343H: Honors AI

Lecture 6: Adversarial Search

2/4/2014

Kristen Grauman

UT-Austin

Slides courtesy of Dan Klein, UC-Berkeley
Unless otherwise noted

Announcements

- Assignments
  - Reminder - PS1 due Thursday by 11:59 pm
  - PS2 will be out Thursday, due 2 weeks later
- Autograder:
  - The autograder isn't perfect, and it is only a lower bound on your score (… though the autograder is quite good, and if your code autogrades as wrong, the autograder is almost always correct)

Today

- Wrap up local search
- Adversarial search with game trees

Last time: local search

- Local search
  - Hill climbing
  - Simulated annealing
  - Genetic algorithms
  - Continuous search spaces

Review: Exercise 4.1

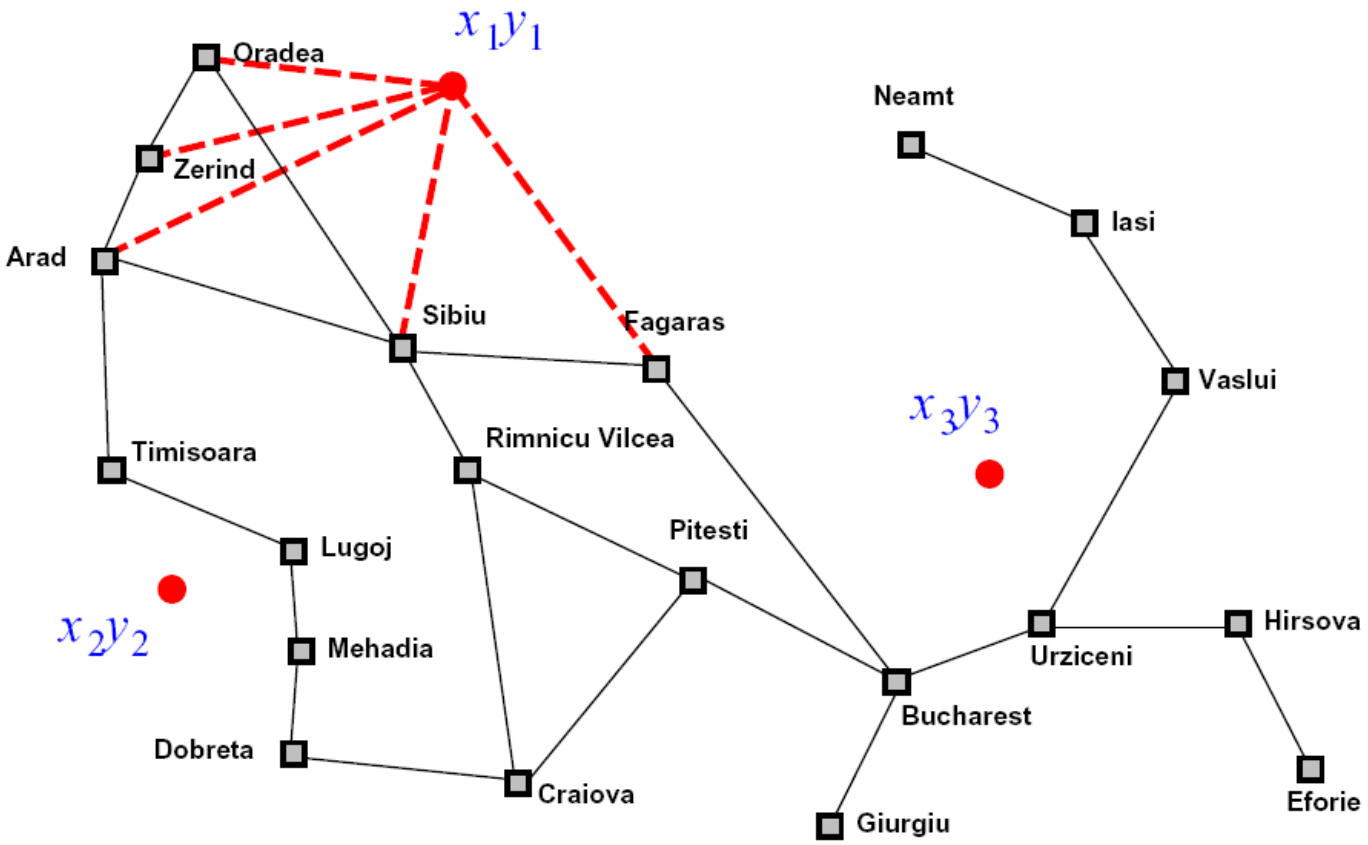
- Which algorithm results from these special cases?
  1. Local beam search with k=1
  2. Local beam search with one initial state and no limit on the number of states retained
  3. Simulated annealing with T=0 at all times
  4. Simulated annealing with T= inf at all times
  5. Genetic algorithm with population size N=1

Last time: local search

- Local search
  - Hill climbing
  - Simulated annealing
  - Genetic algorithms
  - Continuous search spaces

Continuous Problems

- Placing airports in Romania
  - States: $(x_1,y_1,x_2,y_2,x_3,y_3)$
  - Cost: sum of squared distances to closest city



Gradient Methods

- How to deal with continous (therefore infinite) state spaces?
- Discretization: bucket ranges of values
  - E.g. force integral coordinates
- Continuous optimization
  - E.g. gradient ascent

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)$$

$$x \leftarrow x + \alpha \nabla f(x)$$

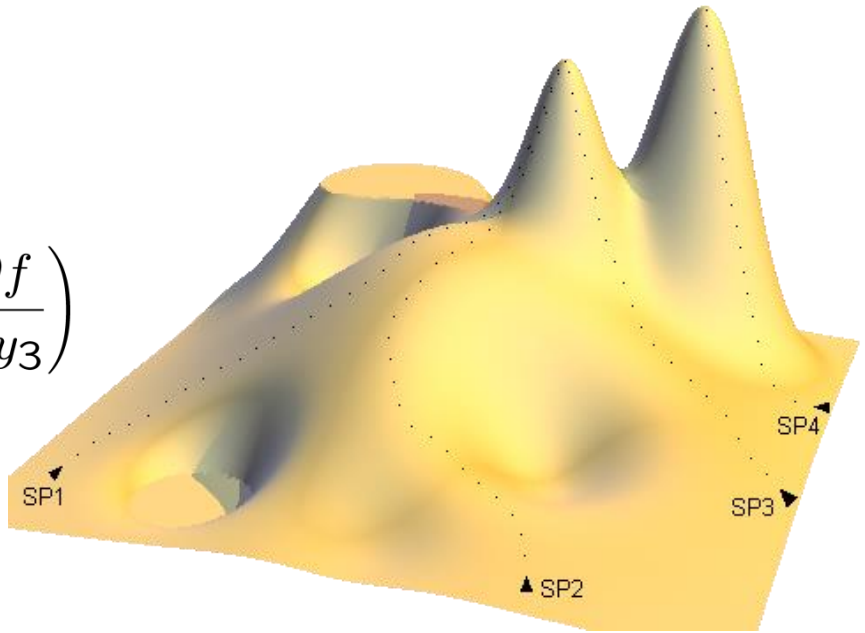


Image from vias.org

Example: Continuous local search

Goal: Enable an Aibo to walk as fast as possible

- Start with a **parameterized walk**
- **Learn** fastest possible parameters
- **No simulator** available:
  - Learn entirely on robots
  - Minimal human intervention

A parameterized walk

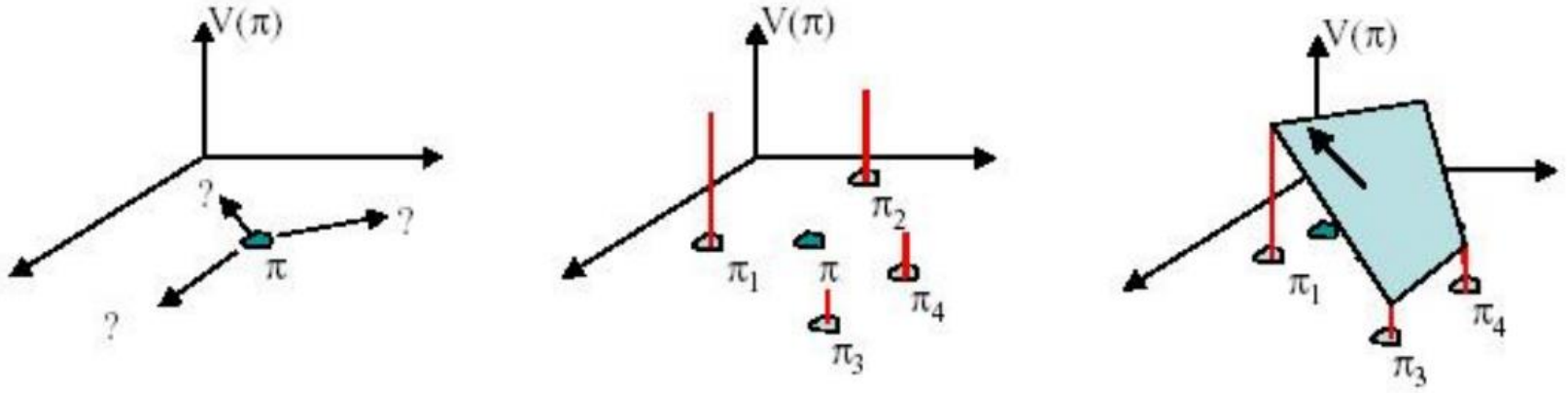
- Trot gait with elliptical locus on each leg
- 12 continuous parameters (ellipse length, height, position, body height, etc)

Experimental setup

- Policy $\pi = \{\theta_1, \ldots, \theta_{12}\}$, $V(\pi) =$ walk **speed** when using π
- Training Scenario
  - Robots **time themselves** traversing fixed distance
  - Multiple traversals (3) per policy to account for **noise**
  - **Multiple robots** evaluate policies simultaneously
  - Off-board computer **collects results**, **assigns policies**

Policy gradient reinforcement learning

- From π want to move in direction of **gradient** of $V(\pi)$
  - Can't compute $\frac{\partial V(\pi)}{\partial \theta_i}$ directly: **estimate** empirically
- Evaluate **neighboring policies** to estimate gradient
- Each trial randomly varies **every parameter**



AUSTIN VILLA
ROBOT SOCCER TEAM
THE UNIVERSITY OF TEXAS AT AUSTIN

Home
Nao League
Sim. League
4 Legged League
@Home
Research
Papers
Competitions
Members
Press
Downloads
Related Sites
Restricted

W3C CSS
W3C XHTML 1.0

Learning to Walk

To learn to walk faster, the Aibos evaluated different gaits by walking back and forth across the field between pairs of beacons, timing how long each lap took. **The learning was all done on the physical robots with no human intervention** (other than to change the batteries). To speed up the process, we had three Aibos working simultaneously, dividing up the search space accordingly.

Experimental Setup
(11.0 MB MPEG)

Initial Gait
(2.8 MB MPEG)

Initially, the Aibo's gait is clumsy and fairly slow (less than 150 mm/s). We deliberately started with a poor gait so that the learning process would not be systematically biased towards our best hand-tuned gait, which might have been locally optimal.

Midway through the training process, the Aibo is moving much faster than it was initially. However, it still exhibits some irregularities that slow it down.

After traversing the field a total of just over 1000 times over the course of 3 hours, we achieved our best learned gait, which allows the Aibo to move at approximately 291 mm/s. To our knowledge, this is **the fastest reported walk on an Aibo** as of November 2003. The hash marks on the field are 200 mm apart. The Aibo traverses 9 of them in 6.13 seconds demonstrating a speed of 1800mm/6.13s >

Today

- Wrap up local search
- Adversarial search with game trees
  - Minimax
  - Alpha-beta pruning

Game Playing State-of-the-Art

- **Checkers:** 1950: First computer player. 1994: First computer champion. Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions. Checkers is now solved!

- **Chess:** Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue examined 200 million positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

- **Go:** Human champions are just beginning to be challenged by machines, though the best humans still beat the best machines. In go, $b > 300$! Classic programs use pattern knowledge bases, but big recent advances using Monte Carlo (randomized) expansion methods.

- **Pacman:** ?

Game Playing

- Many different kinds of games!
- Axes:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?
- Want algorithms for calculating a strategy (policy) which recommends a move in each state

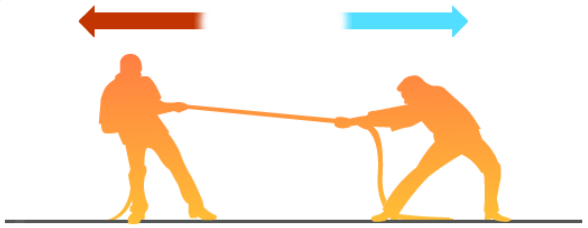
Deterministic Games

- 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: SxA $\rightarrow$ S
  - Terminal Test: S $\rightarrow$ {t,f}
  - Terminal Utilities: SxP $\rightarrow$ R

- Solution for a player is a policy: S $\rightarrow$ A

Zero-sum games

- Zero-sum games
  - Agents have opposite utilities (values on the outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition

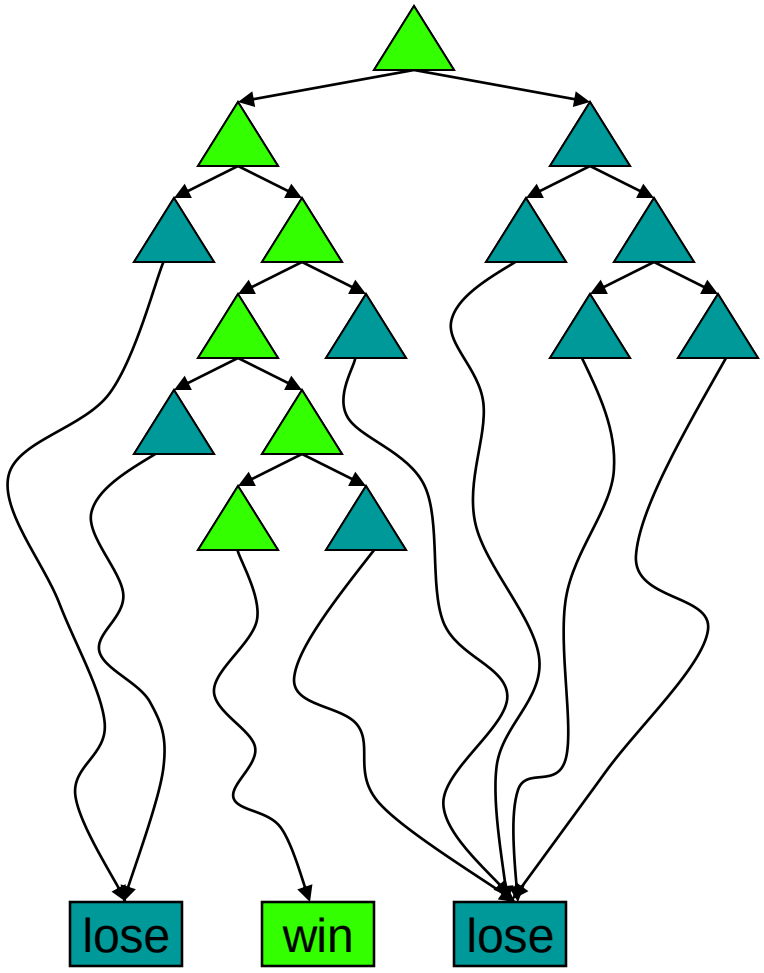


- General games
  - Agents have independent utilities
  - Cooperation, indifference, competition, …
  - More later on non-zero-sum games

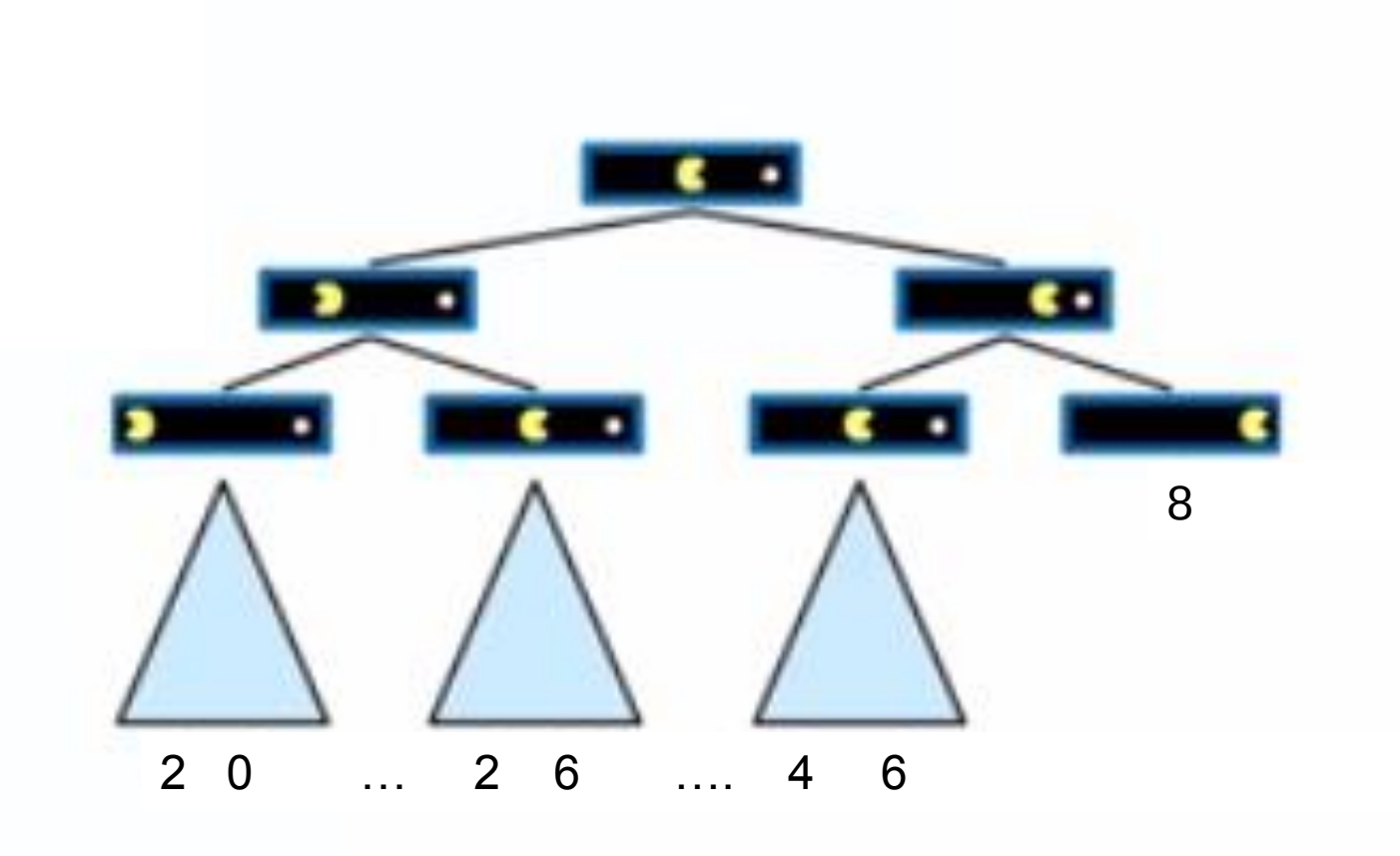
Adapted from Dan Klein

From single player to adversarial

- Deterministic, single player, perfect information:
  - Know the rules
  - Know what actions do
  - Know when you win
  - E.g. Freecell, 8-Puzzle, Rubik's cube
- … it's just search!
- Now, a reinterpretation:
  - Each node stores a value: the best outcome it can reach
  - This is the maximal outcome of its children (the max value)
  - Note that we don't have path sums as before (utilities at end)
- After search, can pick move that leads to best node

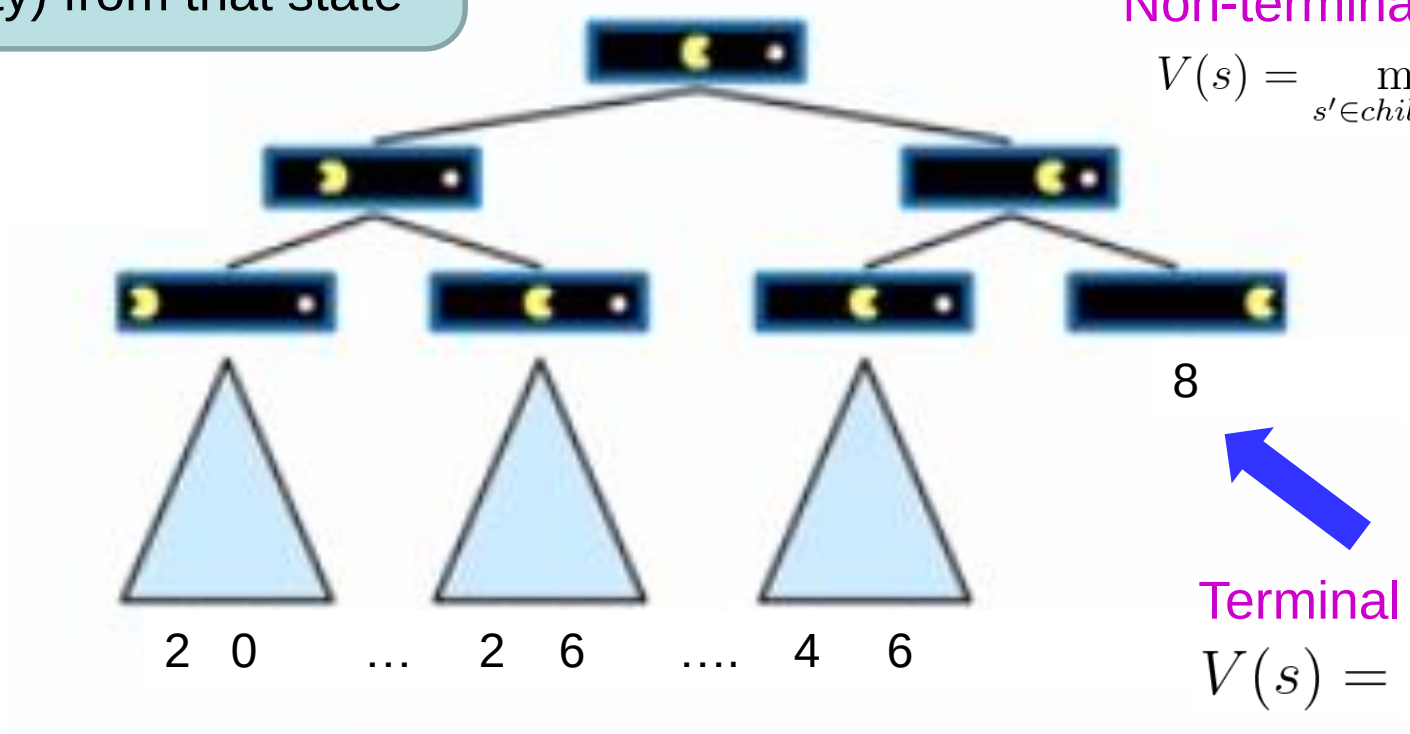


Recall: Single-agent trees



Value of a state

Value of a state: the best achievable outcome (utility) from that state



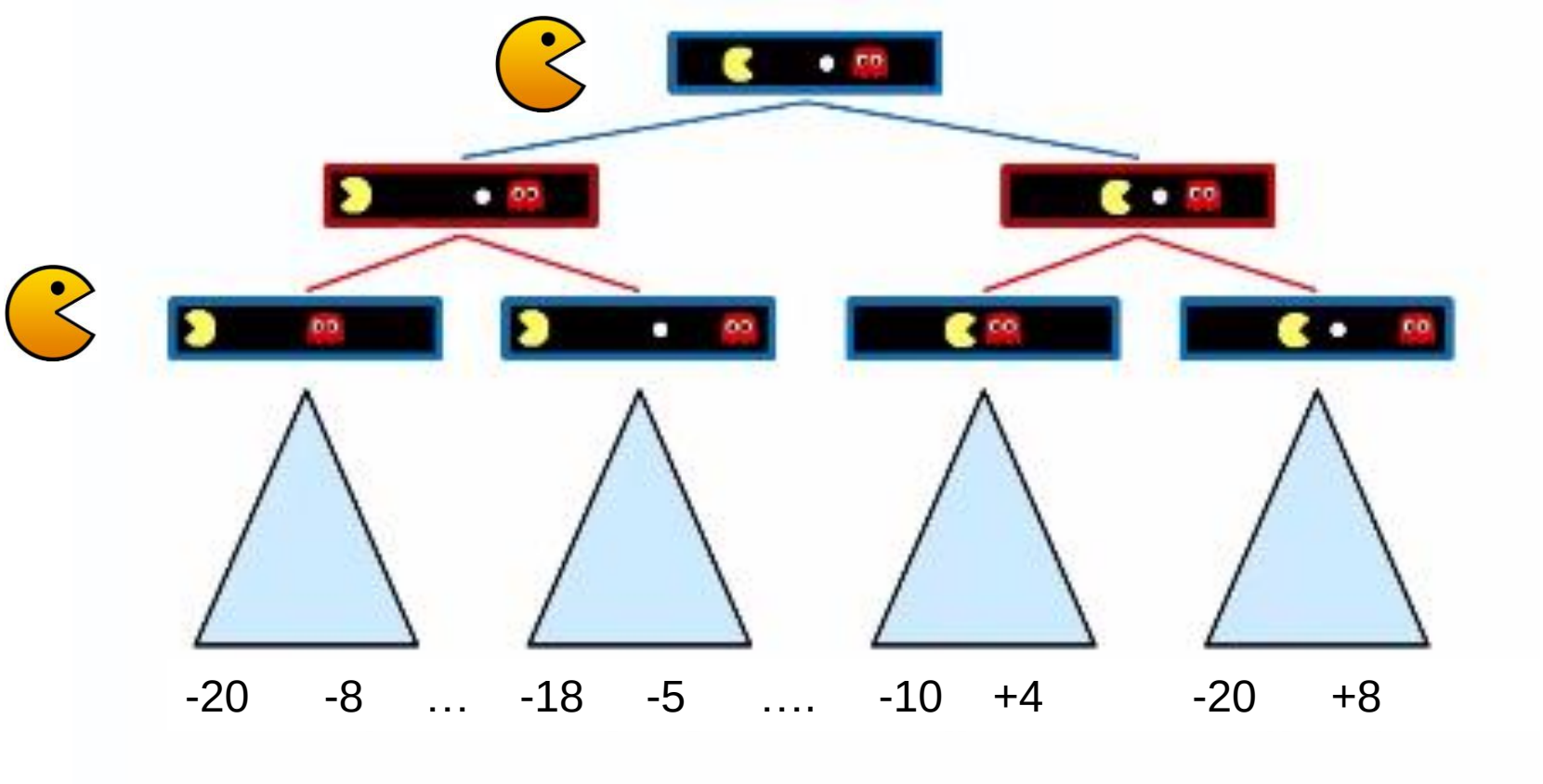
Non-terminal states:

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

Terminal states:

$$V(s) = known$$

Adversarial game trees



What is the value of a state in the case of an adversary?

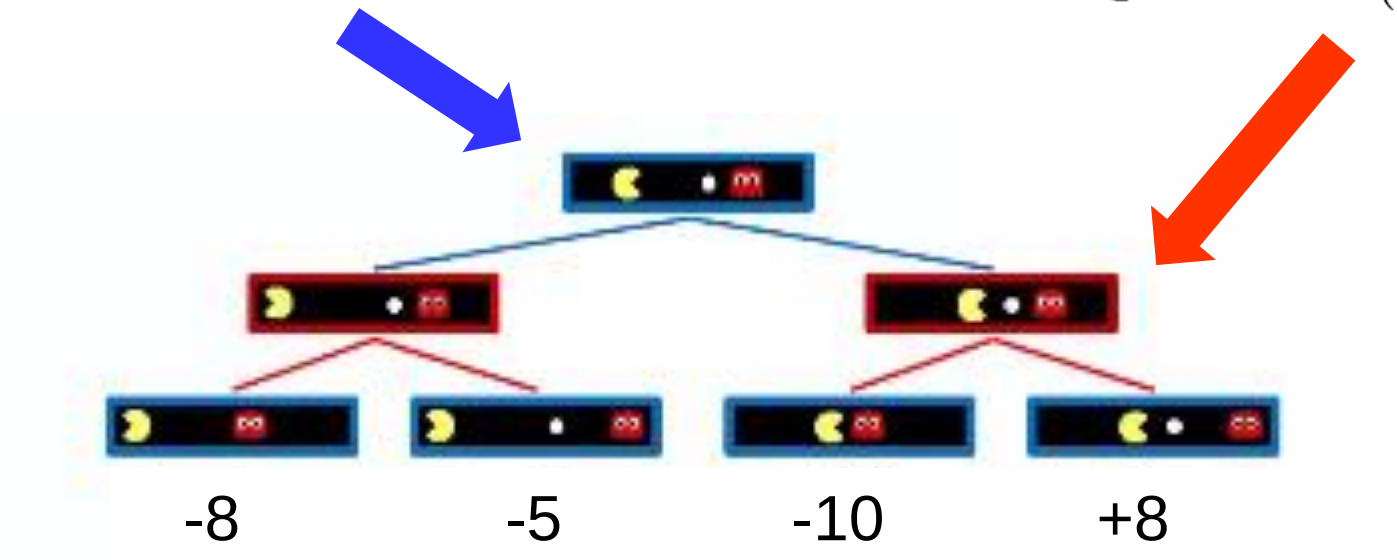
Minimax values

States under agent's control:

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

States under opponent's control:

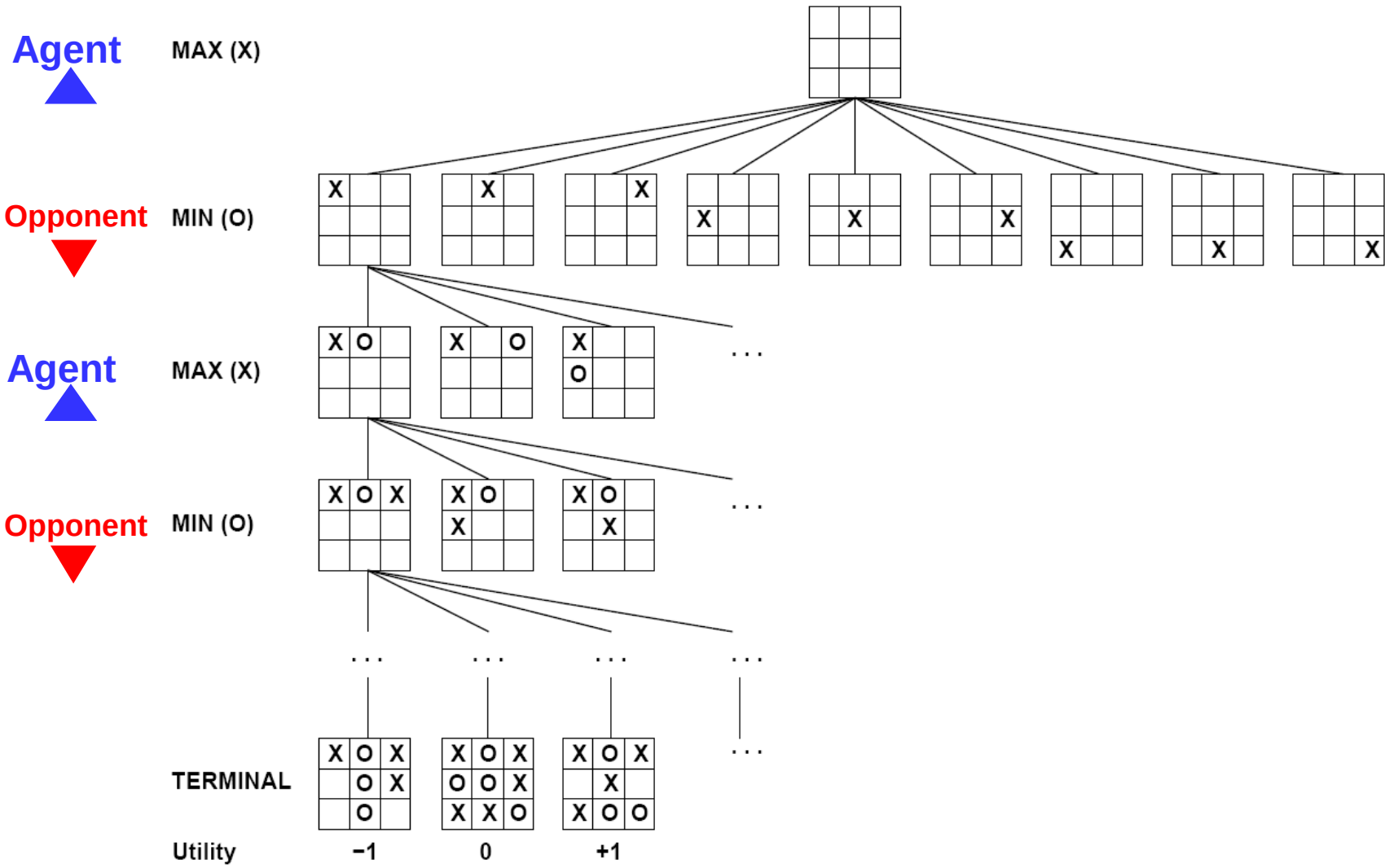
$$V(s') = \min_{s \in successors(s')} V(s)$$



Terminal states:

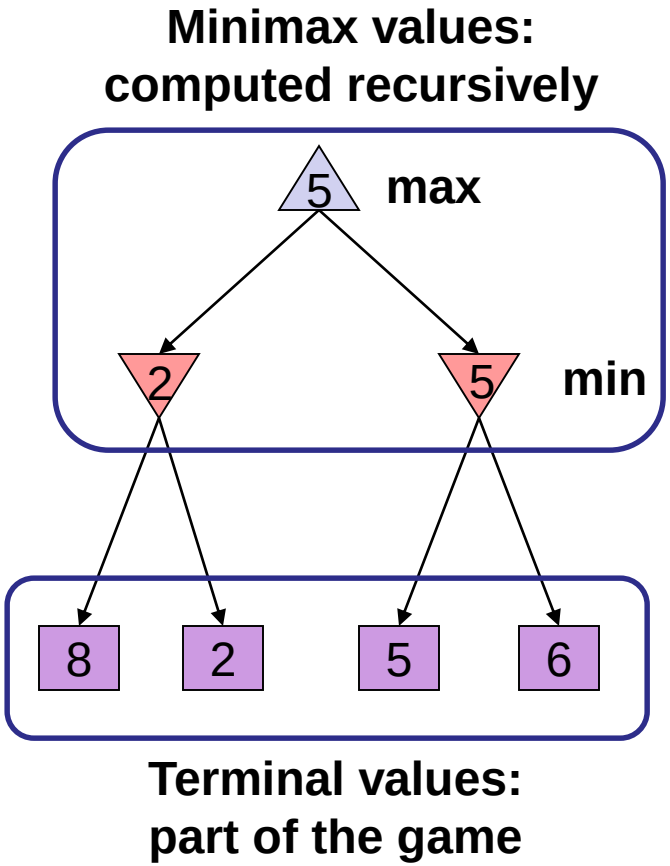
$$V(s) = known$$

Tic-tac-toe Game Tree



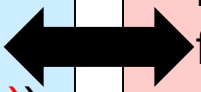
Adversarial search: Minimax

- Deterministic, zero-sum game
- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: best achievable utility against a rational (optimal) adversary



Minimax implementation

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```



```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

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

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

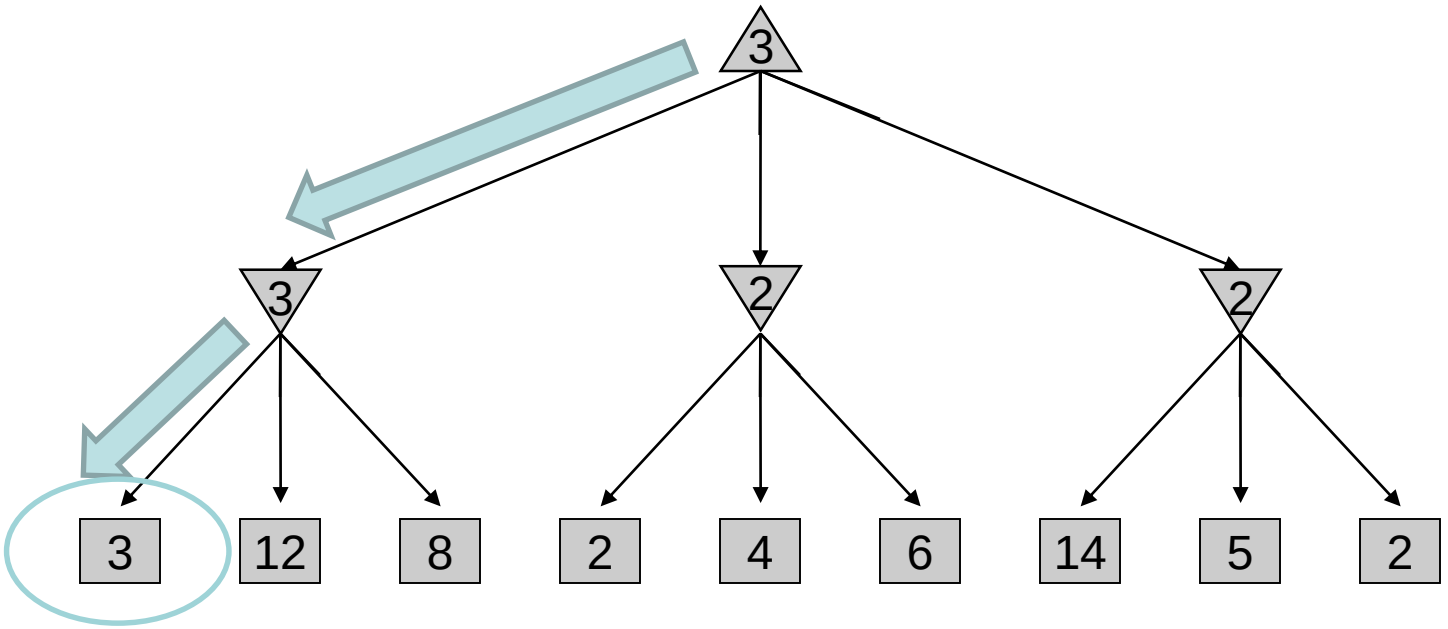
Minimax implementation

```
def value(state):
    If the state is a terminal state: return the state's utility
    If the next agent is MAX: return max-value(state)
    If the next agent is MIN: return min-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

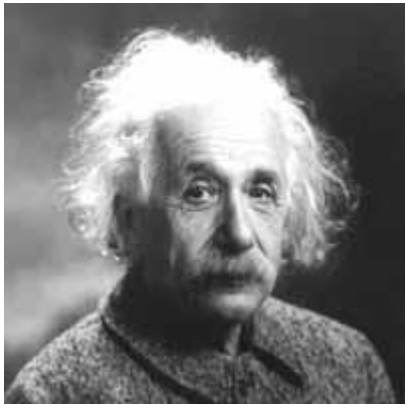
Minimax Example



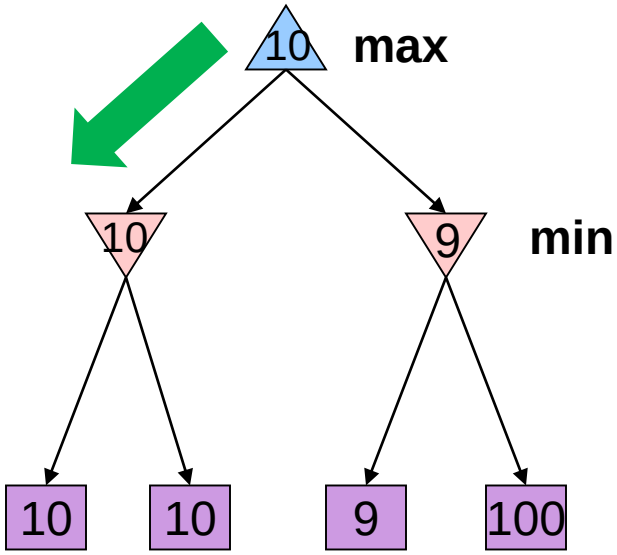
Minimax efficiency

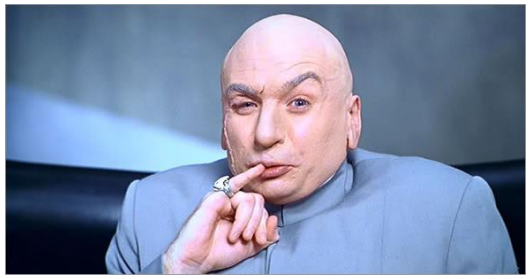
- Time complexity?
  - $O(b^m)$
- Space complexity?
  - O(bm)
- For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?

Minimax efficiency



Optimal against a perfect player.

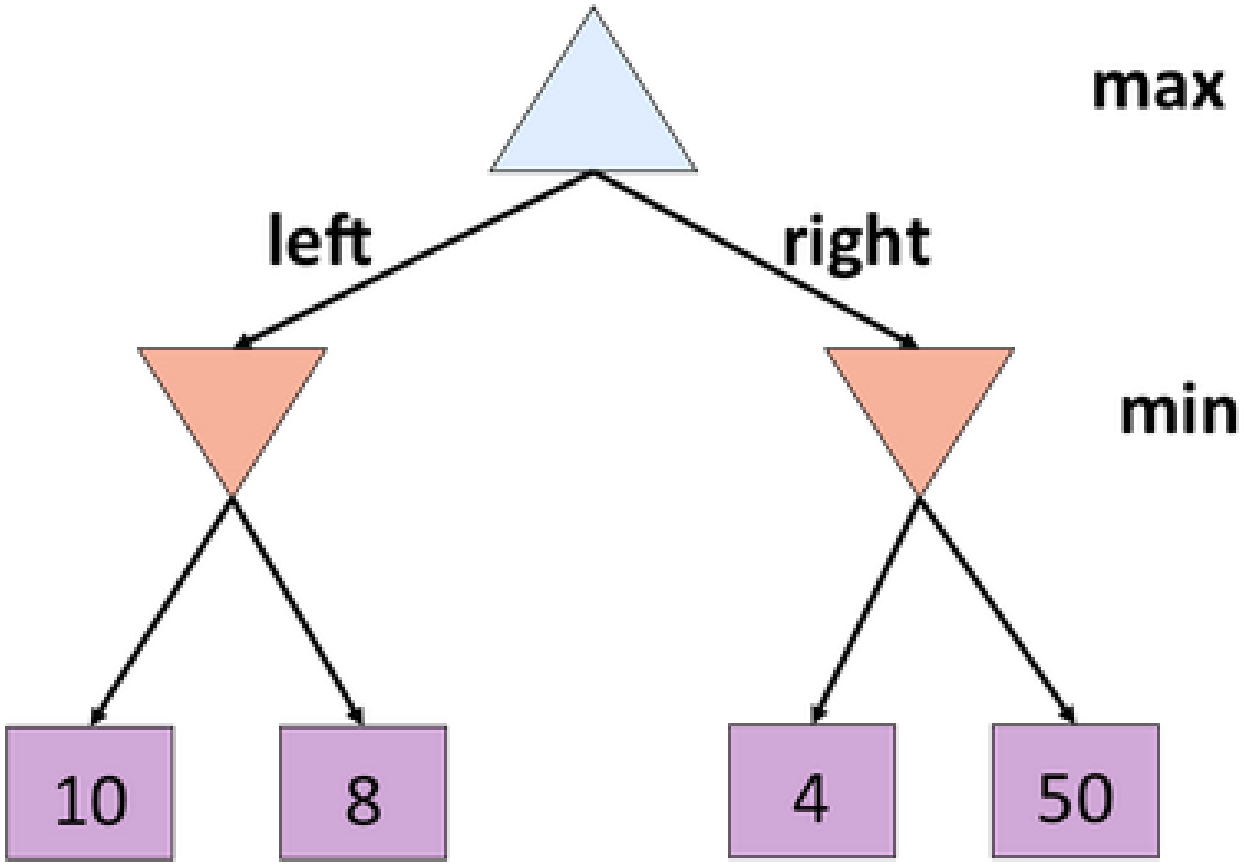




Otherwise?

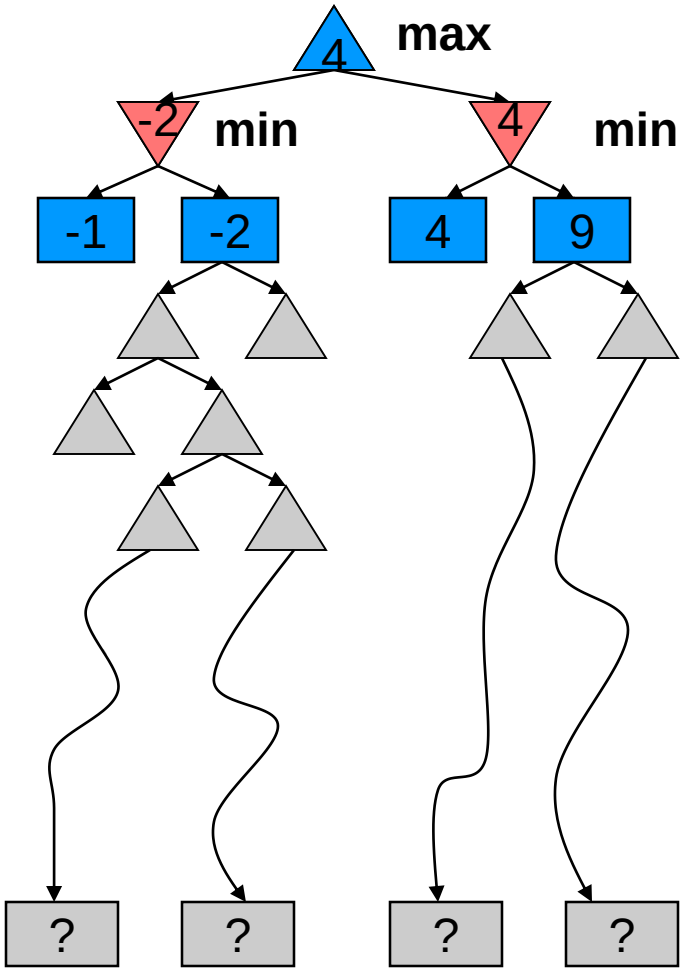
Adapted from Dan Klein

Quiz: Minimax



Dealing with resource limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Guarantee of optimal play is gone
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - With α-β reaches about depth 8 – decent chess program

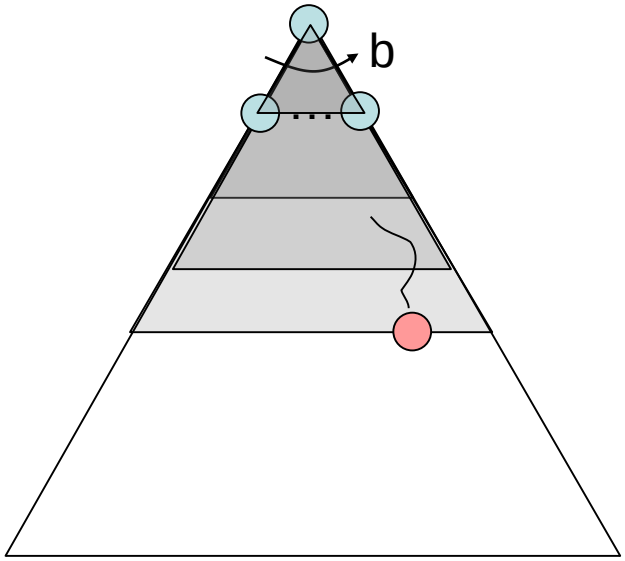


Iterative deepening for “anytime” algorithm

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.

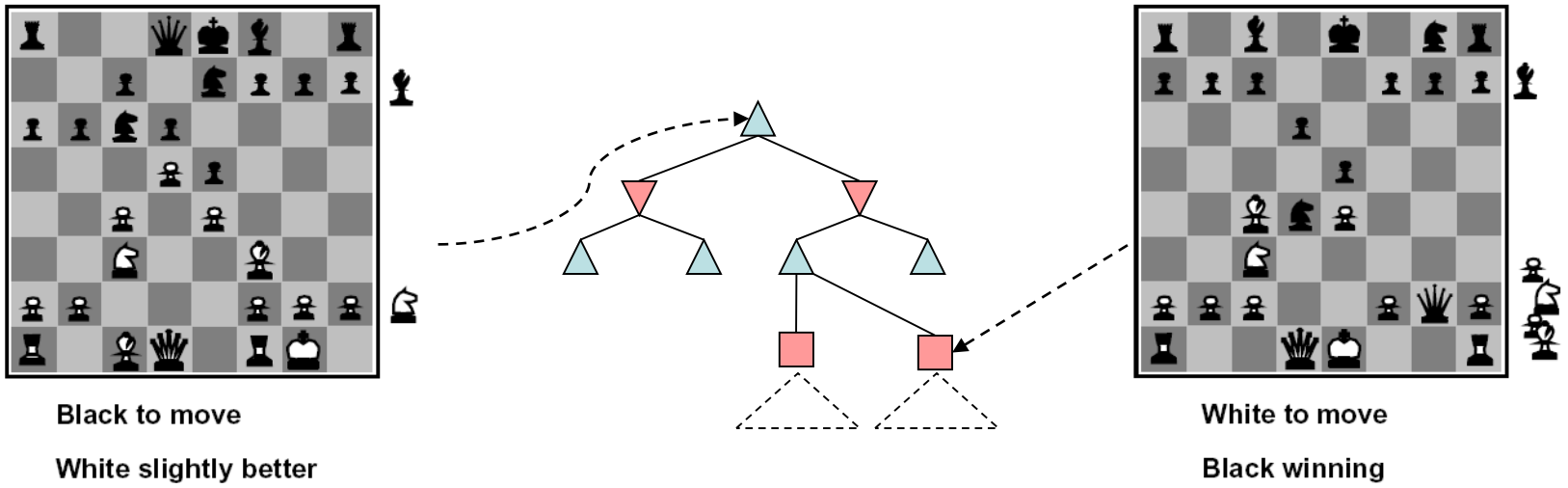


Trade offs in complexity

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

Evaluation Functions

- Function which scores non-terminals in depth-limited search

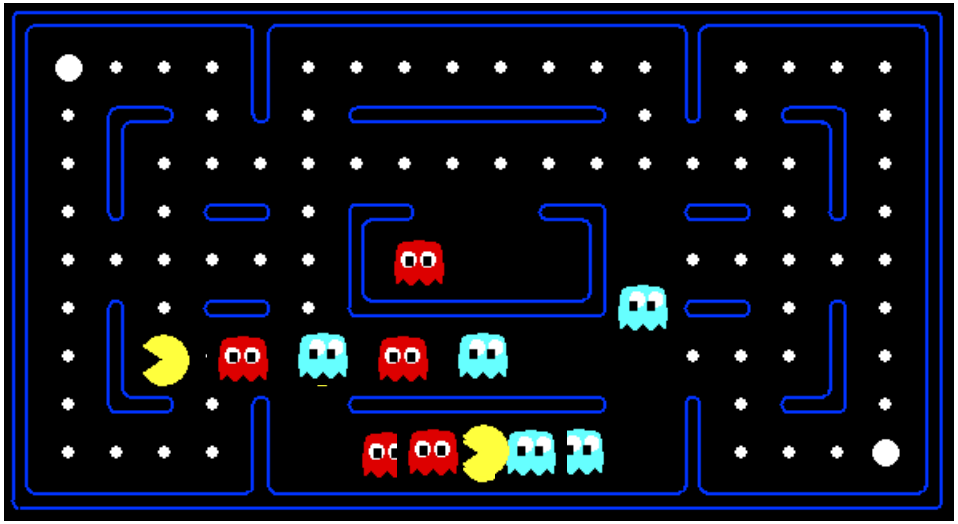


- Ideal function: returns the utility of the position
- In practice: typically weighted linear sum of features:

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

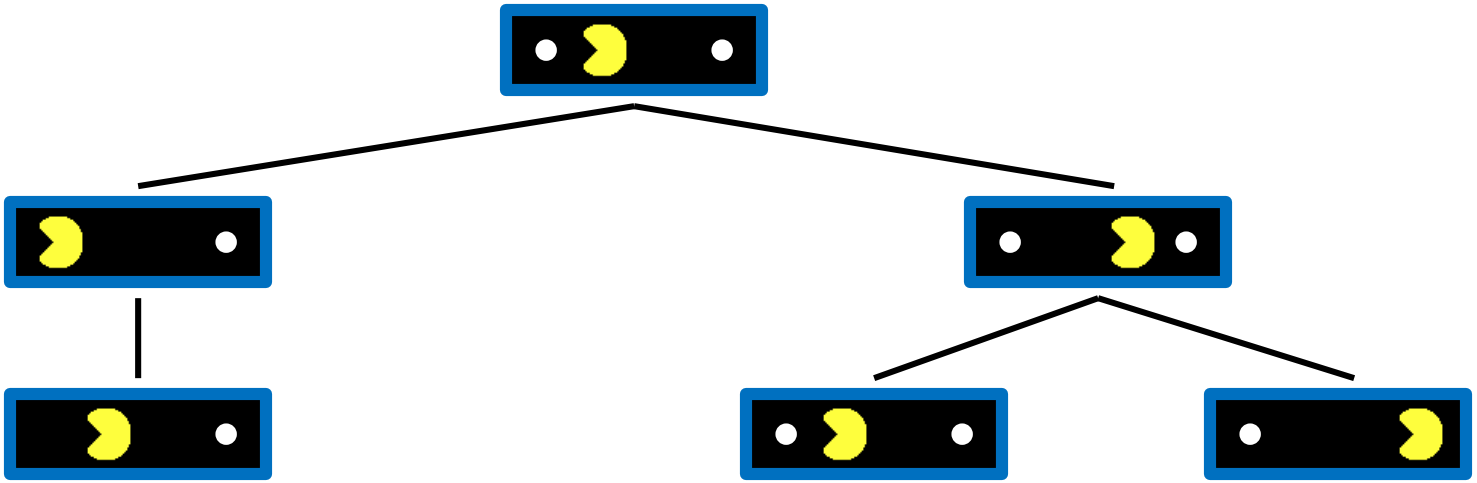
- e.g. $f_1(s)$ = (num white queens – num black queens), etc.

What should the evaluation function report?



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

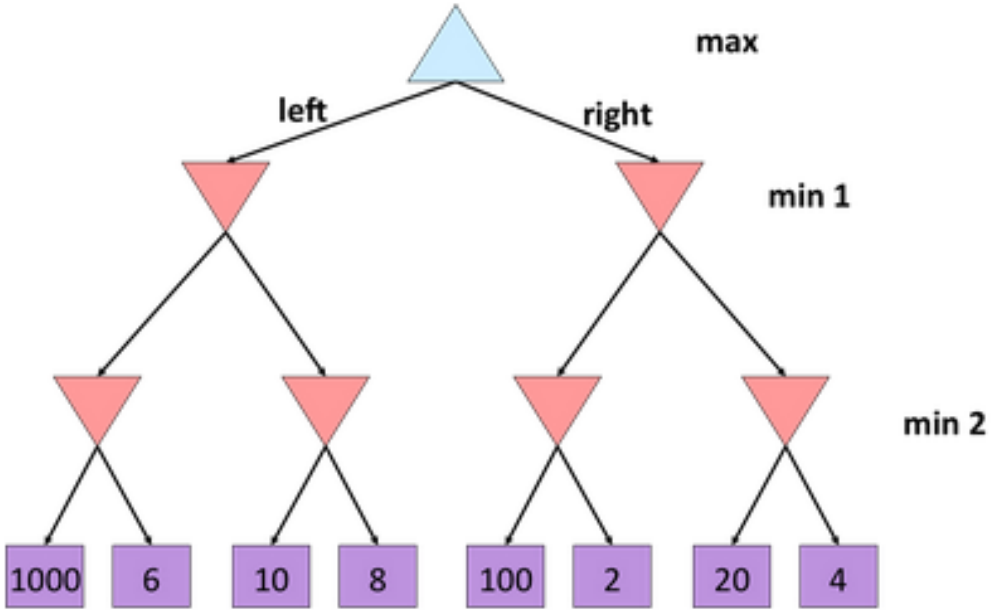
Danger of replanning agents



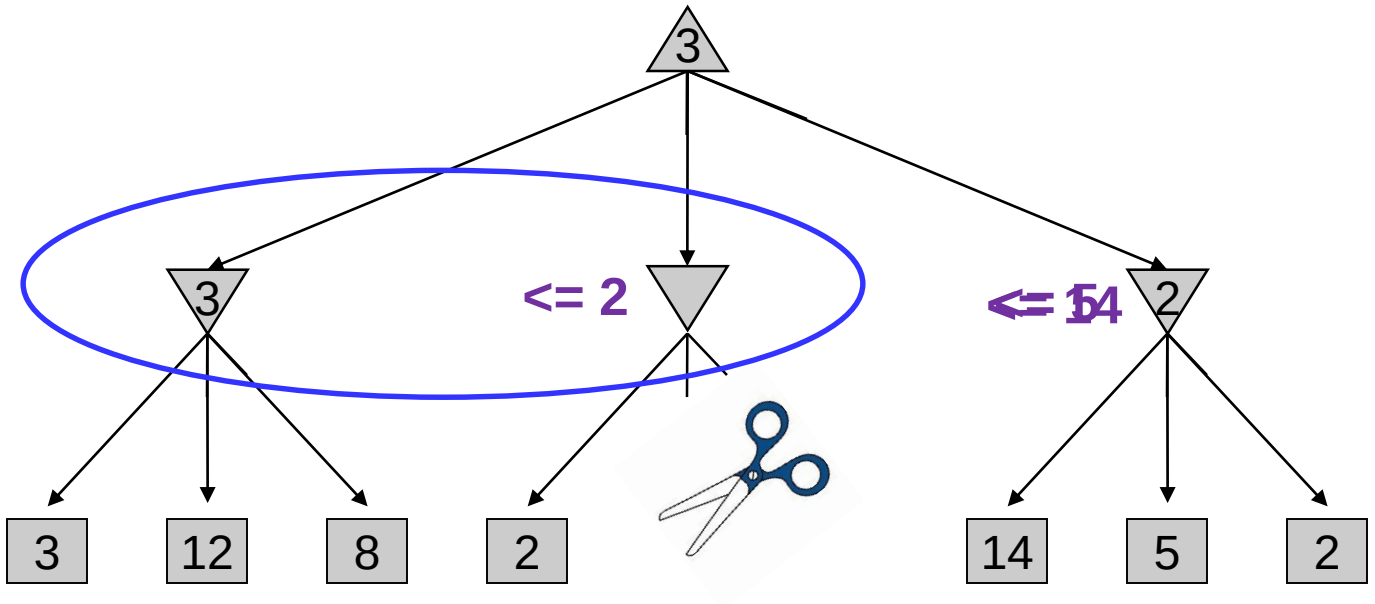
- He knows his score will go up by eating the dot now (west, east)
- He knows his score will go up just as much by eating the dot later (east, west)
- There are no point-scoring opportunities after eating the dot (within the horizon, two here)
- Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!

Quiz: collaboration

- By modeling each ghost as a minimizer, the “collaboration” behavior we saw before naturally arises from minimax.
- Below is an example of a game tree with two minimizer players (min 1 and min 2), and one maximizer player.



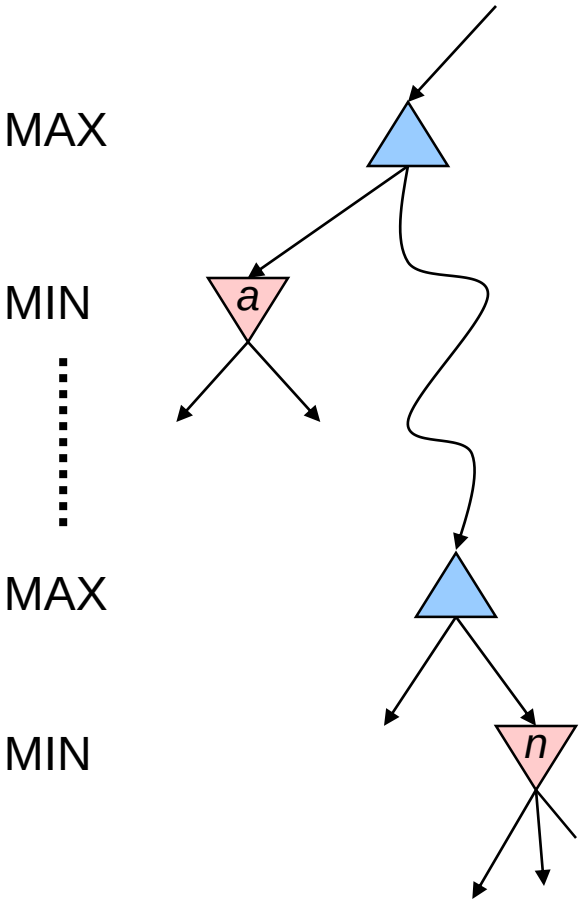
Pruning in Minimax Search



Here, as soon as a node we're minimizing dropped below the available max so far, we could stop.

Alpha-Beta Pruning

- General case (MIN version)
  - We're computing the MIN-VALUE at *n*
  - We're looping over *n*'s children
  - *n*'s value estimate is dropping
  - Who cares about n's value? MAX
  - Let *a* be the best value MAX can get at any choice point along the current path from the root
  - If *n* becomes worse than *a*, MAX will avoid it, so can stop considering *n*'s other children
- MAX version is symmetric

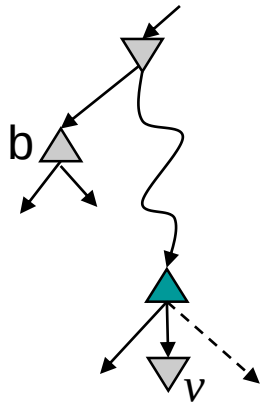


Alpha-Beta Pseudocode

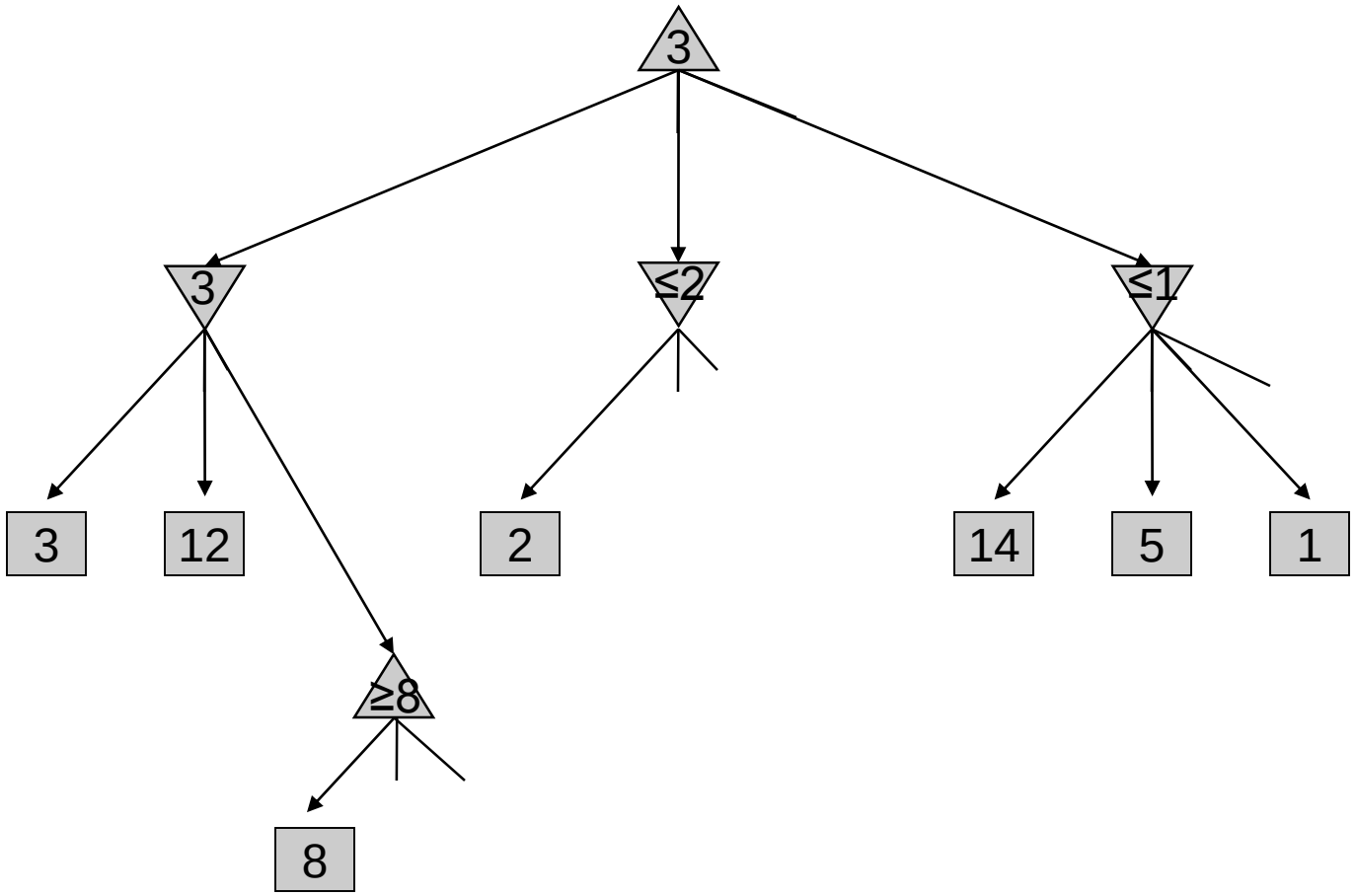
```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
    return v
```

```
function MAX-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

If so large that MIN prefers β elsewhere in the tree, then stop.



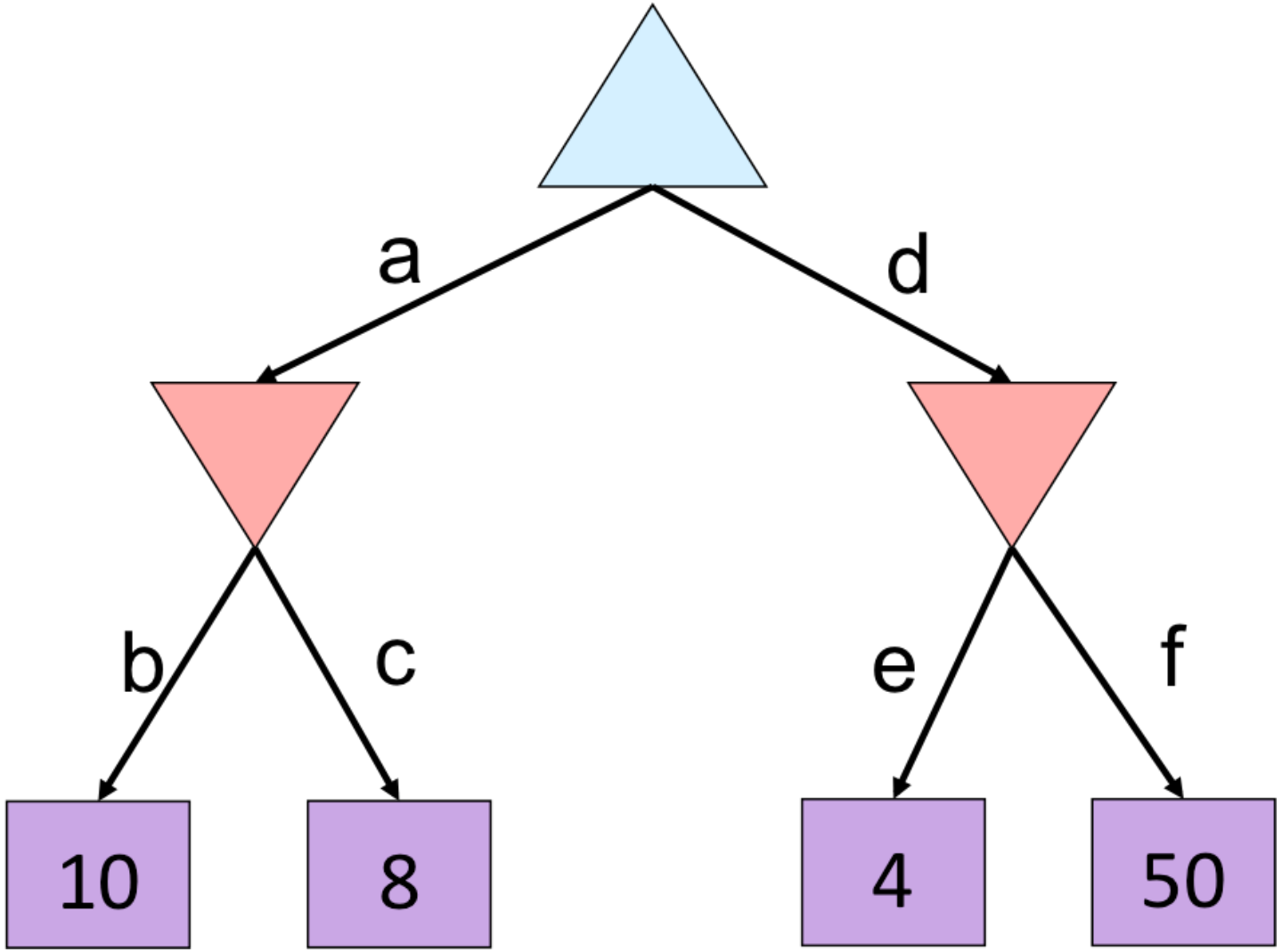
Alpha-Beta Pruning Example



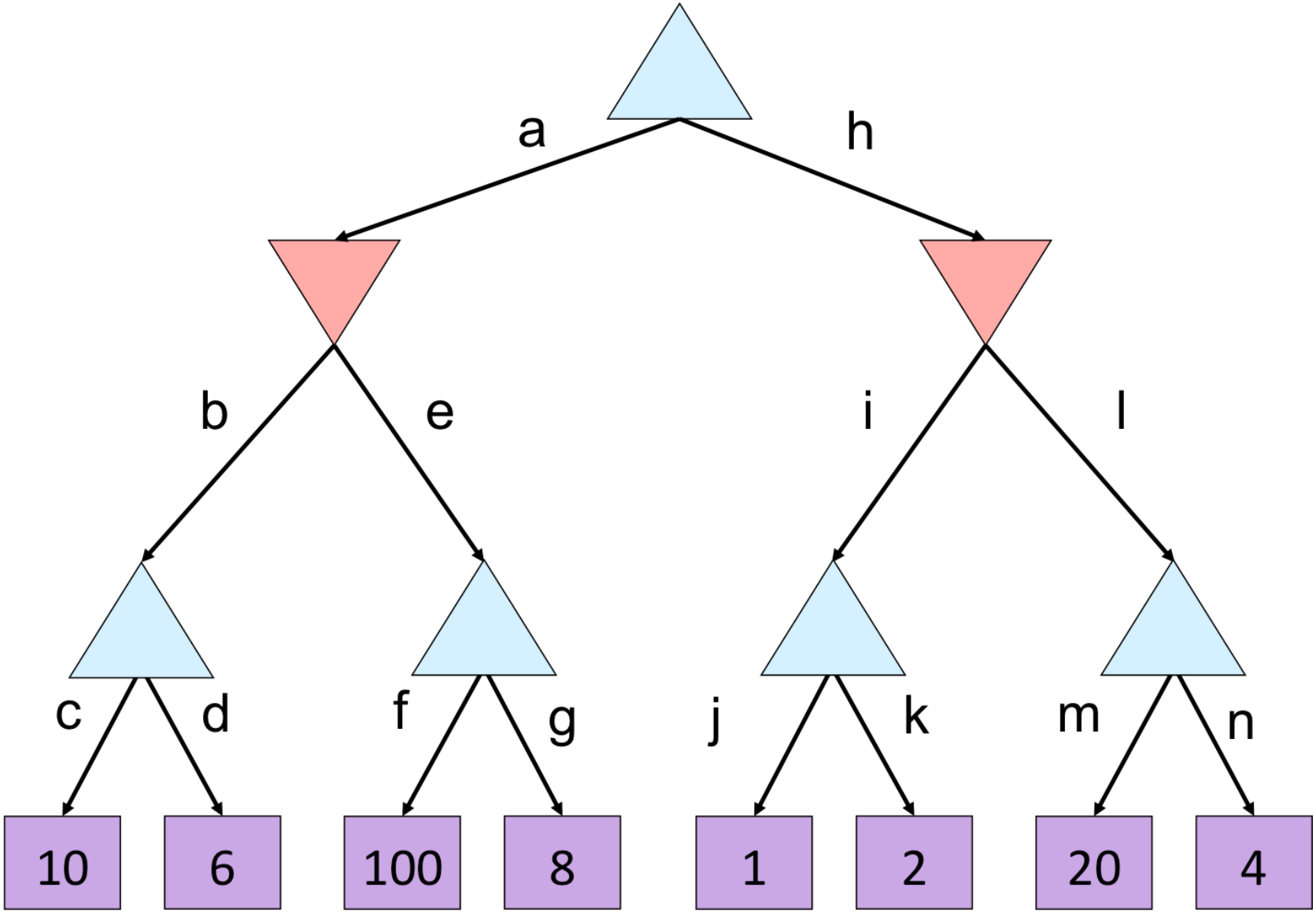
Alpha-Beta Pruning Properties

- This pruning has no effect on final result at the root
- Values of intermediate nodes might be wrong!
  - Important: children of the root may have the wrong value
- Good child ordering improves effectiveness of pruning
- With “perfect ordering”:
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…
- This is a simple example of metareasoning (computing about what to compute)

Quiz: alpha-beta pruning



Quiz: alpha-beta pruning



Next time: Uncertainty!

- What if some other agents are not necessarily adversaries?
  - Indifferent to you – e.g., a roll of a die
  - Inept adversary that makes mistakes
- Where do the terminal utilities come from?