Adversarial Search

Dr. Demetrios Glinos
University of Central Florida

CAP4630 – Artificial Intelligence

Today

- Adversarial Games
- Search in Games
- Optimal Decisions in Games
- Alpha-Beta Pruning

Adversarial Games

- We can characterize adversarial games according to:
  - Number of teams:
    - 2-player games: chess, checkers, go
    - More than two: card games, online games, board games
  - Number of players on a team
    - Football: 11 at a time
    - Tennis: 1
  - Level of information
    - Complete information: chess, checkers, go
    - Partial information: poker, Kriegspiel chess
  - Stochastic component
    - None: chess, checkers, go
    - Some: poker, Pac-Man

Our Focus Today

- Two-person
- Deterministic
- Complete information
- Zero-sum
  - What one side wins, the other loses

    →One side maximizes, the other minimizes
  - herein of *payoffs* and *utilities*

Games as Search

- We can formalize games as follows:
  - State space S
  - initial state: S_0
  - Players: P = {1, ..., N}, usually take turns
  - Actions: A, the set of legal moves for a state
  - Transition function: S x A → S
  - Terminal Test: S → { true, false }
  - Terminal Utilities: S x P → R

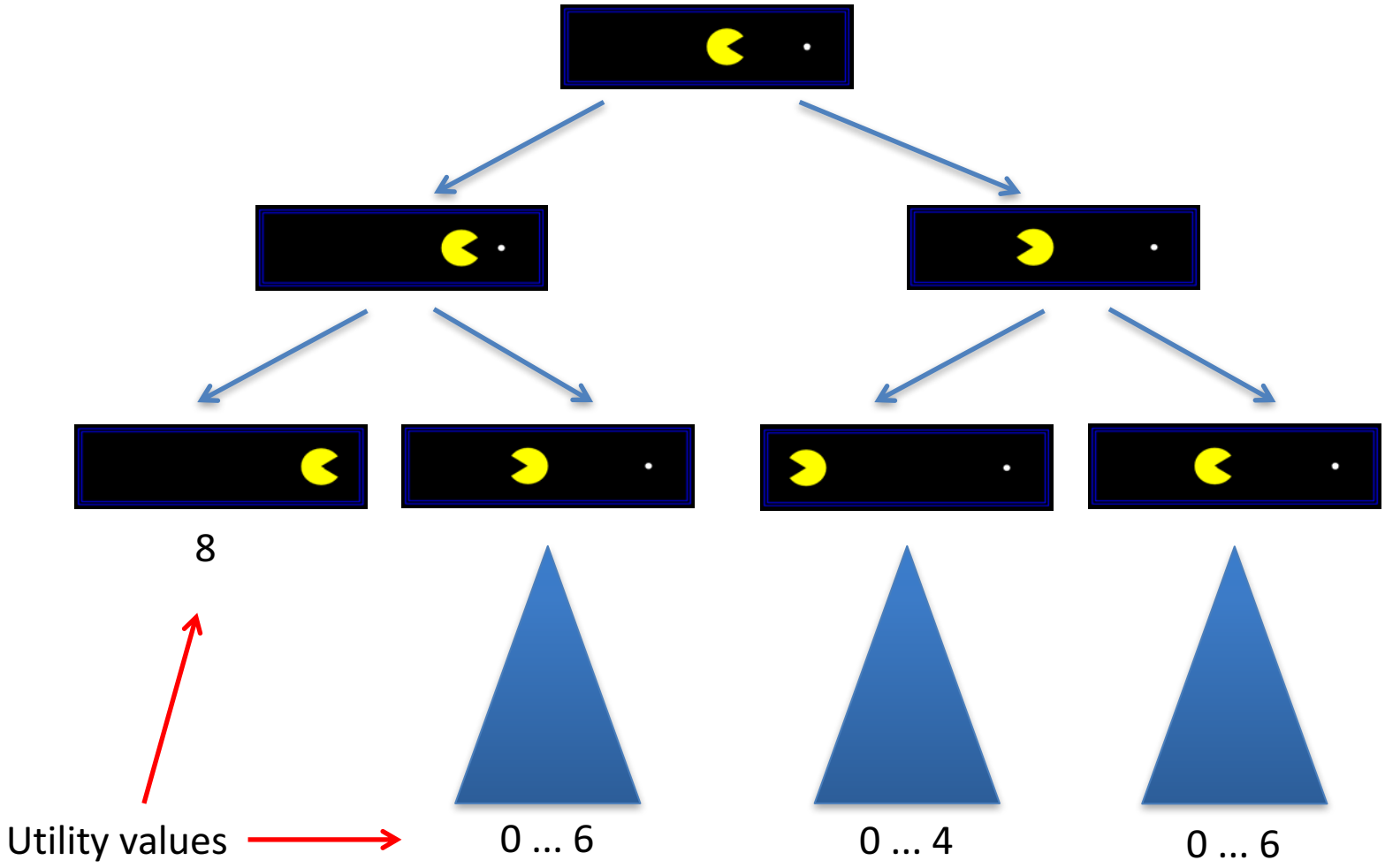
The 1996 match

Game #	White	Black	Result	Comment
1	**Deep Blue**	Kasparov	1–0	
2	**Kasparov**	Deep Blue	1–0	
3	Deep Blue	Kasparov	½–½	Draw by mutual agreement
4	Kasparov	Deep Blue	½–½	Draw by mutual agreement
5	Deep Blue	**Kasparov**	0–1	Kasparov offered a draw after the 23rd move.
6	**Kasparov**	Deep Blue	1–0	
Result: **Kasparov–Deep Blue: 4–2**				

The 1997 rematch

Game #	White	Black	Result	Comment
1	**Kasparov**	Deep Blue	1–0	
2	**Deep Blue**	Kasparov	1–0	
3	Kasparov	Deep Blue	½–½	Draw by mutual agreement
4	Deep Blue	Kasparov	½–½	Draw by mutual agreement
5	Kasparov	Deep Blue	½–½	Draw by mutual agreement
6	**Deep Blue**	Kasparov	1–0	
Result: **Deep Blue–Kasparov: 3½–2½**				

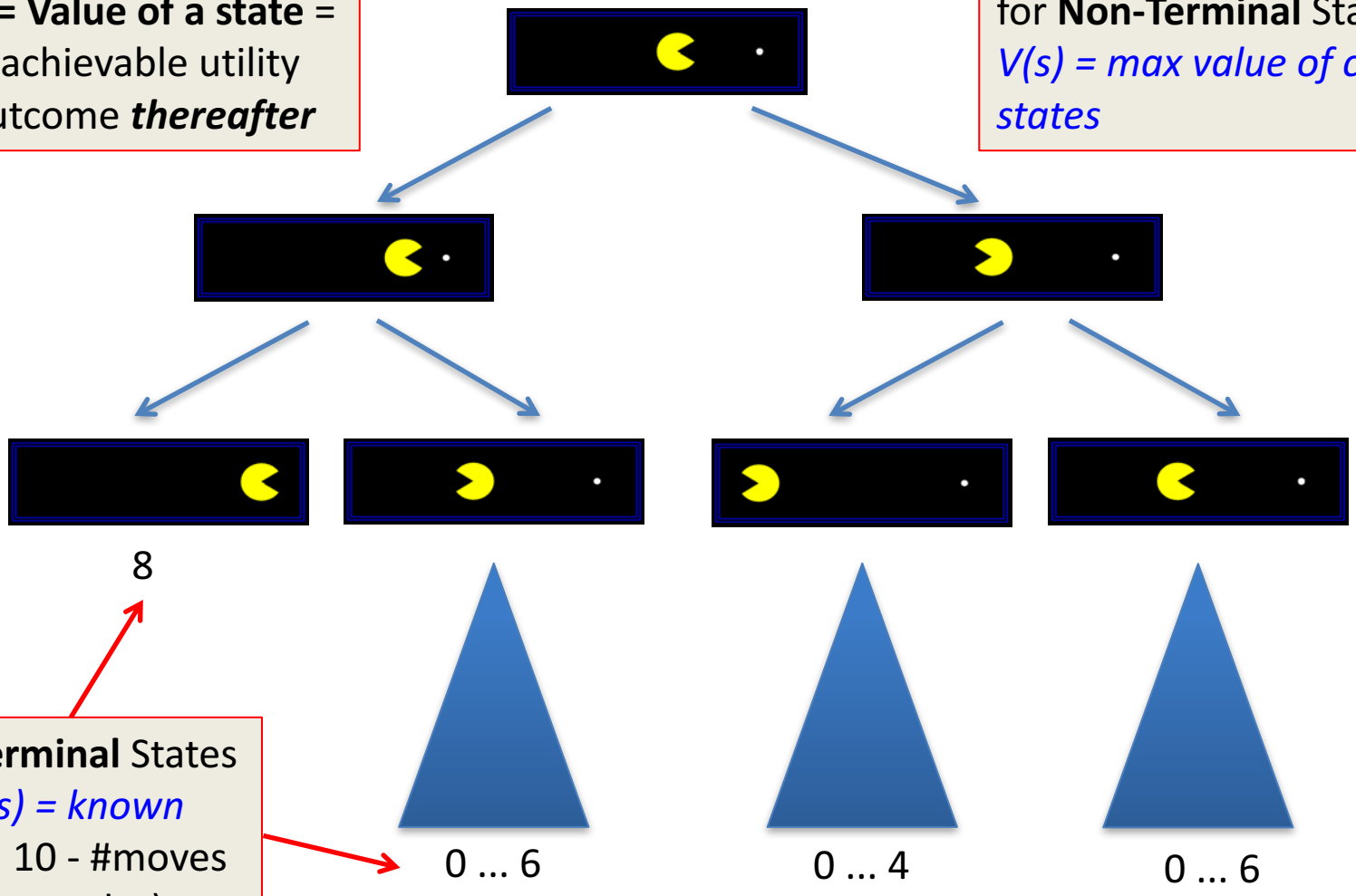
Single-Agent Search Tree



Value of a State

V(s) = Value of a state = best achievable utility or outcome ***thereafter***

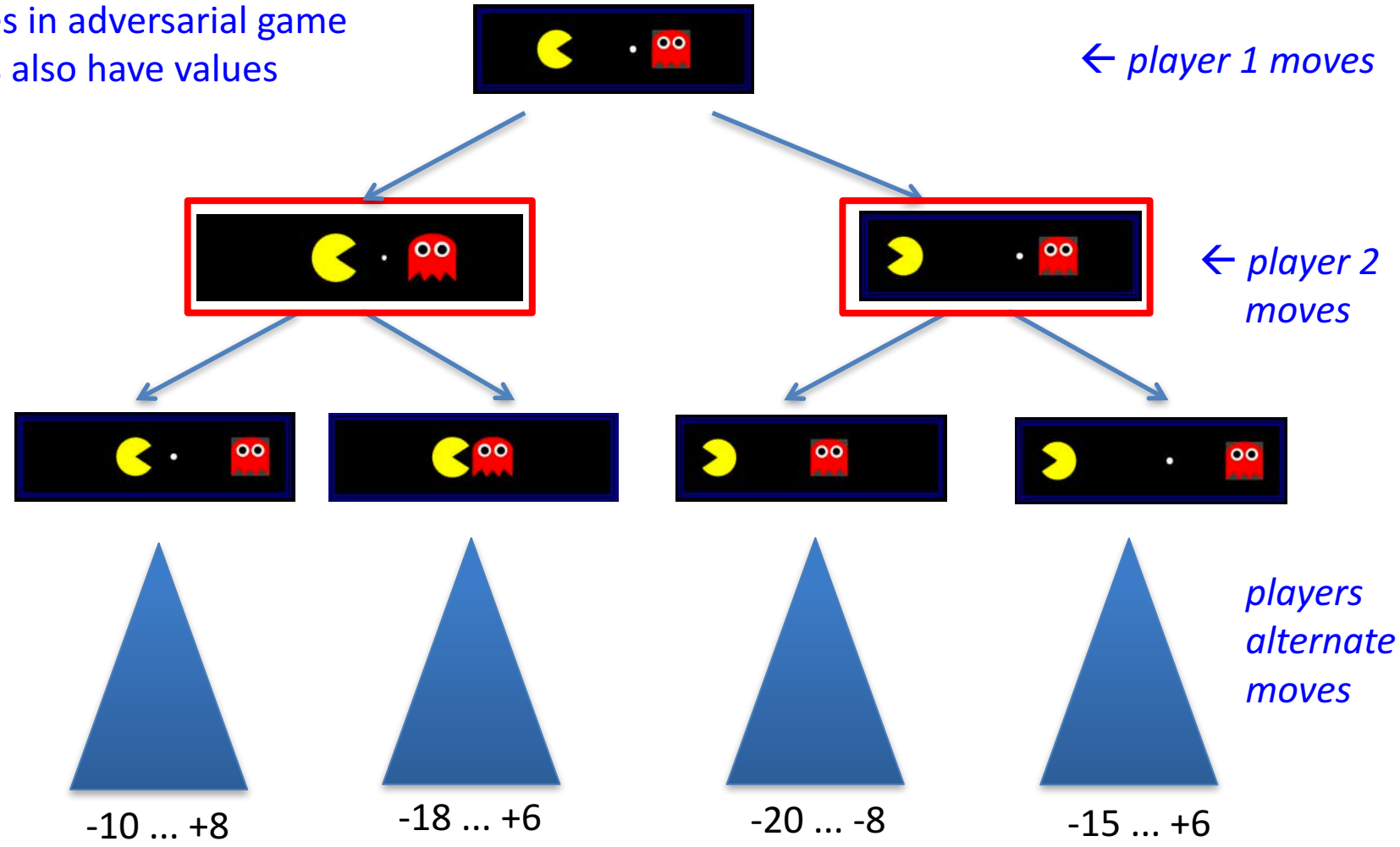
for **Non-Terminal** States *V(s) = max value of child states*



for **Terminal** States *V(s) = known* (e.g., 10 - #moves to eat dot)

Adversarial Game Trees

States in adversarial game trees also have values



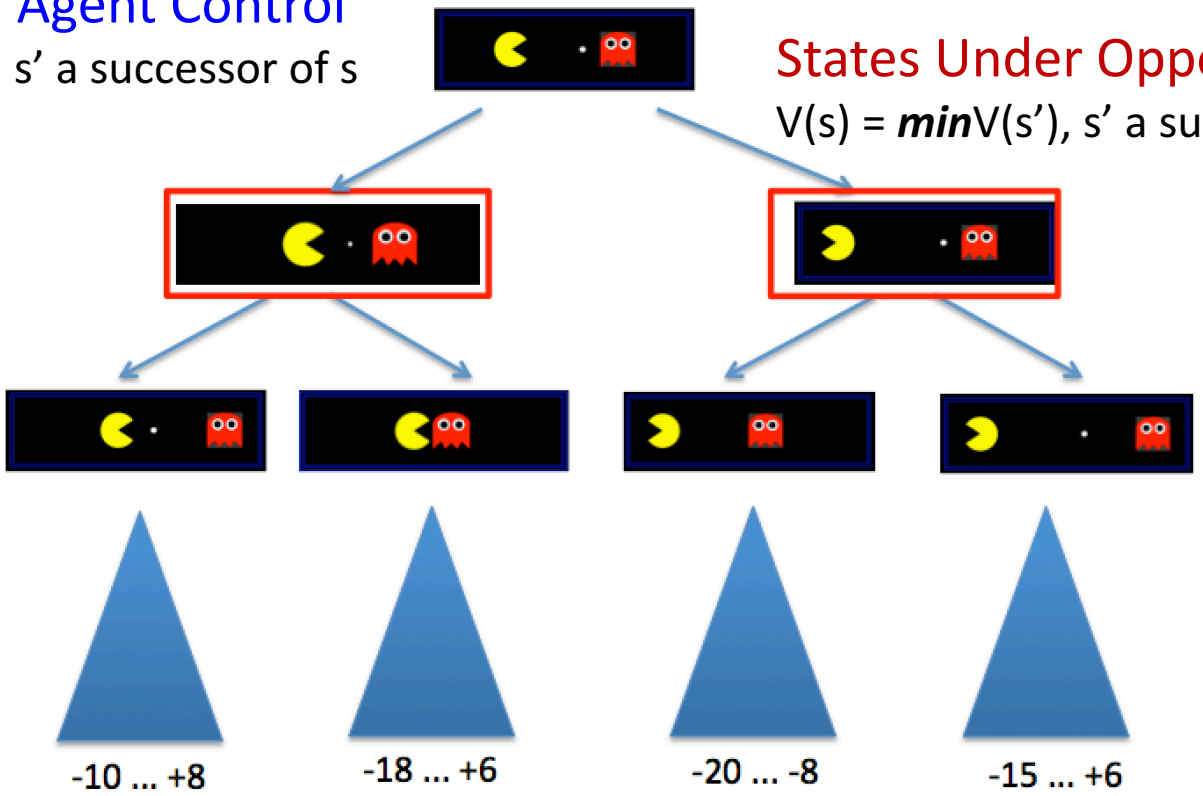
Minimax Values

States Under Agent Control

V(s) = ***max***V(s'), s' a successor of s

States Under Opponent Control

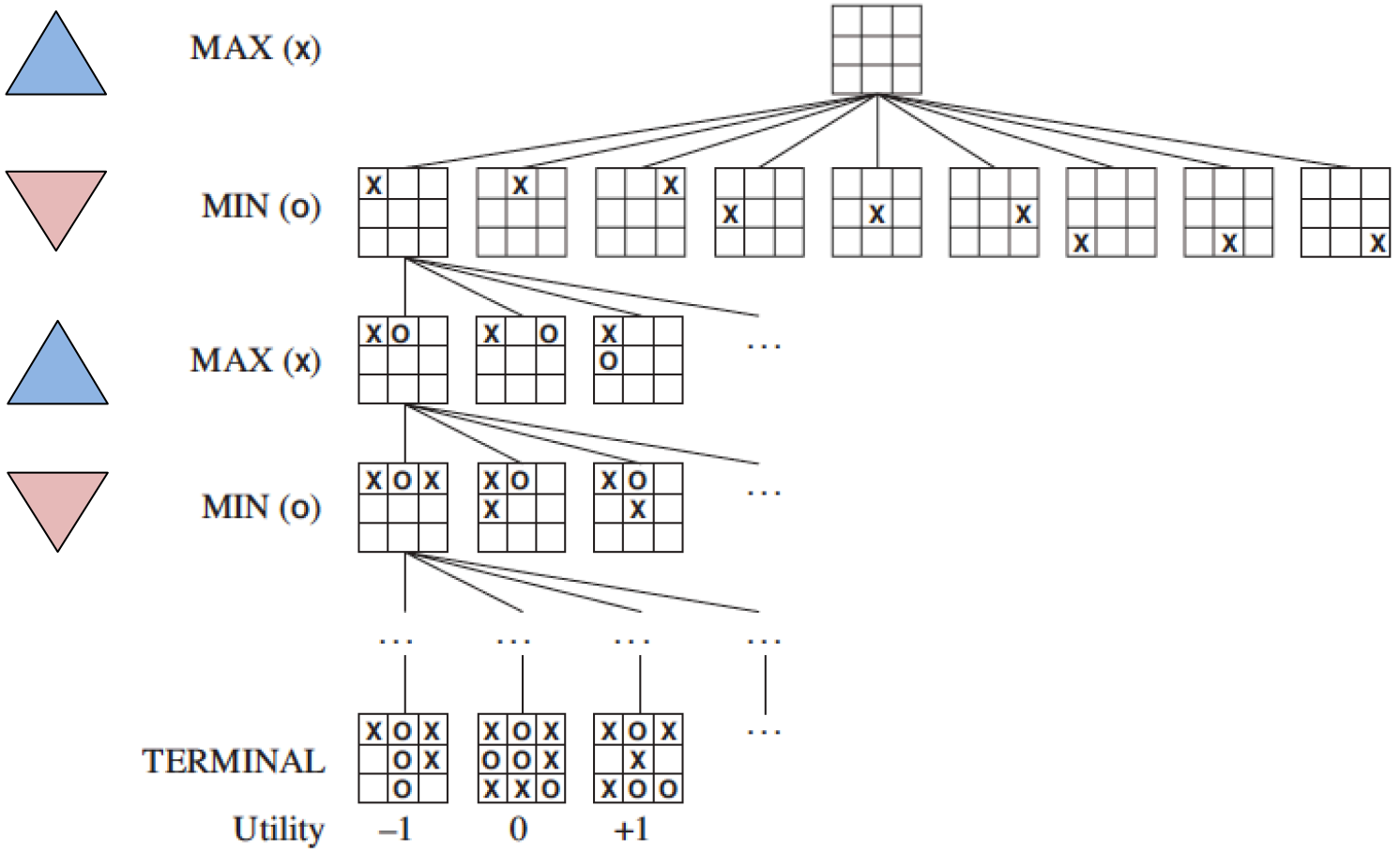
V(s) = ***min***V(s'), s' a successor of s



Terminal States

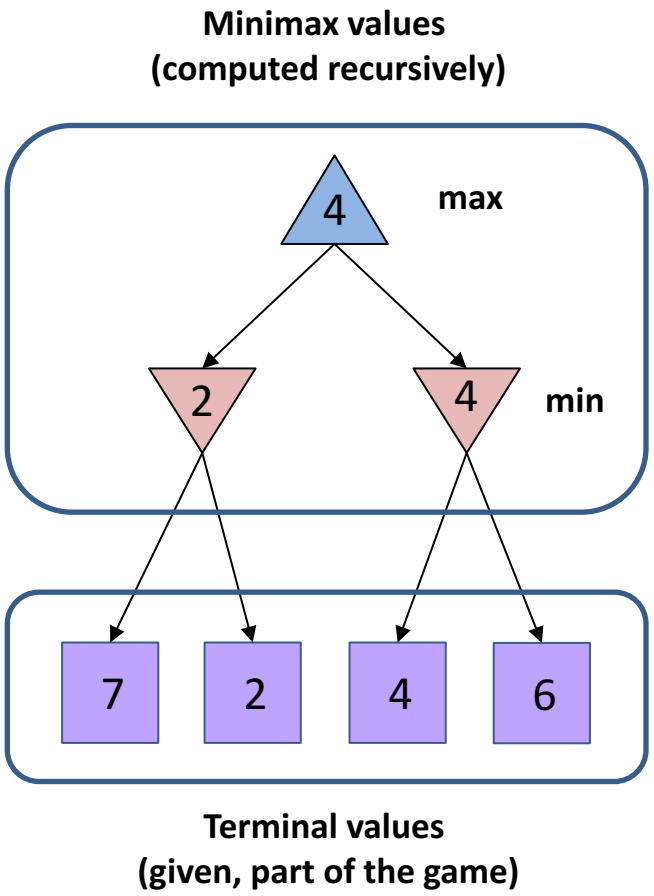
V(s) = known

Tic-Tac-Toe Game Tree



Minimax Search

- For solving certain adversarial search problems
  - Deterministic
  - Zero-sum
  - One player maximizes value
  - The other player minimizes value
- **Minimax search**
  - a state-space search tree
  - players alternate turns
  - algorithm computes the minimax value for each nonterminal node
- The **minimax value** represents the best achievable utility against a rational (optimal) opponent



Minimax Algorithm

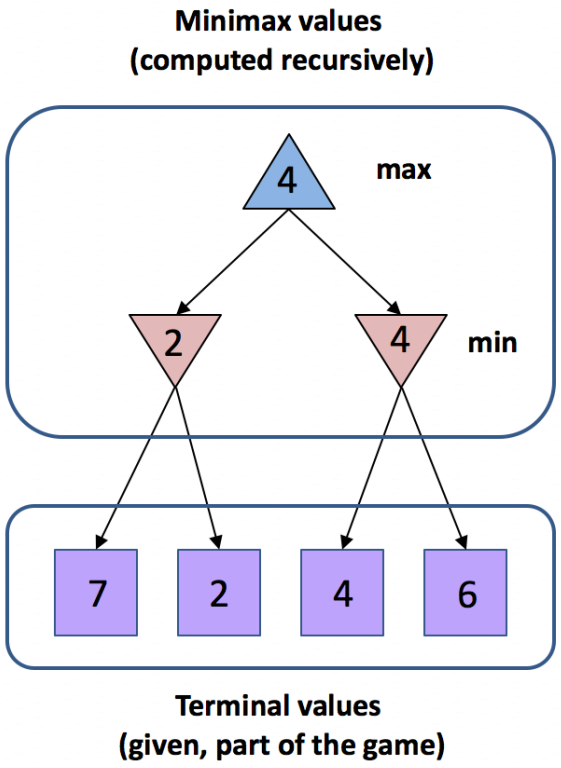
Value(state) =
if terminal state, then return the state's utility
else if it is MAX's turn to move, then return **Max-Value(state)**
else if it is MIN's turn to move, then return **Min-Value(state)**

Max-Value(state):

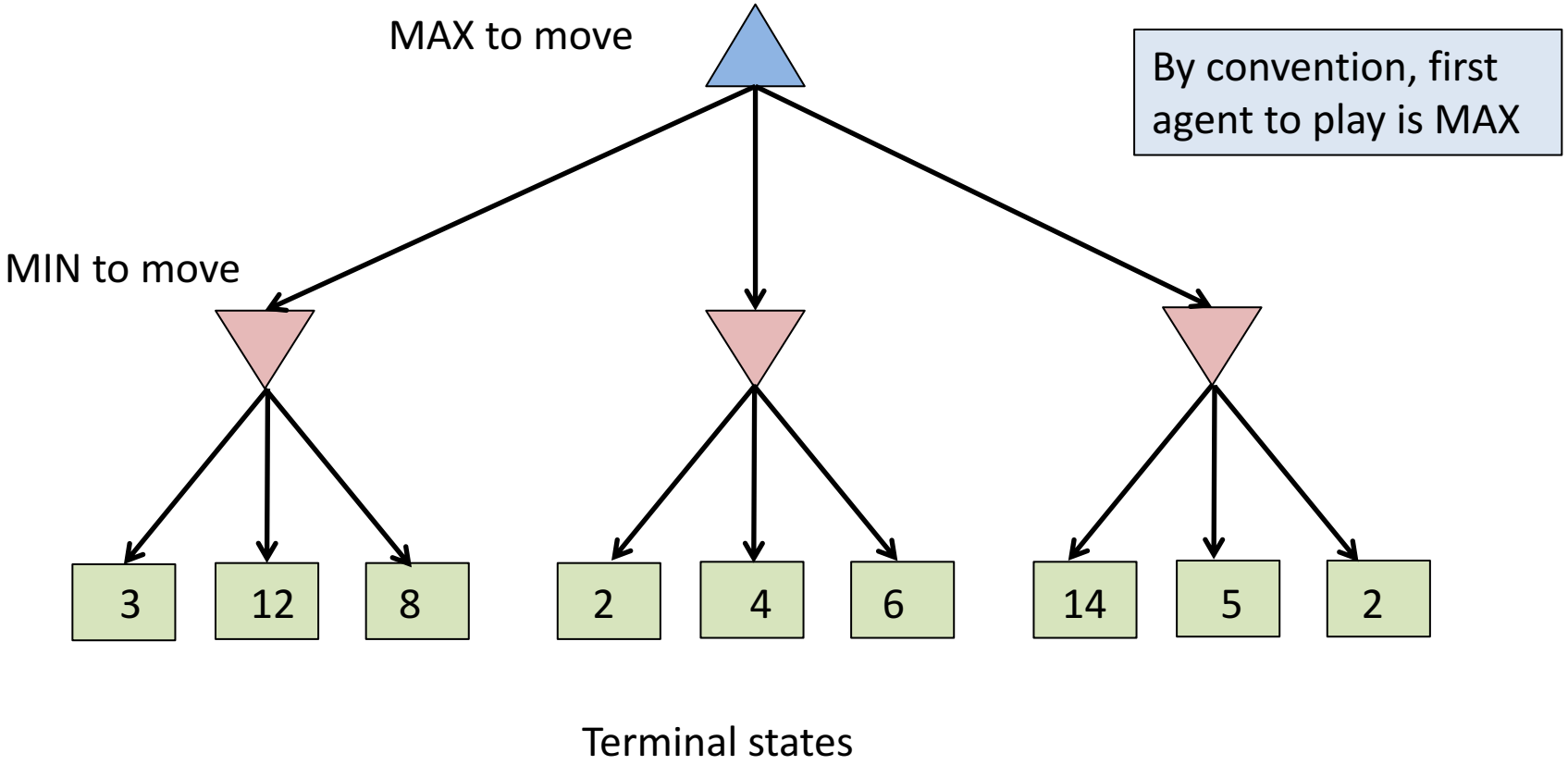
```
v ← - ∞
for each successor s' of s {
    v = max( v, value( s' ) )
}
return v
```

Min-Value(state):

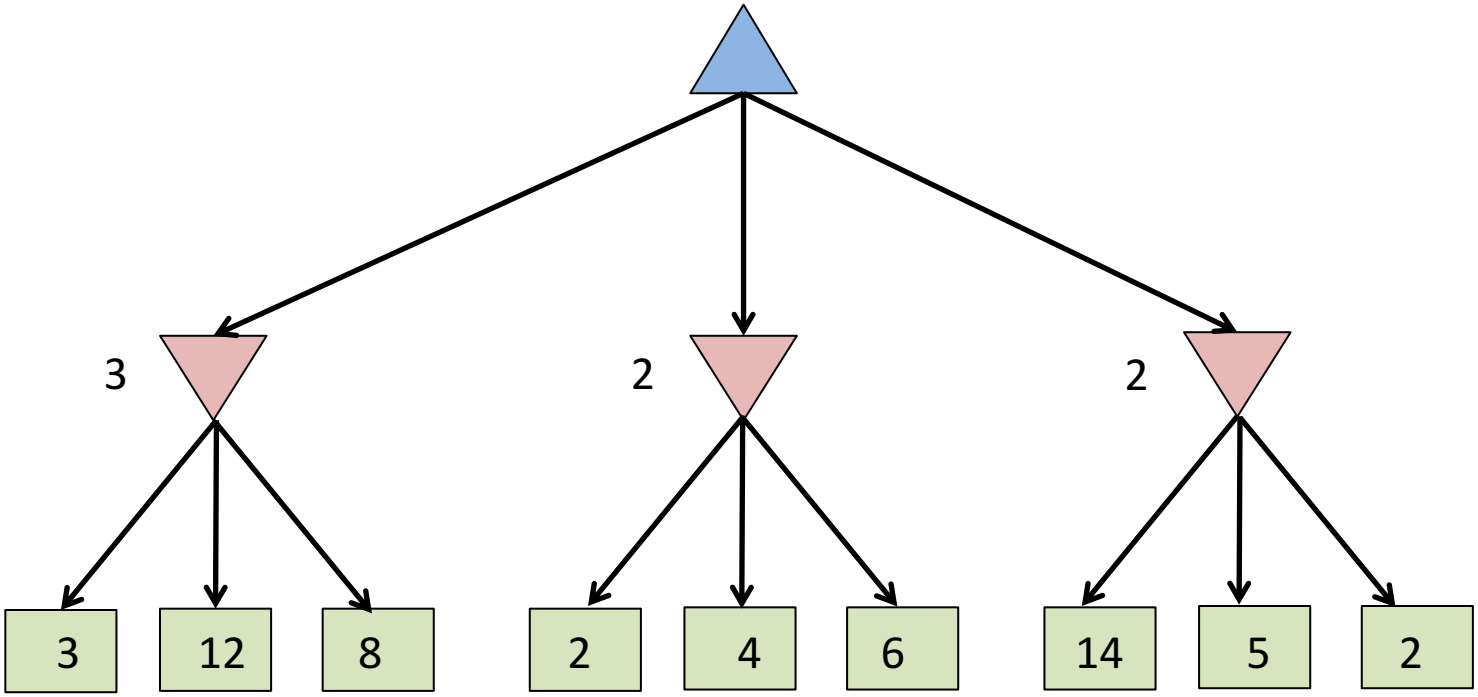
```
v ← + ∞
for each successor s' of s {
    v = min( v, value( s' ) )
}
return v
```



Minimax Example: Setup

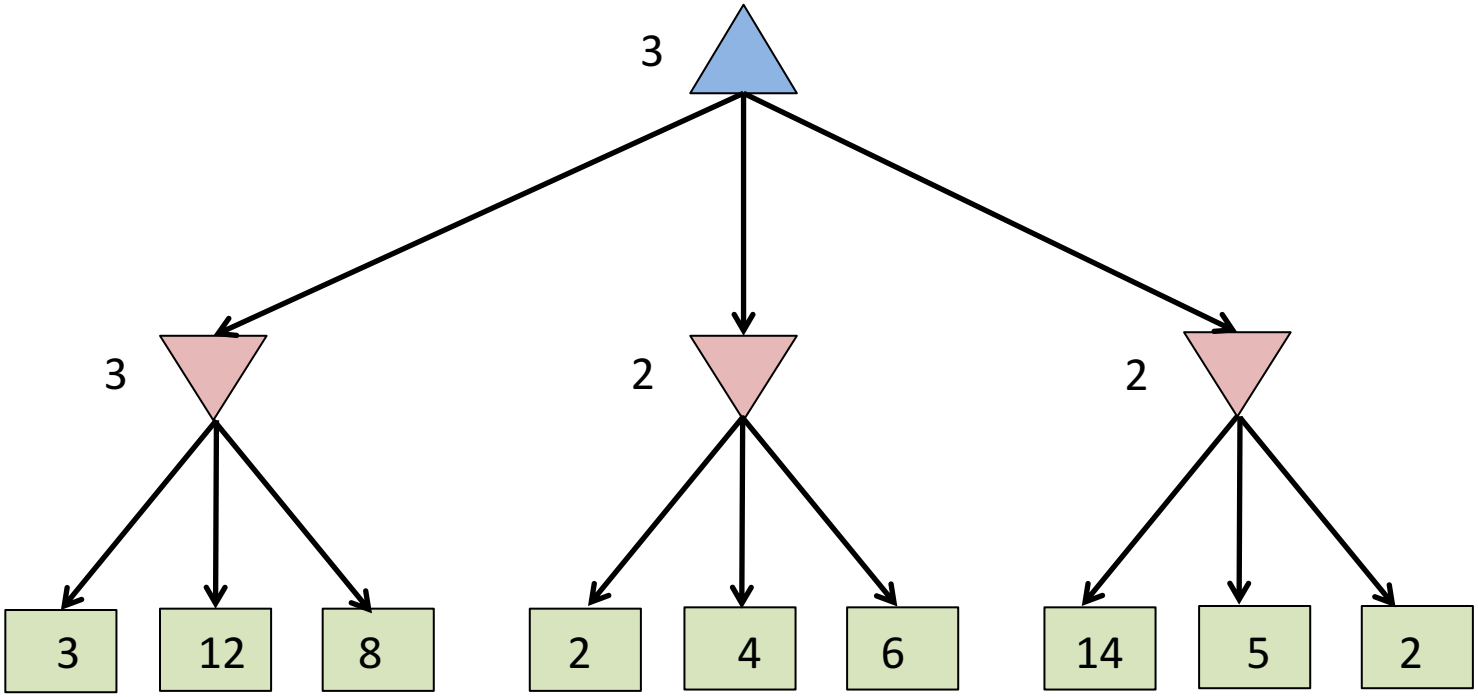


Minimax Example: After MIN



We compute minimax values recursively, from the bottom up

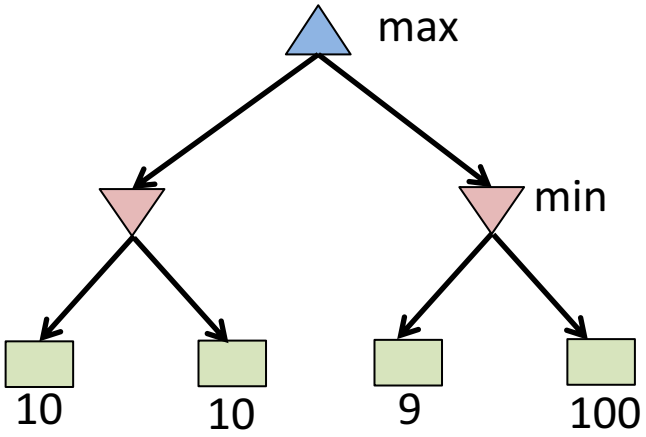
Minimax Example: After MAX



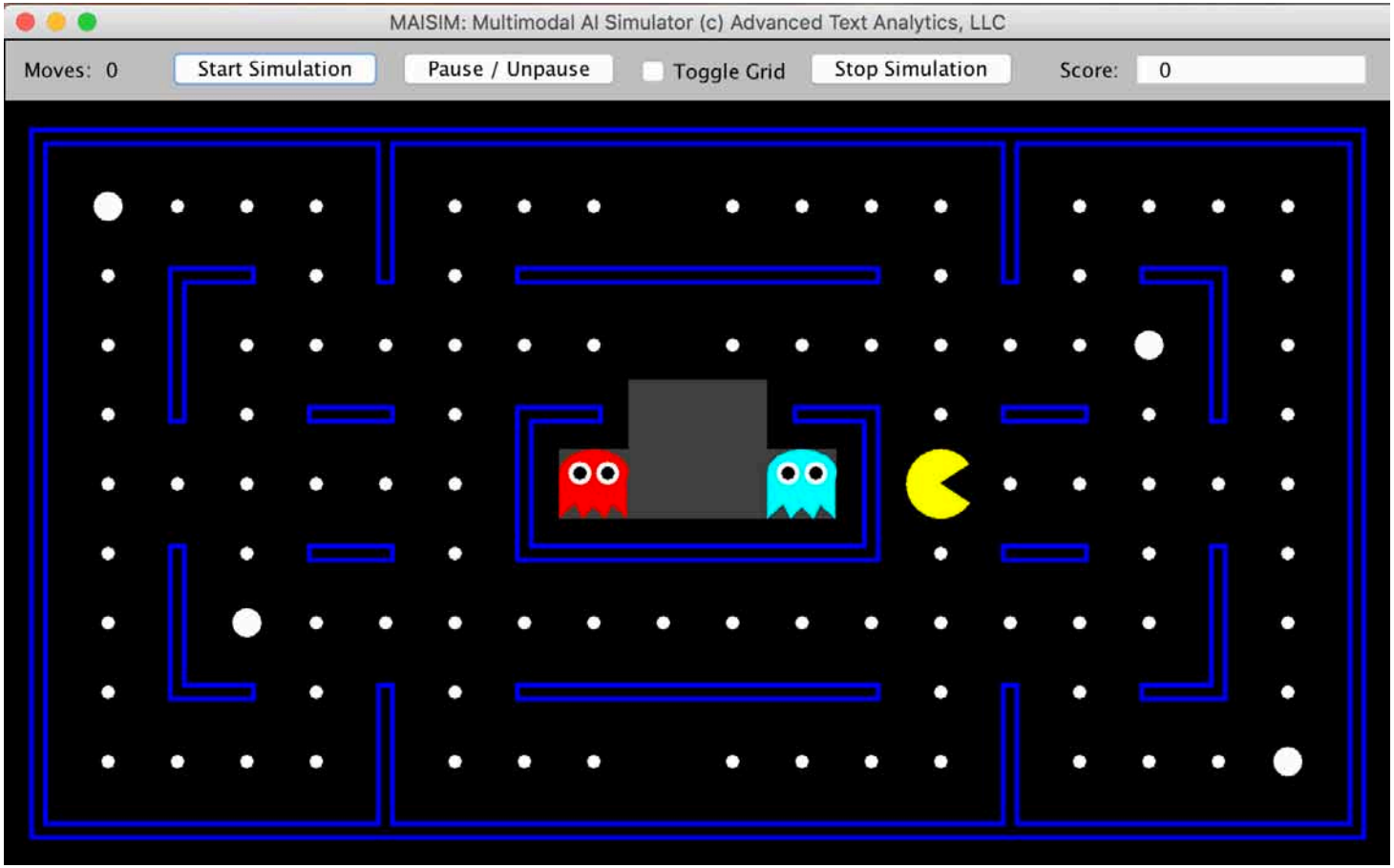
Interpretation: 3 is the best MAX can hope to achieve against an perfect (rational) opponent

Minimax Complexity

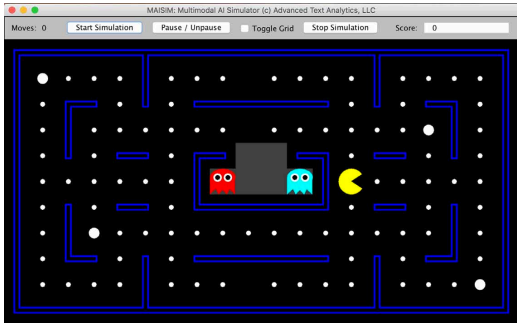
- Just like exhaustive DFS
  - Time: O(b^m)
  - Space: O(bm)
- What about chess?
  - b ≈ 35, m ≈ 65
  - Exact solution infeasible
- Questions:
  - Do we need to examine the entire tree?
  - What about playing a non-optimal opponent?



Pac-Man Minimax



Pac-Man Minimax



- Pac-Man
  - uses minimax with depth = 2
  - evaluation function avoids ghosts and heads for food
  - does not actively seek out power pellets or ghosts
- Ghosts
  - **Blinky** – heads for Pac-Man's current location
  - **Inky** – draws line from Blinky past Pac-Man by an equal amount and heads there
- Ghost behavior
  - scatter for 7 moves, chase for 20 moves
  - when scattering, Blinky heads for upper right corner, Inky to lower right
  - when Pac-Man eats a power pellet, they move randomly for 20 moves

demo: minimax

Smart and Random Opponents

- Against smart opponents, replan Pac-Man has no chance

  demo: trapsmart

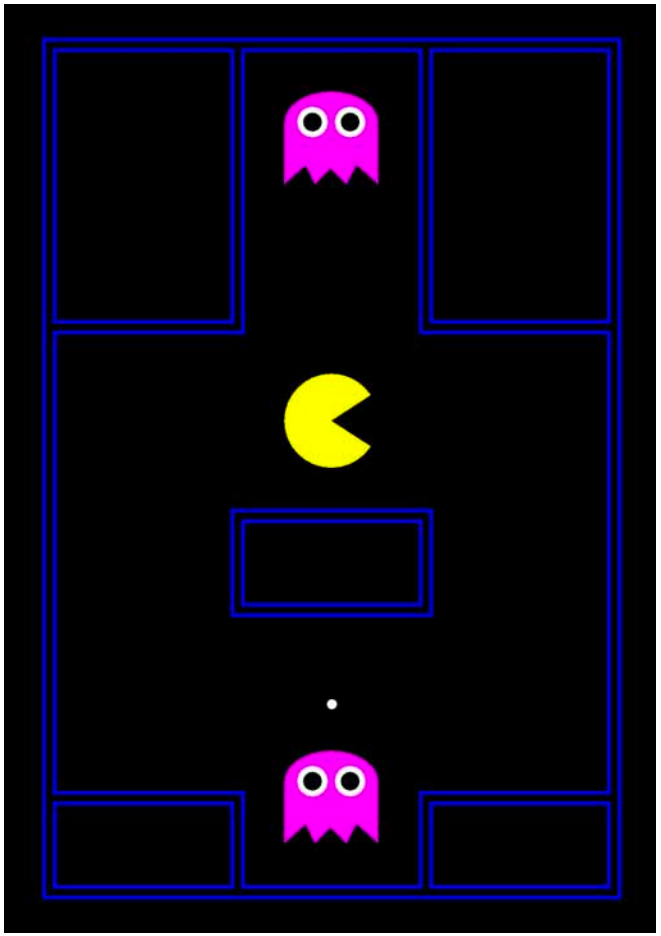
- Against random or suboptimal opponents, replan Pac-Man can sometimes win

  demo: traprandom

- Minimax pacman can win against smart opponents (but it depends on the game)

  demo: minimax2 (minimax depth 2)

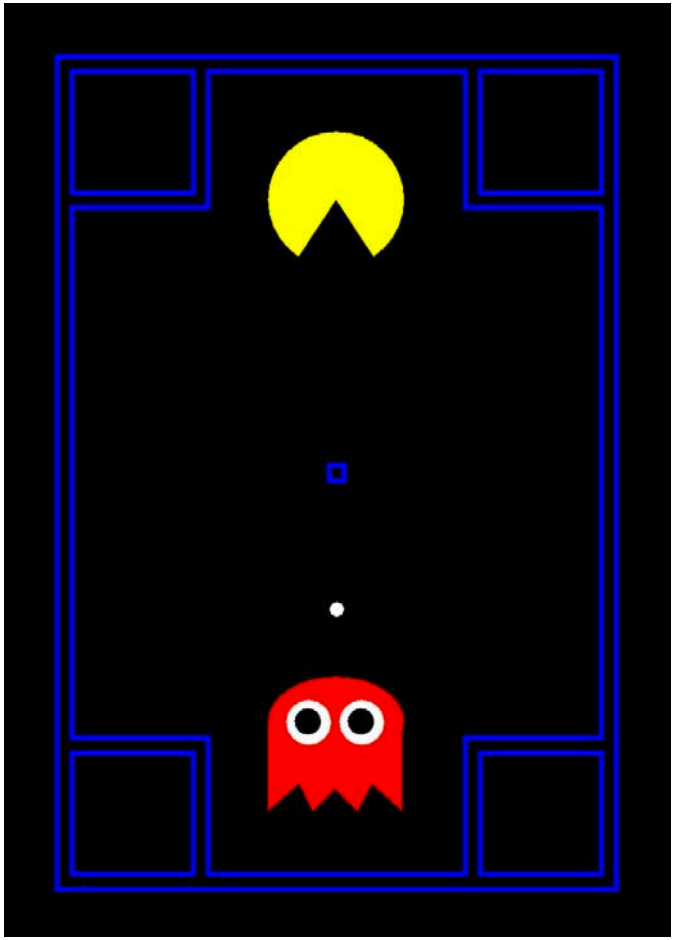
  minimax1 (minimax depth 1)



Winnable Games

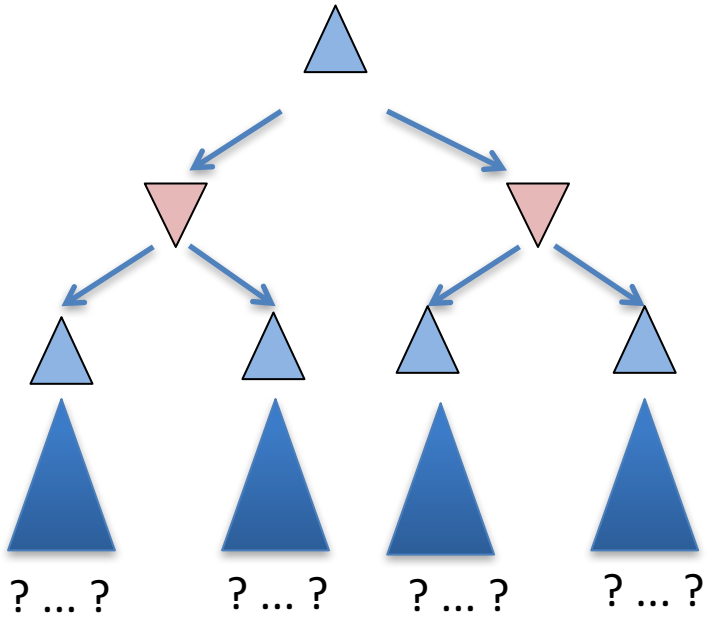
- Some games are not winnable against an optimal opponent
  - e.g., tic-tac-toe for 'O'
- This one is winnable against Blinky, but not against optimal opponent

demo: trapsmall



Resource Limitations

- Problem:
  - Basic Minimax searches entire tree
  - Infeasible in realistic games
- Solution: Depth-Limited Search
  - search only to limited depth
  - ***estimate*** values beyond using an evaluation function (heuristic)
  - Optimality no longer guaranteed
- Tree-pruning
  - Allows us to search deeper within same resource limits
  - Example: $\alpha - \beta$
- If not time-limited, could use IDS

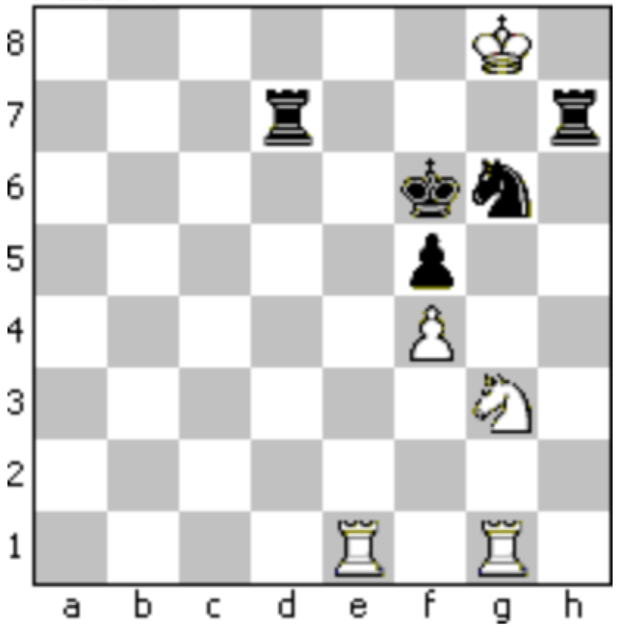


Example: Suppose have 100 sec to make move and can examine 10K nodes/sec

- can examine about 1M nodes/move
- $\alpha - \beta$ reaches about depth 8
- this would be a decent chess program

Depth Matters

- Explicity searching more deeply ***always*** helps
  - Evaluation functions are always inaccurate
  - Inaccuracies in evaluation function hurt less when buried deeper in tree
- Example:
  - A chess program that can look ahead 8 moves explicitly before it uses evaluation function can find 8-move combinations
- Resource trade-off:
  - complexity of evaluation function
  - depth of search

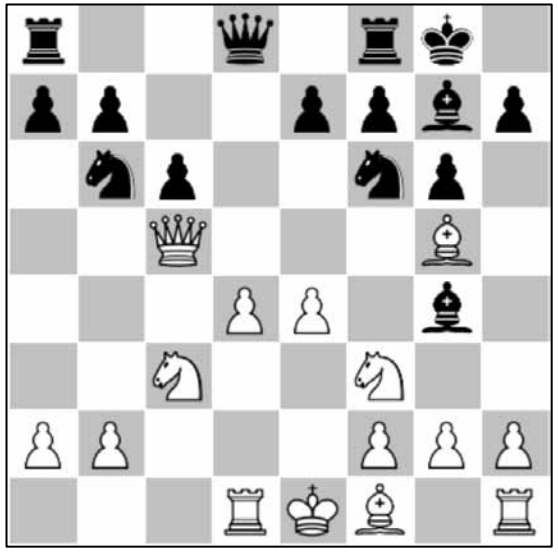


Q: Can you find the 3-move mate for White?

Note: Black has mate-on-the move

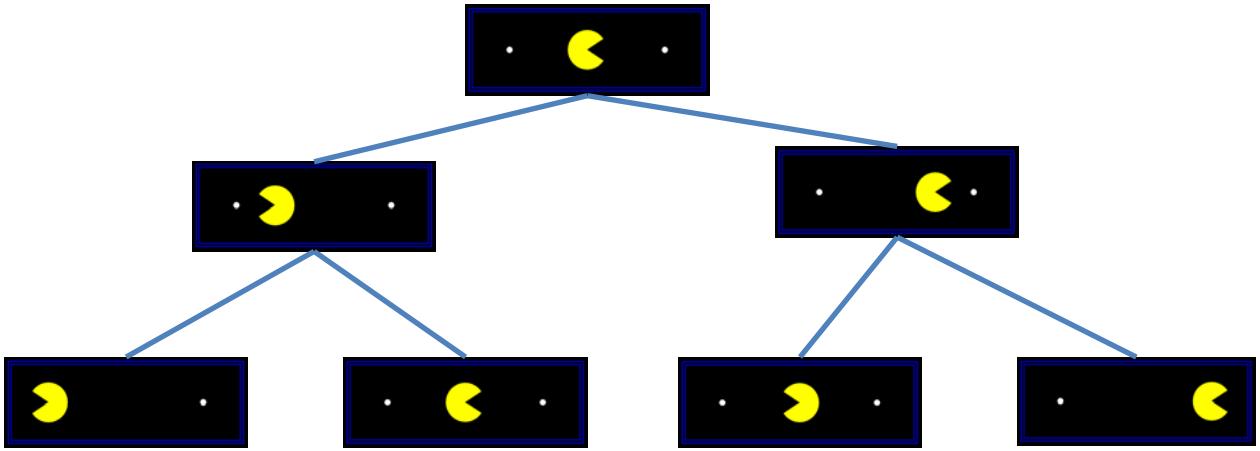
Evaluation Functions

- Generally problem-specific
- Usually expressed as a **linear combination** of features
- Each feature is a function of the state
  - For chess:
    - value of pieces remaining
    - control of center
    - scope of major pieces
    - passed pawns
    - connected rooks
    - …



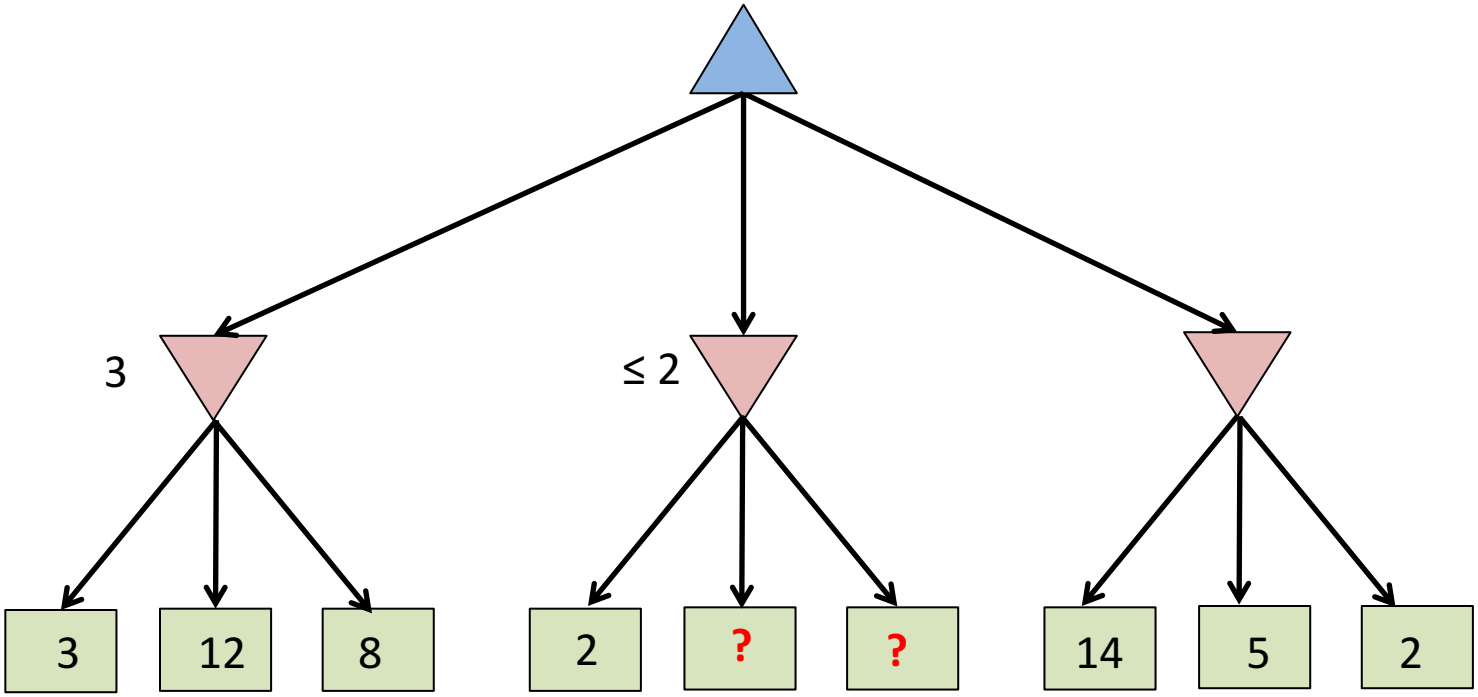
$$Eval(s) = w_1 * f_1(s) + w_2 * f_2(s) + \ldots + w_n * f_n(s)$$

Minimax and Replanning



- **Minimax is applied at every turn**
  - **so, a minimax agent is a replanning agent**
- **Thrashing** (oscillating between 2 states) is possible
- The problem is usually the evaluation function
  - If Eval(s) doesn't care if the food dot is eaten or how many moves until it is eaten, then why should Pac-Man?

Game Tree Pruning

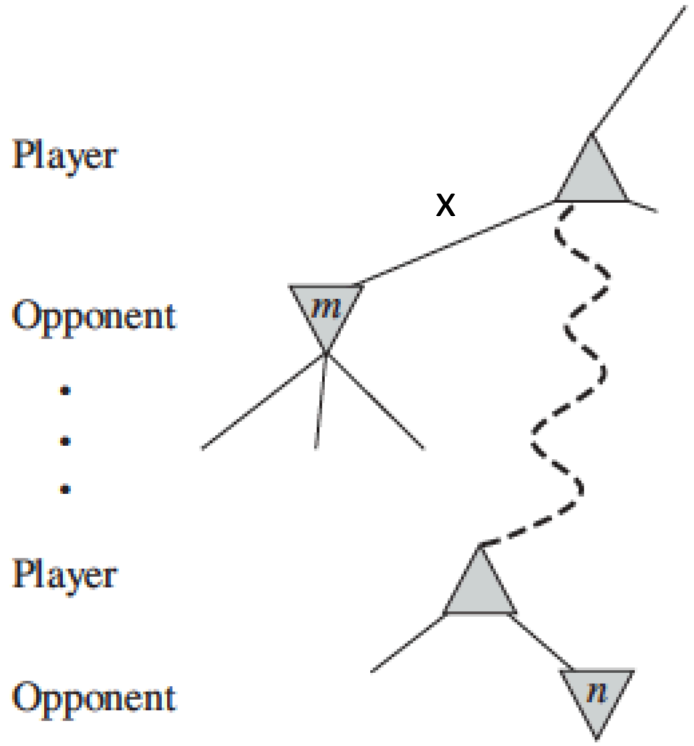


With MAX to move: once the "2" is seen at center, MAX will never choose that path, so why should MAX examine the other child nodes of that MIN node?

Remember: MAX calculates these minimax values *before* MAX makes its move

Alpha-Beta Pruning

- Basic idea:
  - **If a better option is already known, the current option will never be selected**
- Suppose we're computing the MIN value at some node n
  - so, we're looping over n's child nodes
  - n's estimate drops as we see values
- Who cares about n's value? MAX
- Let x be the best value that MAX can get at any choice point ***along the current path from the root***
- If the value at node n ever becomes lower than x, then MAX will avoid it, so we can stop considering n's other children



source: Fig. 5.6

General case: If value x from m is better for player than value from n, then the player will never let the game get to n

Alpha-Beta Algorithm

- We insert logic into our minimax algorithm to stop evaluating a node when the ***opponent*** has a better choice (and so will avoid this path)

α = value of MAX's best choice so far on path to root

β = value of MIN's best choice so far on path to root

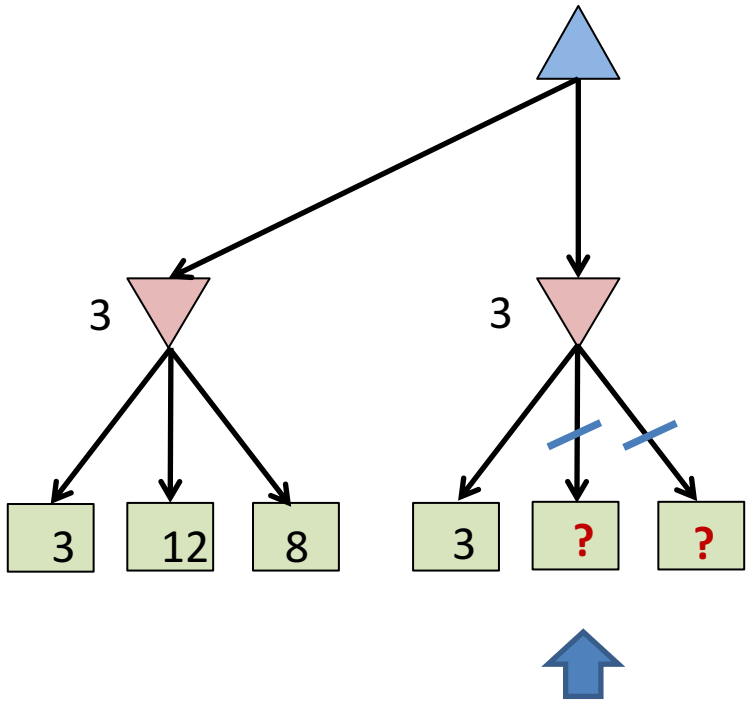
where we stop evaluating

```
Max-Value( state s, α, β ):
        v ← -∞
        for each successor s' of s {
            v = max( v, value( s', α, β ) )
            if v ≥ β return v
            α = max(α, v )
        }
        return v
```

```
Min-Value( state s, α, β ):
        v ← +∞
        for each successor s' of s {
            v = min( v, value( s', α, β ) )
            if v ≤ α return v
            β = min( β, v )
        }
        return v
```

Properties of Alpha-Beta Pruning

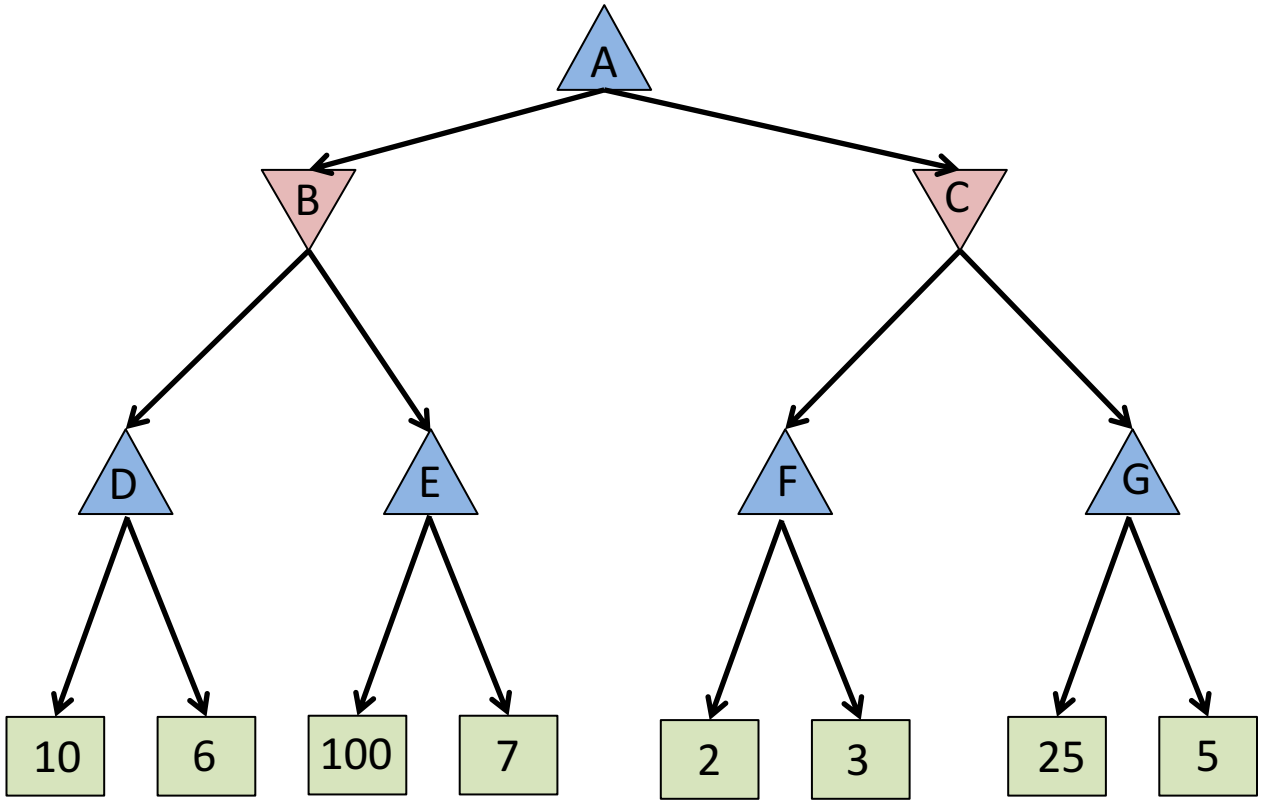
- No effect on minimax value computed for root
- ***But*** the intermediate node values may be wrong
  - can't use naive implementation for move selection
- Best case improvement
  - time complexity becomes $O(b^{m/2})$ with "perfect" child ordering
  - doubles feasible depth
  - but full search still generally infeasible for chess and other real-world problems



value of this MIN node could be lower, so not as good for MAX as the other node

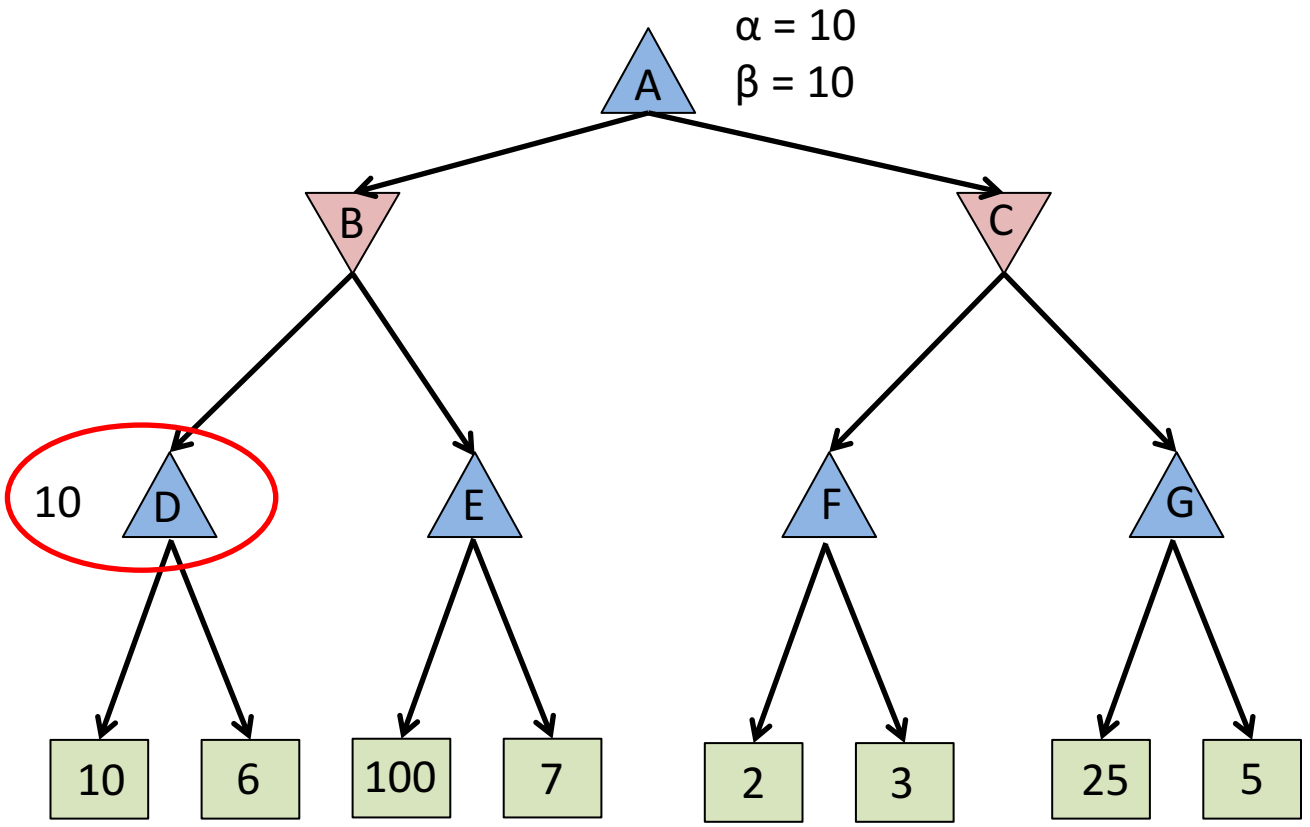
α-β Example

Q: Which paths are pruned? (Assume: leftmost recursion (i.e., DFS) from A)



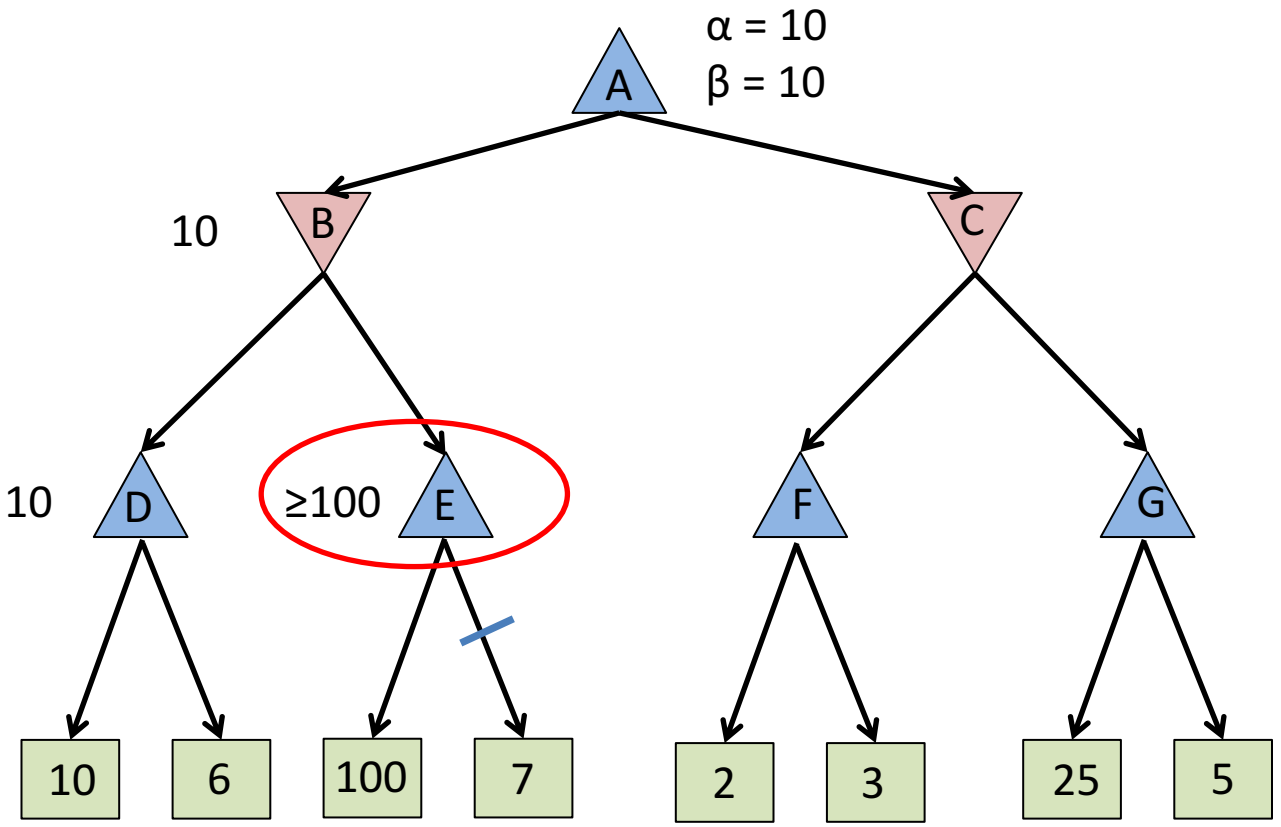
α-β Example

- Initialize: α = - ∞ and β = + ∞
- Evaluate all children of D, compute value 10 for D, and set α = 10
- Node B sees value of D and sets β = 10 for current path to root (A, here)



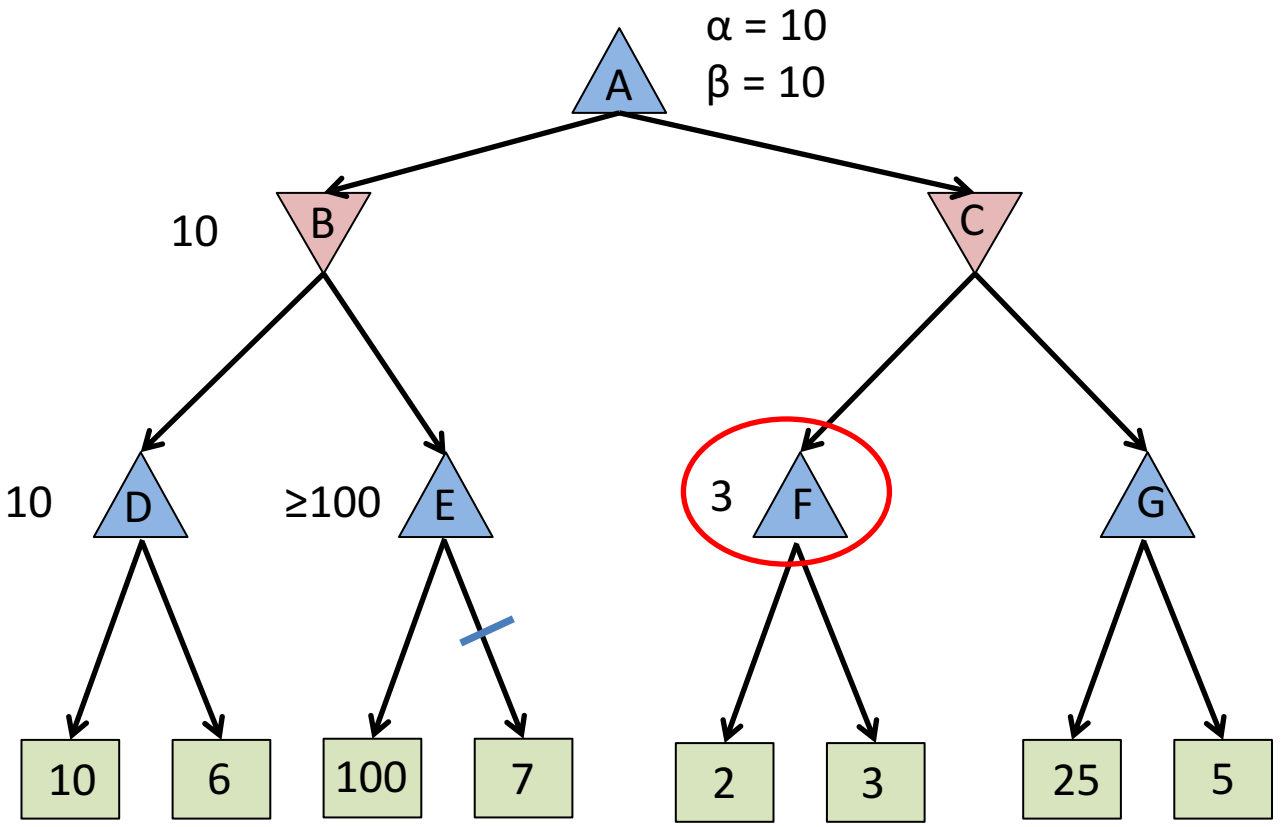
α-β Example

- With α = 10 and β = 10:
- Evaluate first child of E, see value 100 > β, so stop checking children
- Value of node B now 10; no changes to either α or β



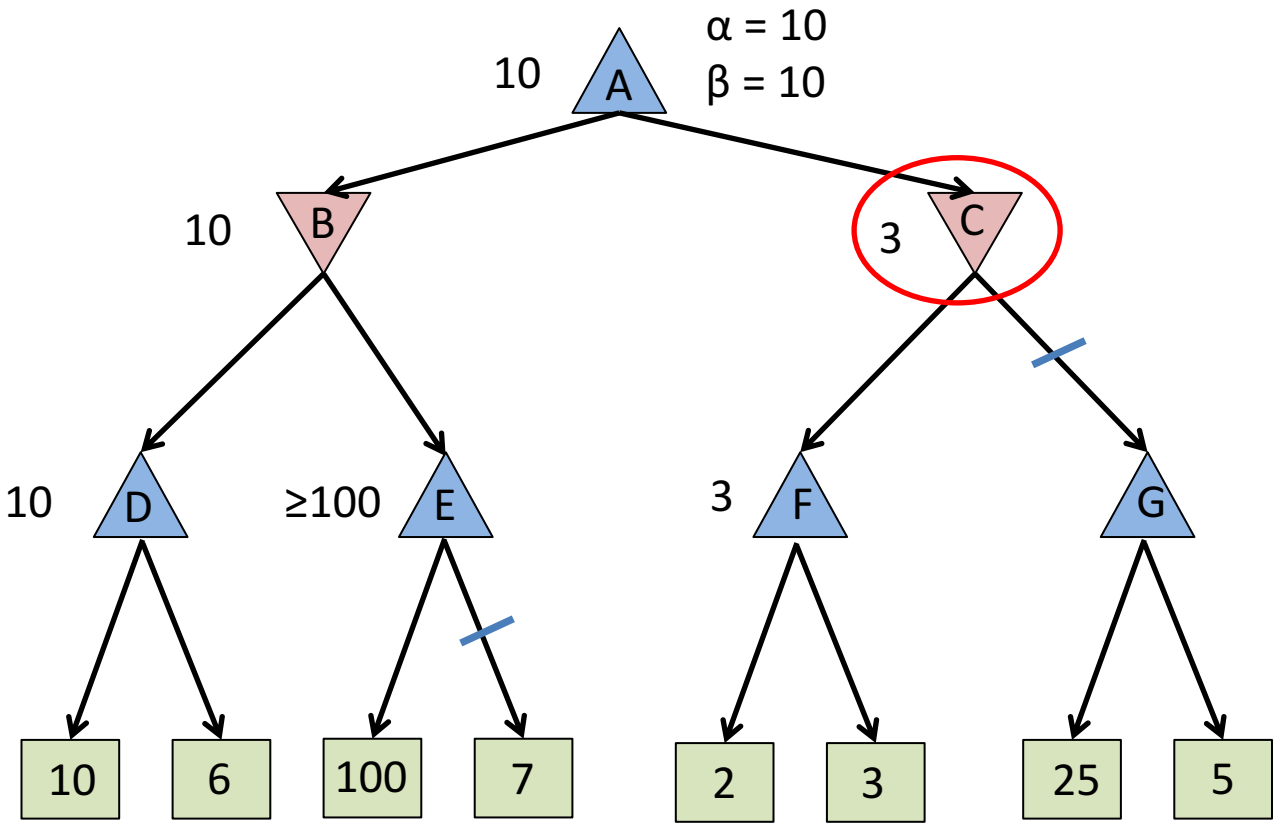
α-β Example

- With α = 10 and β = 10:
- Evaluate all children of F, compute value 3 for F; no change to α



α-β Example

- With α = 10 and β = 10:
- Node C value currently 3, which is less than α, so we stop checking children
- Now compute minimax value 10 for node A



Another α-β Example

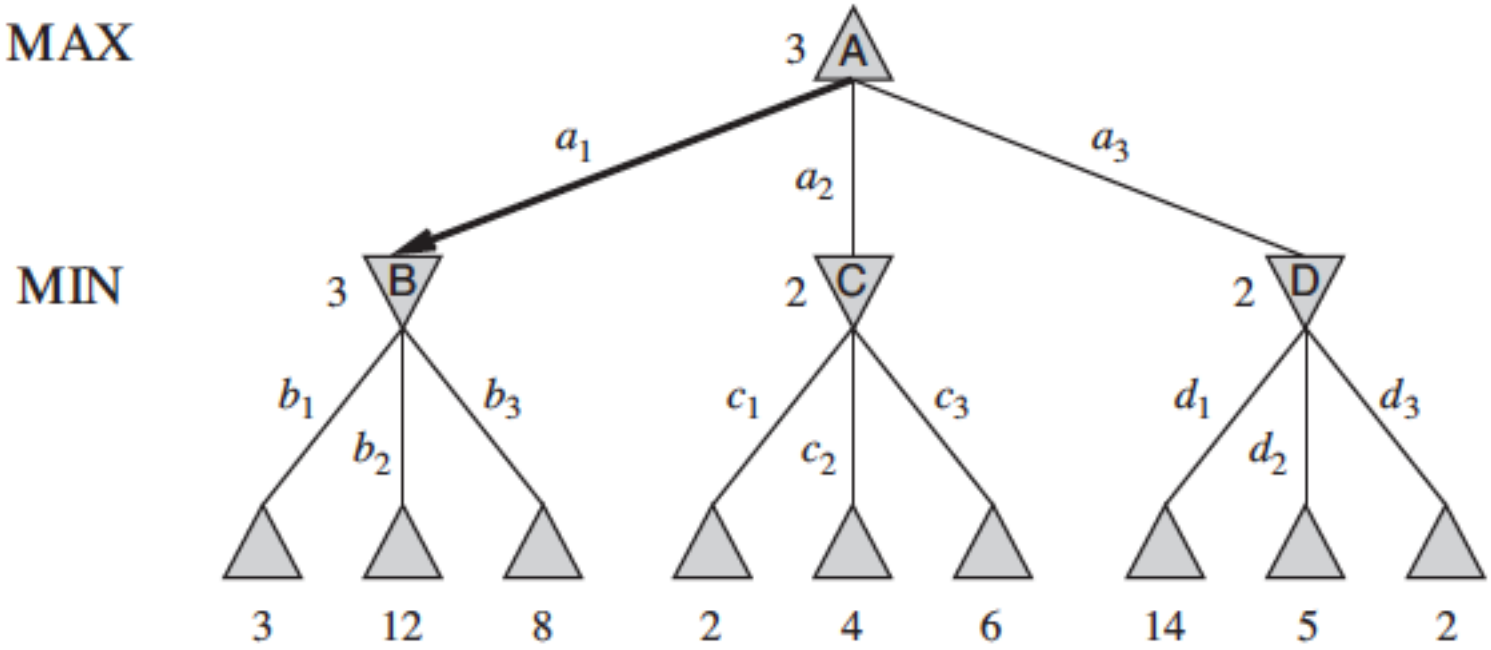


Figure 5.2 from textbook

Another α-β Example (cont'd)

"at most 3"

"exactly 3"

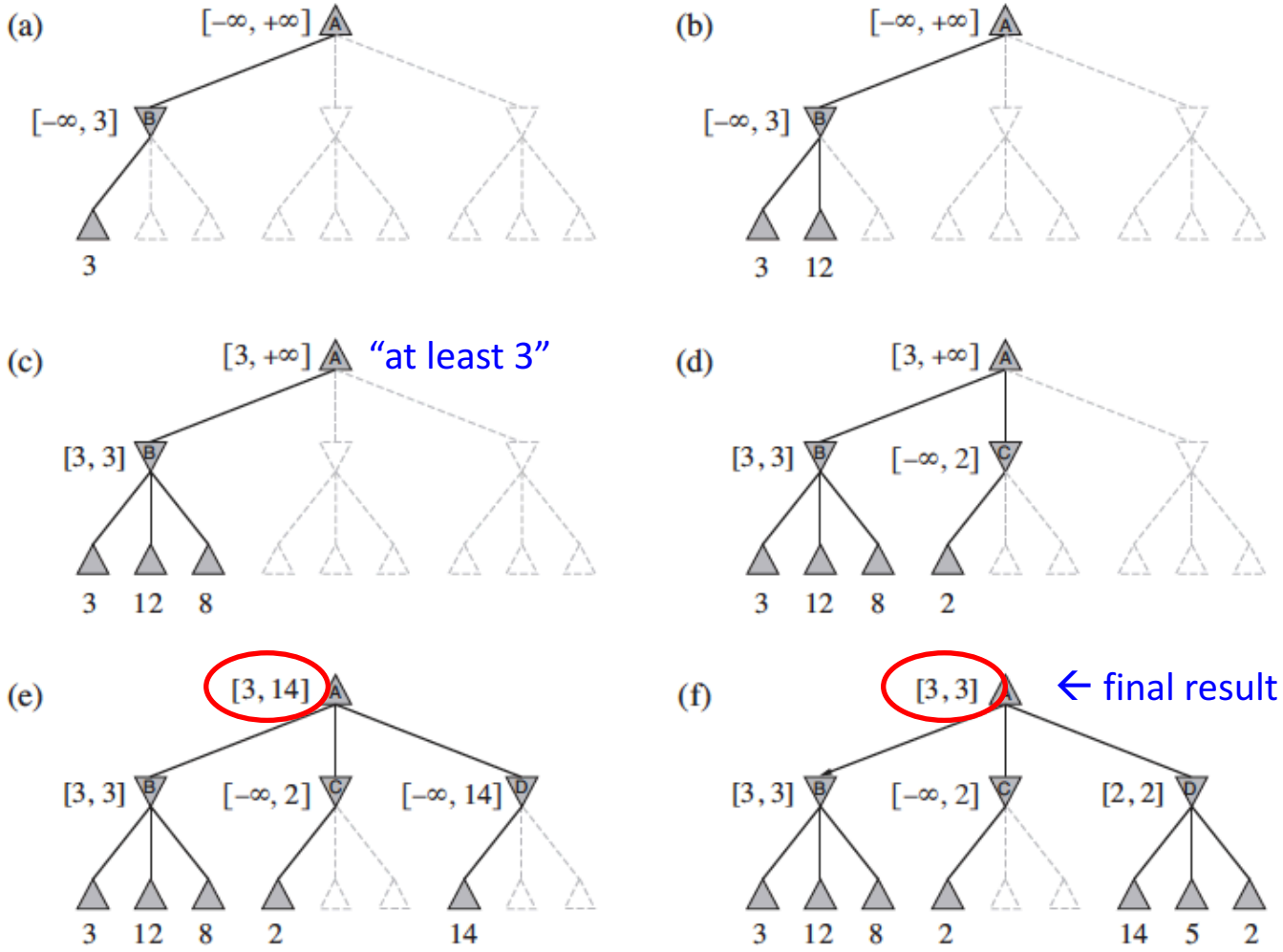


Figure 5.5 from textbook

Another α-β Example (concl.)

Read at your leisure (explanation from textbook):

Figure 5.5 FILES: figures/alpha-beta-progress.eps (Tue Nov 3 16:22:20 2009). Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.