice as deep as minimax in the same amount of time.
4. Both algorithms still have exponential complexity in theory.
5. ... what do we do if we can never reach a true goal state? Those values might be *utility* values or even heuristics!

# Moving Forward

- ▶ This Week:

- 1. nb day Friday

- ▶ Next time: Decision-Making!