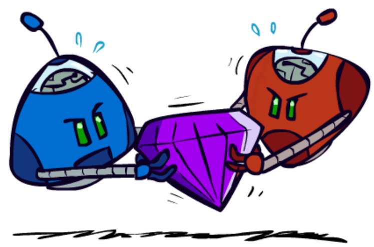
CSE 473: Artificial Intelligence

Adversarial Search

Dan Weld



Based on slides from

Dan Klein, Stuart Russell, Pieter Abbeel, Andrew Moore and Luke Zettlemoyer

(best illustrations from ai.berkeley.edu)

Outline

- Adversarial Search
  - Minimax search
  - α-β search
  - Evaluation functions
  - Expectimax
- Reminder:
  - Project 2 due in 5 days

Types of Games

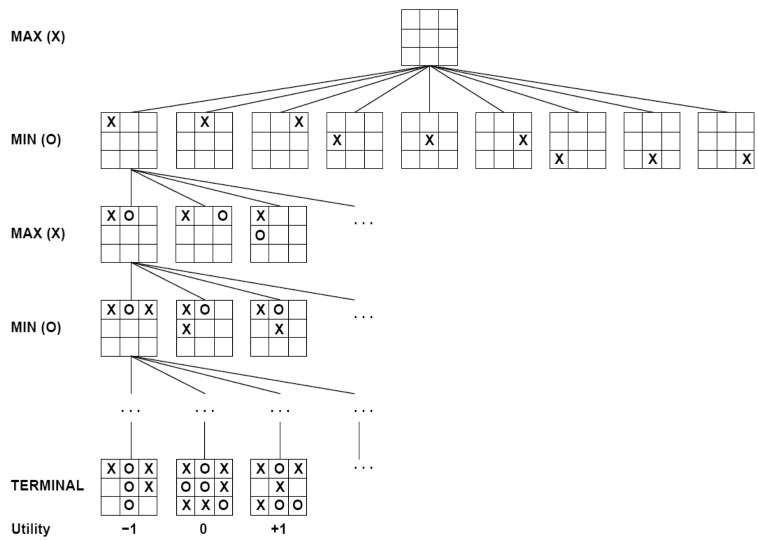
	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon, monopoly
imperfect information	stratego	bridge, poker, scrabble, nuclear war

Number of Players? 1, 2, …?

Deterministic Games

- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: S x A → S
  - Terminal Test: S → {t,f}
  - Terminal Utilities: S x P→ R
- Solution for a player is a ***policy***: S → A

Tic-tac-toe Game Tree



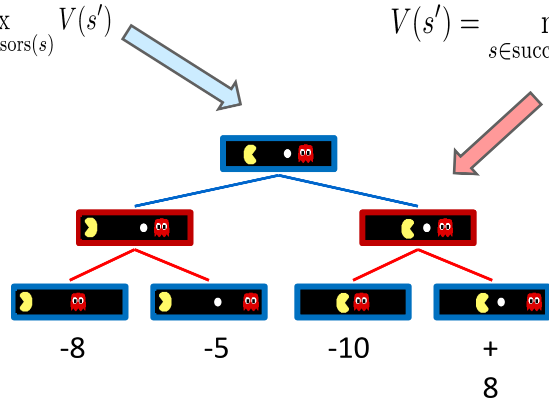
Minimax Values

States Under Agent's Control:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent's Control:

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

$$V(s) = \text{known}$$

Slide from Dan Klein & Pieter Abbeel - ai.berkeley.edu

Minimax Implementation

Need ***Base case*** for recursion

```
def max-value(state):
    if leaf?(state), return U(state)
    initialize v = -∞
    for each c in children(state)
        v = max(v, min-value(c))
    return v
```

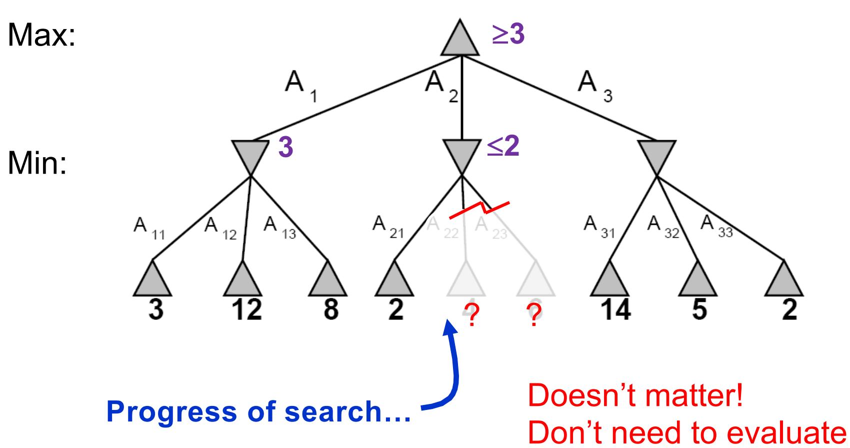
```
def min-value(state):
    if leaf?(state), return U(state)
    initialize v = +∞
    for each c in children(state)
        v = min(v, max-value(c))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

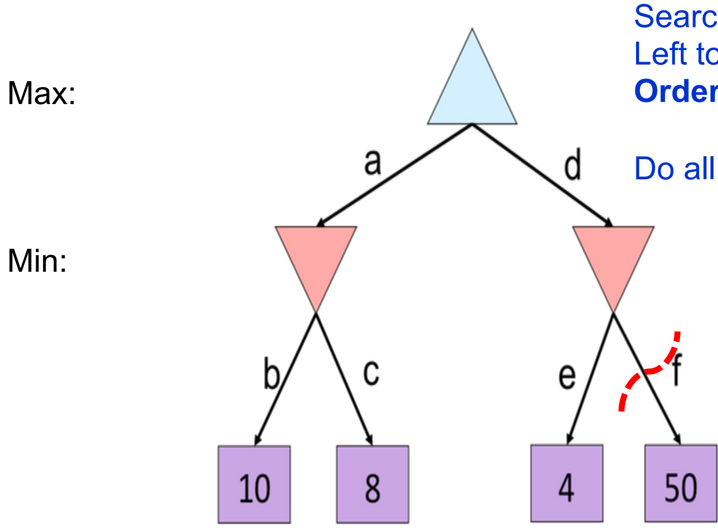
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Slide from Dan Klein & Pieter Abbeel - ai.berkeley.edu

α-β Pruning Example



Alpha-Beta Quiz

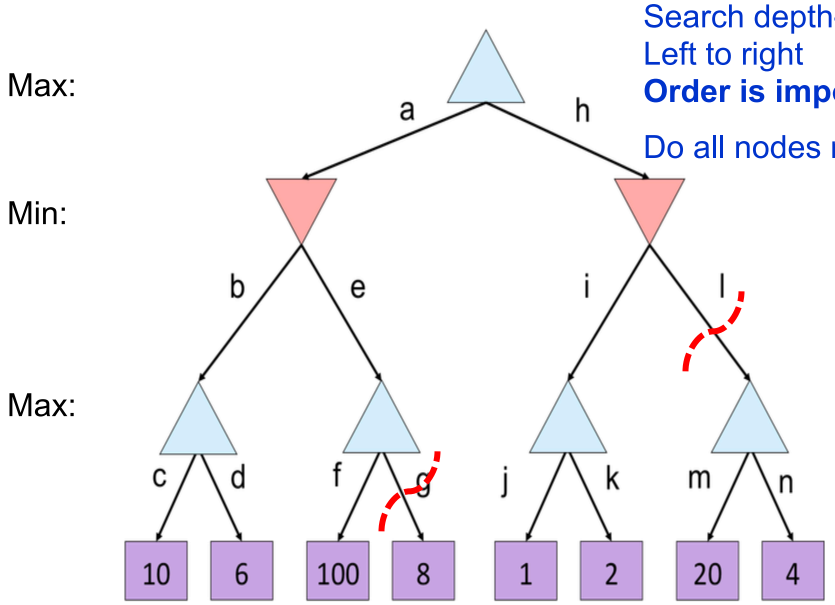


Search depth-first
Left to right
Order is important

Do all nodes matter?

Slide from Dan Klein & Pieter Abbeel - ai.berkeley.edu

Alpha-Beta Quiz 2



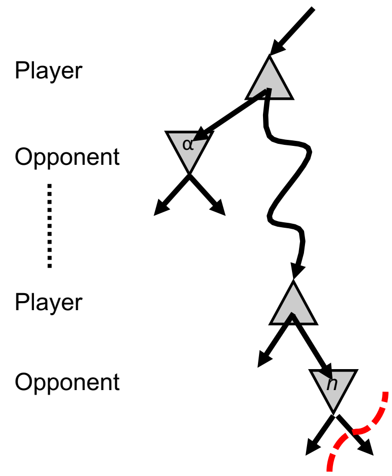
Search depth-first
Left to right
Order is important

Do all nodes matter?

Slide from Dan Klein & Pieter Abbeel - ai.berkeley.edu

α-β Pruning

- α is MAX's best choice on path to root
- If *n* becomes worse than α, MAX will avoid it, so can stop considering *n*'s other children
- Define β similarly for MIN



Min-Max Implementation

```
def max-val(state        ):
    if leaf?(state), return U(state)
    initialize v = -∞
    for each c in children(state):
        v = max(v, min-val(c        ))

    return v
```

```
def min-val(state        ):
    if leaf?(state), return U(state)
    initialize v = +∞
    for each c in children(state):
        v = min(v, max-val(c        ))

    return v
```

Slide adapted from Dan Klein & Pieter Abbeel - ai.berkeley.edu

Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-val(state, α, β):
    if leaf?(state), return U(state)
    initialize v = -∞
    for each c in children(state):
        v = max(v, min-val(c, α, β))

    return v
```

```
def min-val(state , α, β):
    if leaf?(state), return U(state)
    initialize v = +∞
    for each c in children(state):
        v = min(v, max-val(c, α, β))

    return v
```

Slide adapted from Dan Klein & Pieter Abbeel - ai.berkeley.edu

Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-val(state, α, β):
    if leaf?(state), return U(state)
    initialize v = -∞
    for each c in children(state):
        v = max(v, min-val(c, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

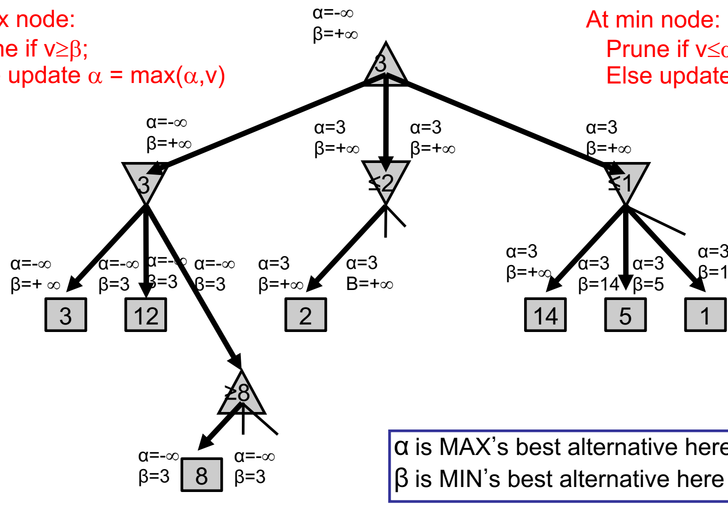
```
def min-val(state, α, β):
    if leaf?(state), return U(state)
    initialize v = +∞
    for each c in children(state):
        v = min(v, max-val(c, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Slide adapted from Dan Klein & Pieter Abbeel - ai.berkeley.edu

Alpha-Beta Pruning Example

At max node:
Prune if v≥β;
Else update α = max(α,v)

At min node:
Prune if v≤α;
Else update β = min(β,v)



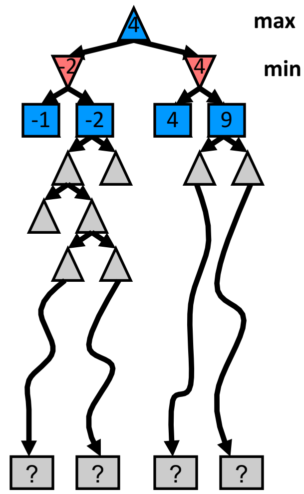
α is MAX's best alternative here or above
β is MIN's best alternative here or above

Alpha-Beta Pruning Properties

- This pruning has no effect on final result at the root
- ***Values*** of intermediate nodes might be wrong!
  - but, they are correct ***bounds***
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - ***Doubles*** solvable depth!
  - (But complete search of complex games, e.g. chess, is still hopeless…

Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an ***evaluation function*** for non-terminal positions
- Example:
  - Suppose we have 3 min/move, can explore 1M nodes / sec
  - So can check 200M nodes per move
  - α-β reaches about depth 10 → decent chess program
- Guarantee of optimal play is gone
- More plies makes a BIG difference



Depth Matters

- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- Good example of the tradeoff between complexity of ***features*** and complexity of ***computation***

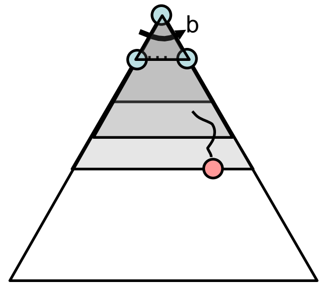
[Demo: depth limited (L6D4,

Iterative Deepening

Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less. (DFS gives up on any path of length 2)
2. If "1" failed, do a DFS which only searches paths of length 2 or less.
3. If "2" failed, do a DFS which only searches paths of length 3 or less.

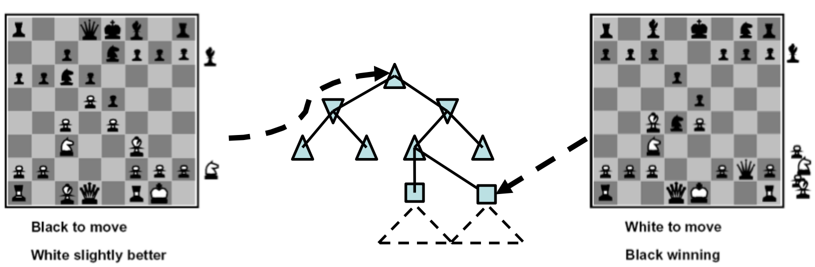
….and so on.



Creates an ***anytime algorithm***

Heuristic Evaluation Function

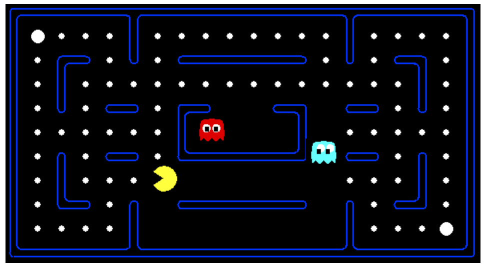
- Function which scores non-terminals



- Ideal function: returns the ***true utility*** of the position
- In practice: need a simple, fast ***approximation***
  - typically weighted linear sum of features:
  - e.g. $f_1(s)$ = (num white queens – num black queens), etc.

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

Evaluation for Pacman



What features would be good for Pacman?

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

Which algorithm?

α-β, depth 4, simple eval fun

QuickTime™ and a
GIF decompressor
are needed to see this picture.

Which algorithm?

α-β, depth 4, better eval fun

QuickTime™ and a
GIF decompressor
are needed to see this picture.

Why Pacman Starves

- He knows his score will go up by eating the dot now
- He knows his score will go up just as much by eating the dot later on
- There are no point-scoring opportunities after eating the dot
- Therefore, waiting seems just as good as eating