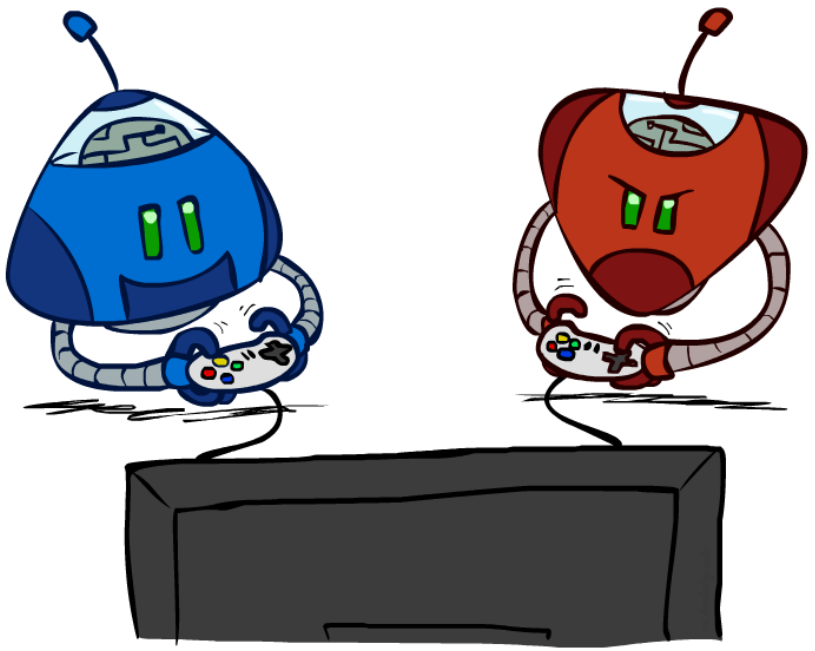
CS 4100 // artificial intelligence

instructor: byron wallace



Adversarial Search

Attribution: many of these slides are modified versions of those distributed with the *UC Berkeley CS188* materials
Thanks to John DeNero and Dan Klein

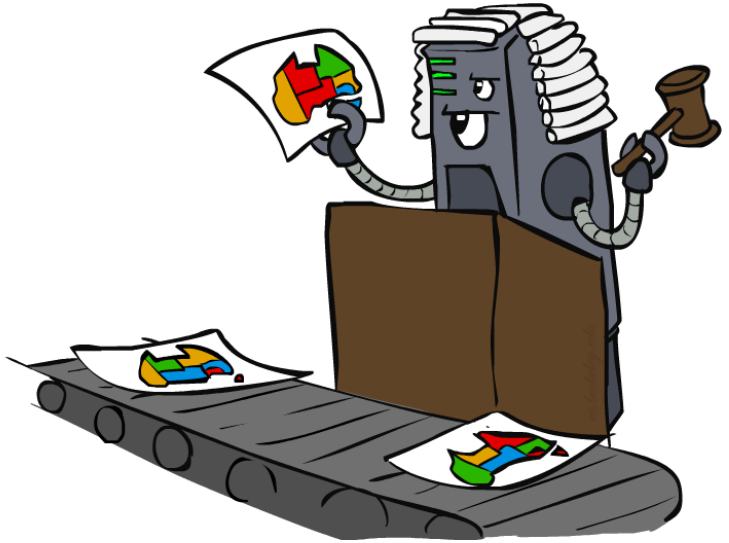
But first… wrap up on CSPs!

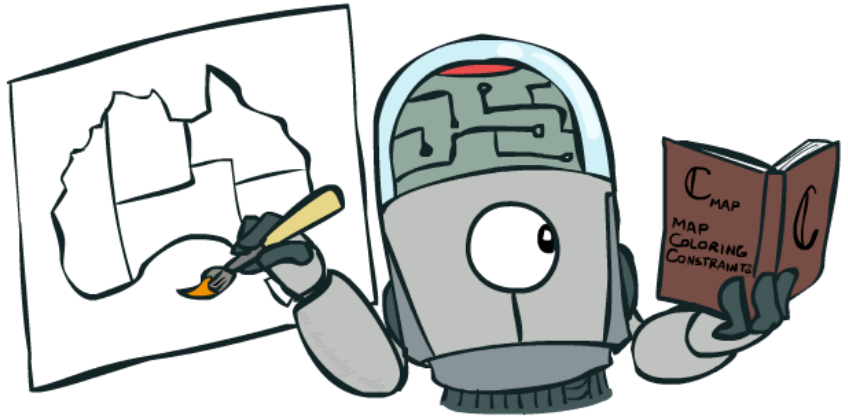
Reminder: CSPs

Constraint satisfaction problems (CSPs):

- A special subset of search problems
- State is defined by variables X_i with values from a domain D (sometimes D depends on i)
- *Goal test* is a set of constraints specifying allowable combinations of values for subsets of variables

Last time we saw how to solve these using ARC3





Exploiting structure in CSPs

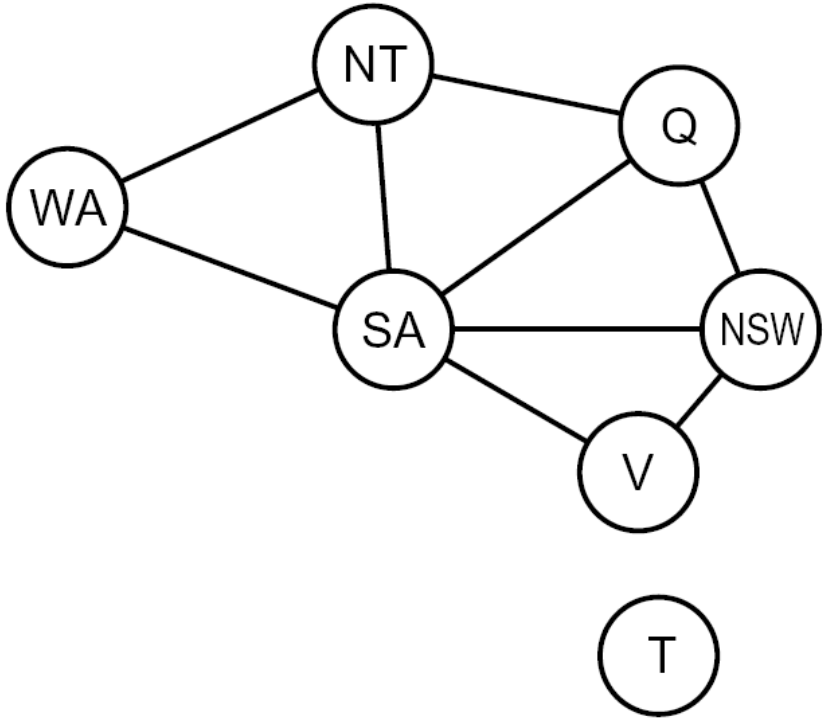
Extreme case: independent subproblems

- Example: Tasmania and mainland do not interact

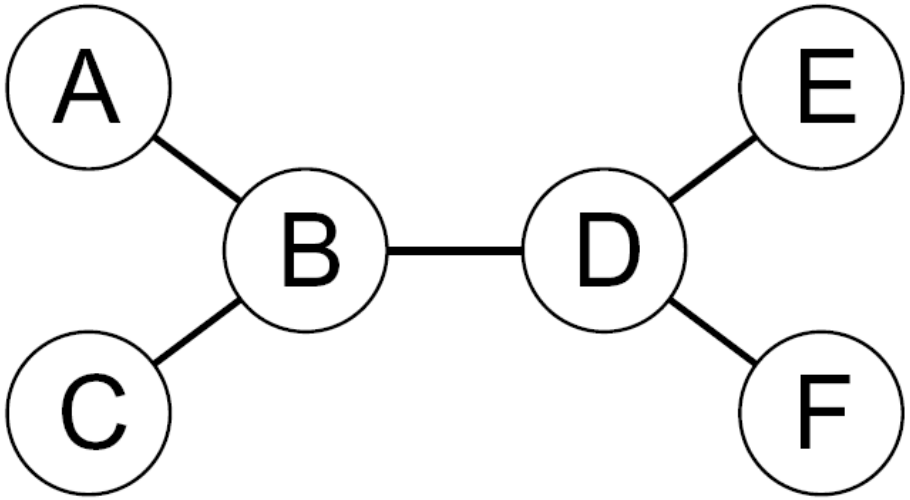
Independent subproblems are identifiable as connected components of constraint graph

Suppose a graph of n variables can be broken into subproblems of only c variables:

- Worst-case solution cost is $O((n/c)(d^c))$, linear in n
- E.g., n = 80, d = 2, c =20 (so 4 problems of size 20)
- 2^{80} = 4 billion years at 10 million nodes/sec
- $(4)(2^{20})$ = 0.4 seconds at 10 million nodes/sec



Tree-structured CSPs



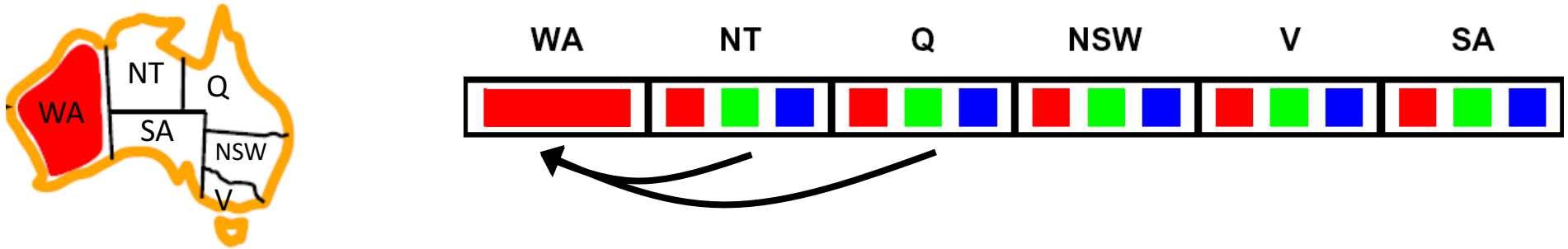
Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time

- Compare to general CSPs, where worst-case time is $O(d^n)$

This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning

Arc consistency

An arc X → Y is **consistent** iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

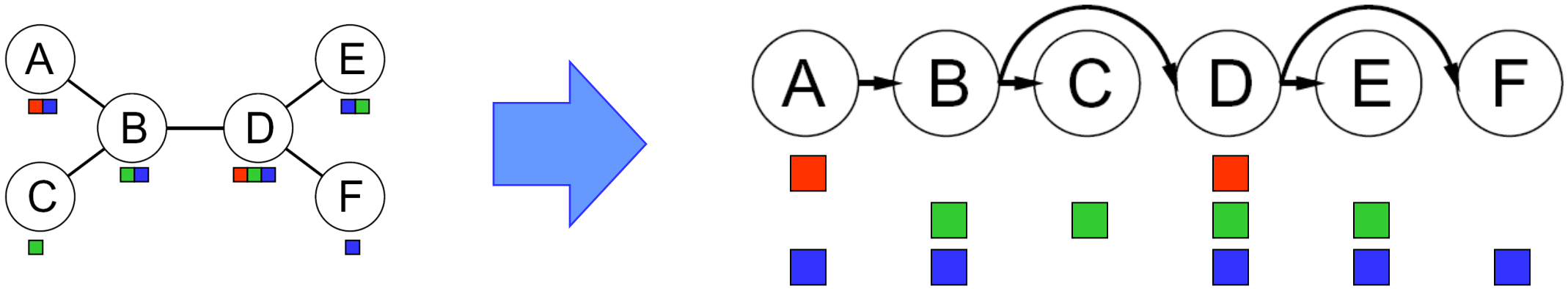


Forward checking: Enforcing consistency of arcs pointing to each new assignment

Tree-structured CSPs

Algorithm for tree-structured CSPs:

Order: Choose a root variable, order variables so that parents precede children



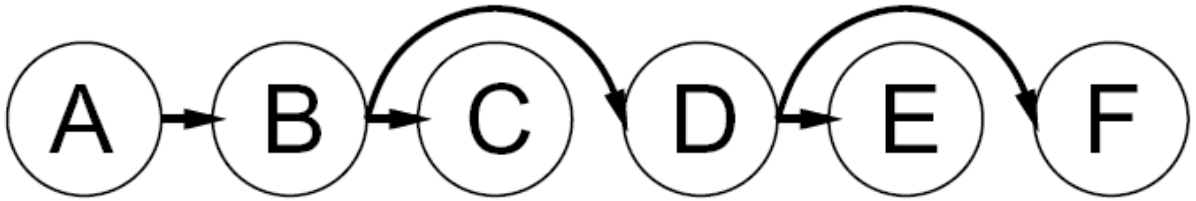
Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)

Runtime: O(n d^2) (why?)

Tree-structured CSPs

Claim 1: After backward pass, all root-to-leaf arcs are consistent

Proof: Each X→Y was made consistent at one point and Y's domain could not have been reduced thereafter (because Y's children were processed before Y)



Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack

Proof: Induction on position

How does this rely on the tree structure again?

Note: we'll see this basic idea again with Bayes' nets

Summary: CSPs

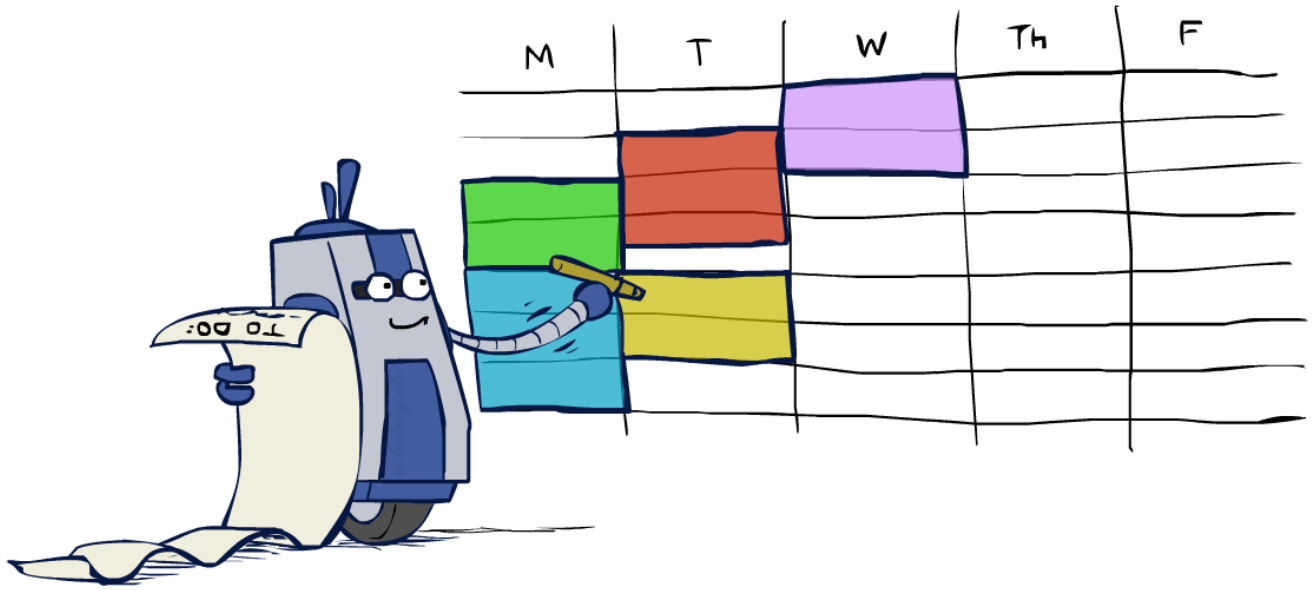
CSPs are a special kind of search problem:

- States are partial assignments
- Goal test defined by constraints

Basic solution: backtracking search

Speed-ups:

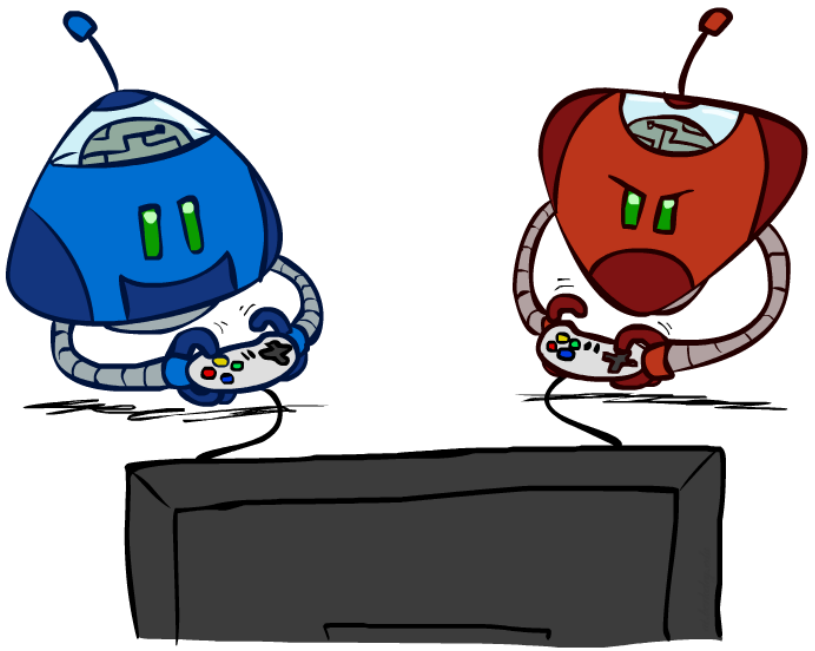
- Ordering
- Filtering
- Structure



Iterative min-conflicts is often effective in practice

CS 4100 // artificial intelligence

instructor: byron wallace

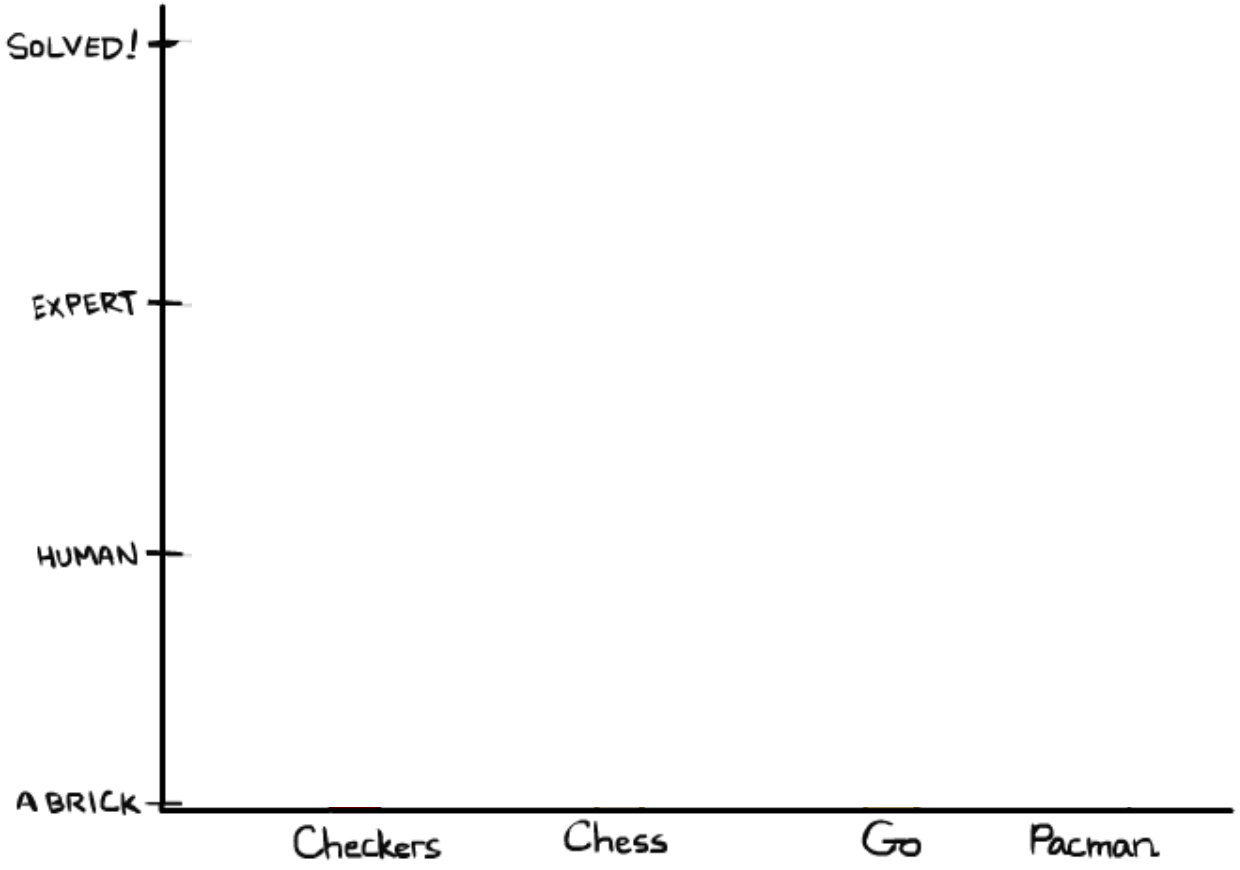


Adversarial Search

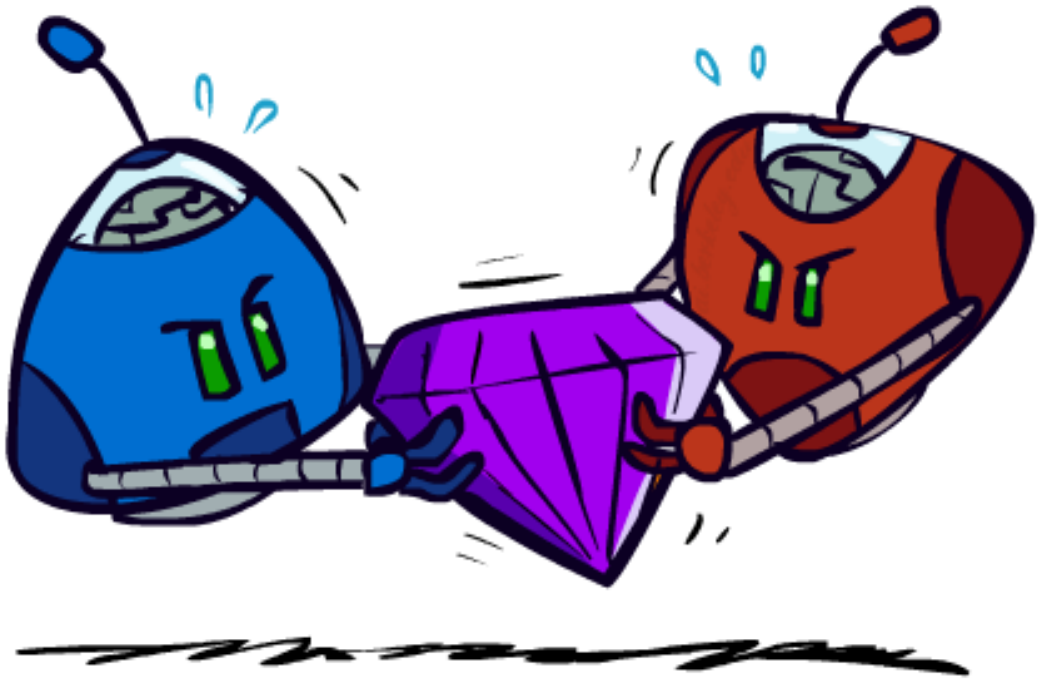
Attribution: many of these slides are modified versions of those distributed with the *UC Berkeley CS188* materials
Thanks to John DeNero and Dan Klein

Game playing state-of-the-art

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!
- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M 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 recently (2016!) defeated by machines; specifically **alphago!** Checkout https://deepmind.com/research/alphago/ for more
- **Pacman**



Adversarial games

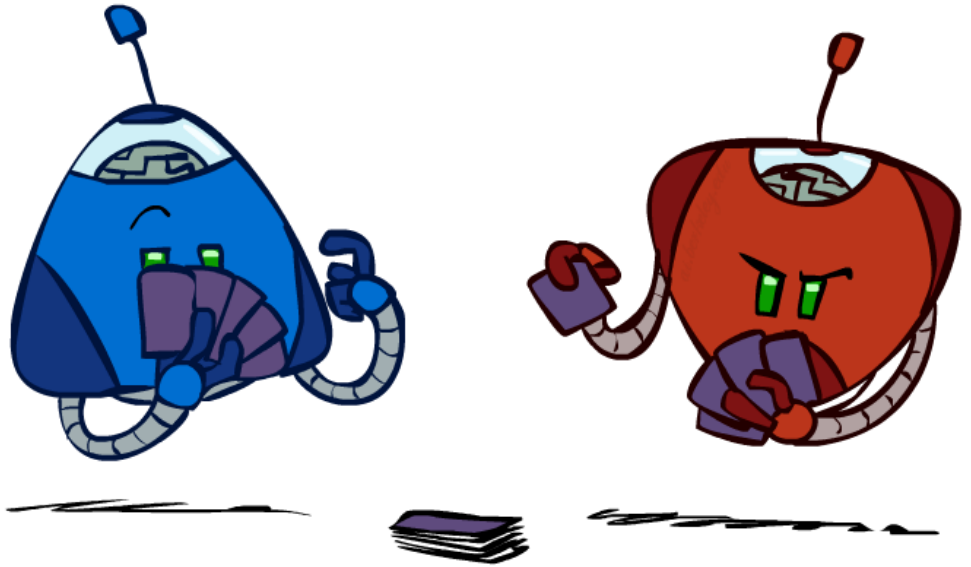


Types of games

- Many different kinds of games!

- Axes:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?

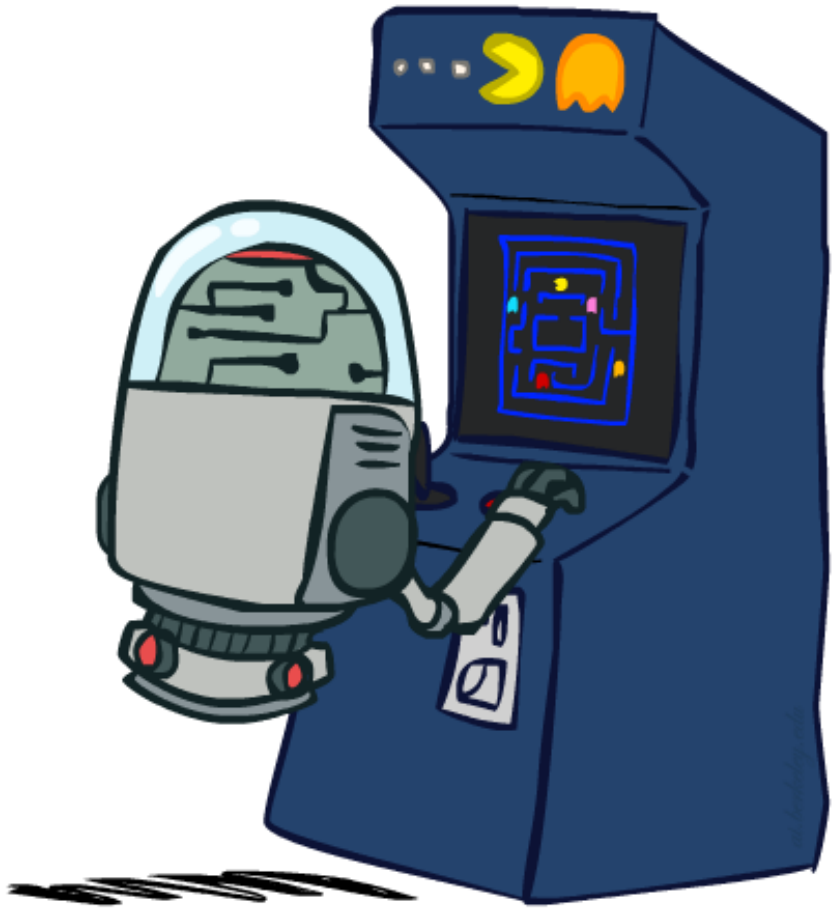
- Want algorithms for calculating a **strategy (policy)** which recommends a move from each state



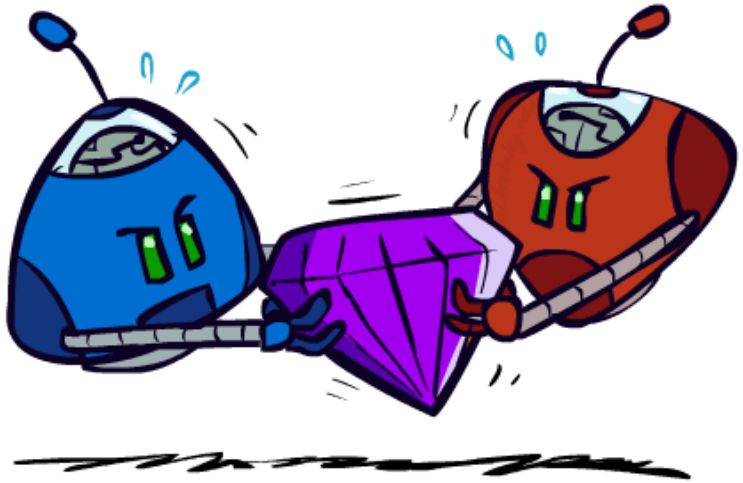
Deterministic games

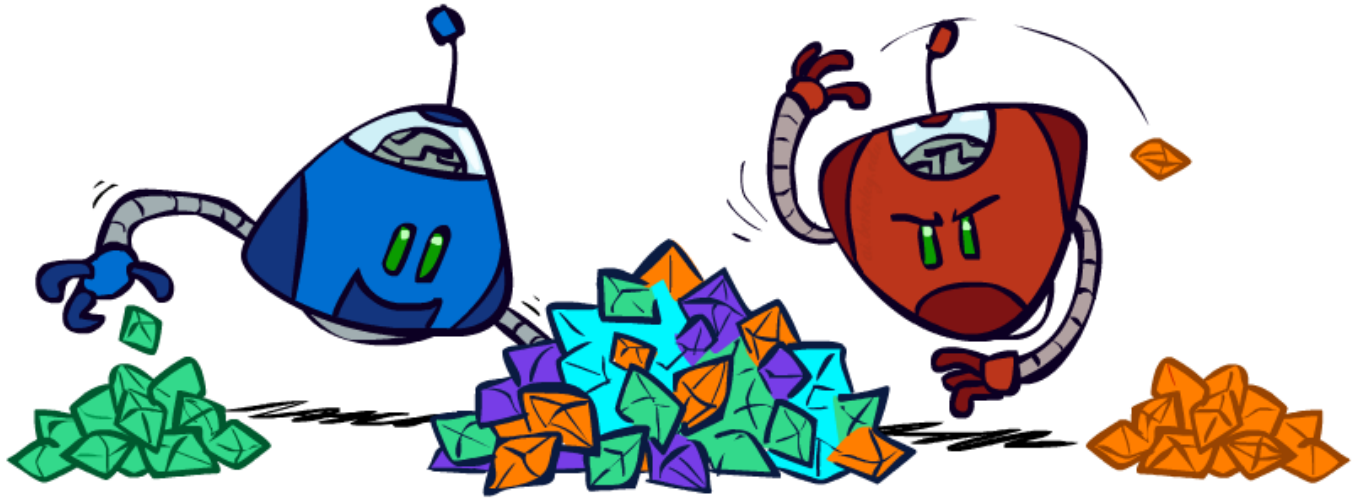
- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: S x A → S
  - Terminal Test: S → {t,f}
  - Terminal Utilities: S x P → R

- Solution for a player is a **policy**: S → A



Zero-Sum games





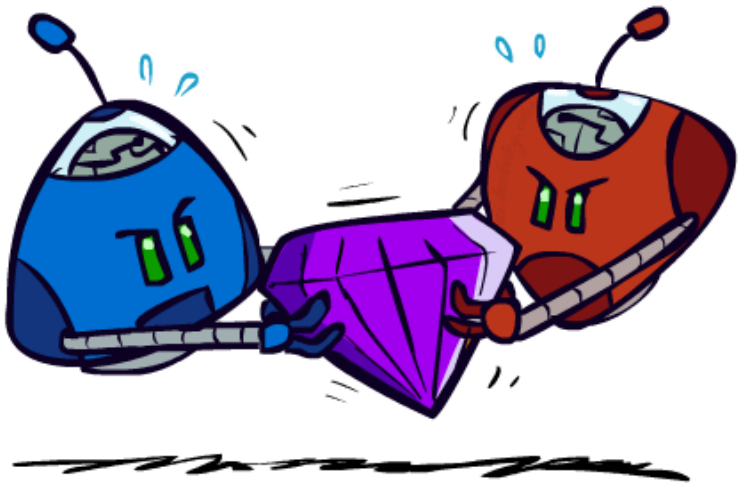
Zero-Sum Games

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

General Games

- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible
- More later on non-zero-sum games

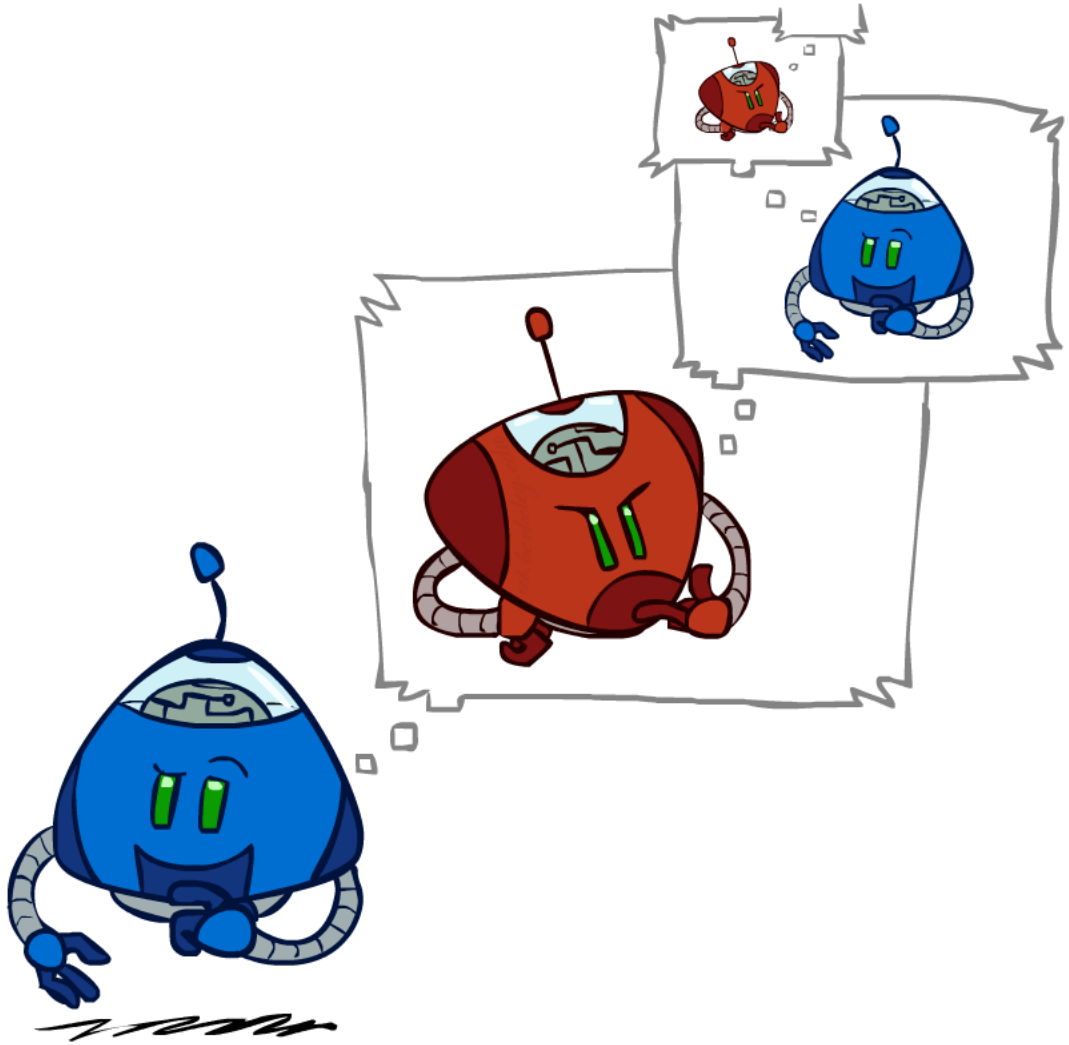
Zero-Sum games



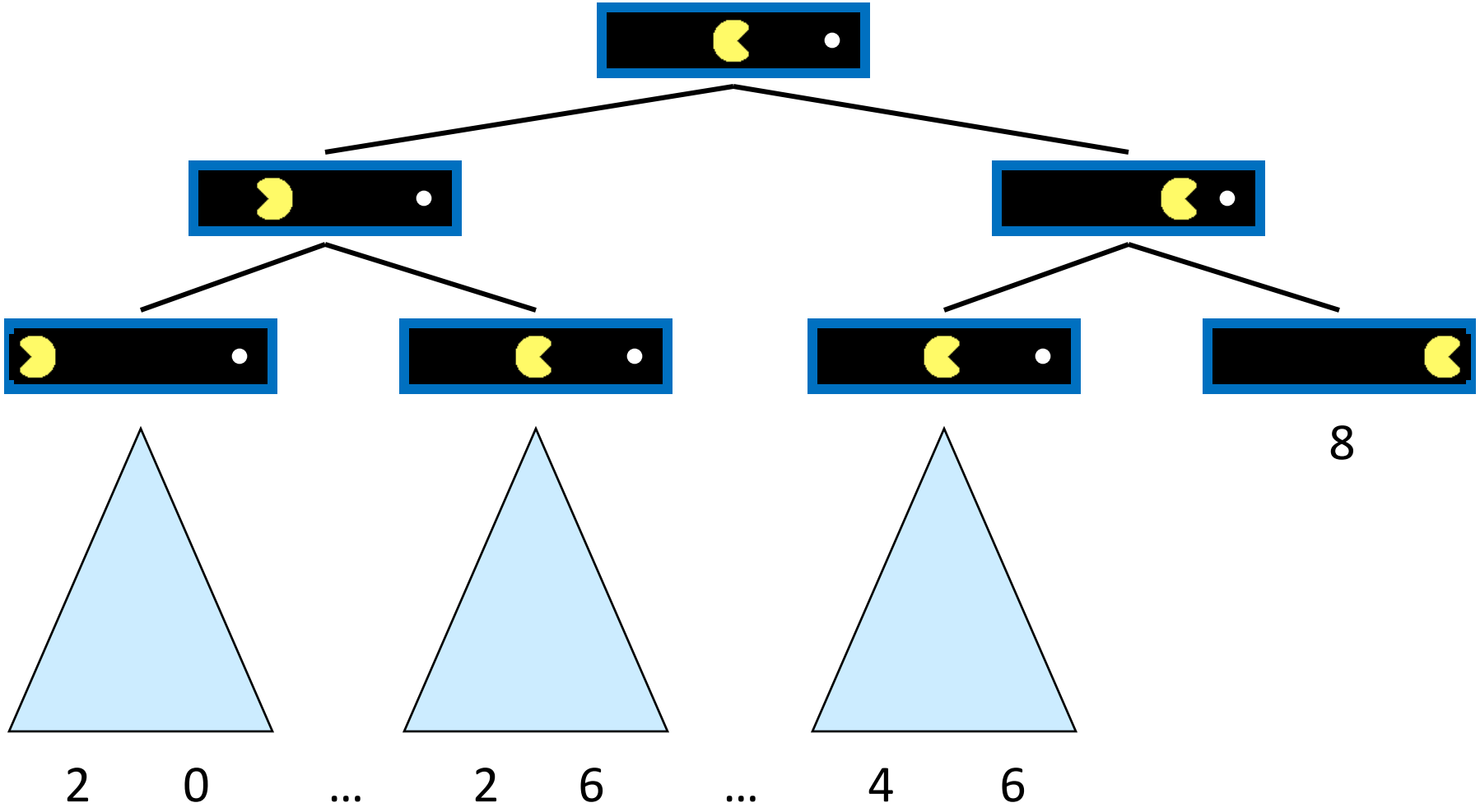
Zero-Sum Games

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

Adversarial search

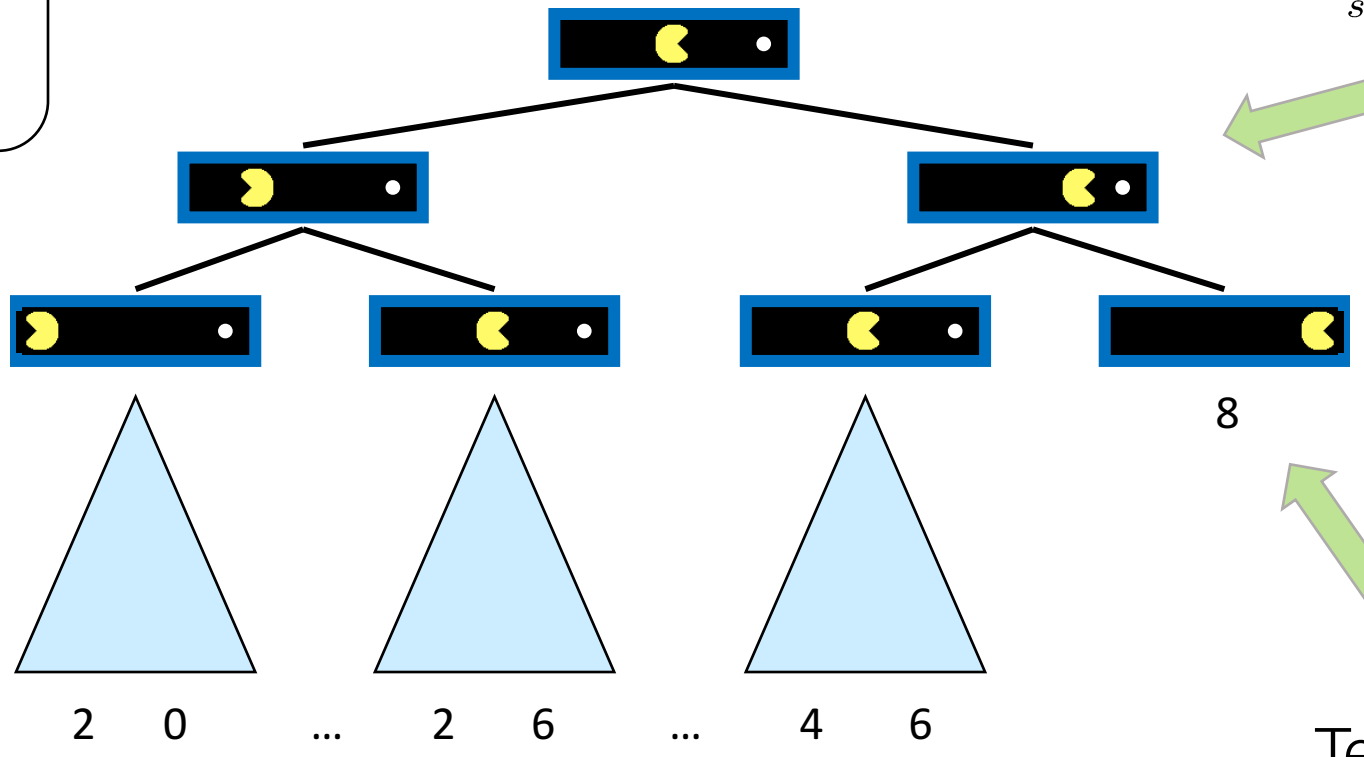


Single-agent trees



Value of a State

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



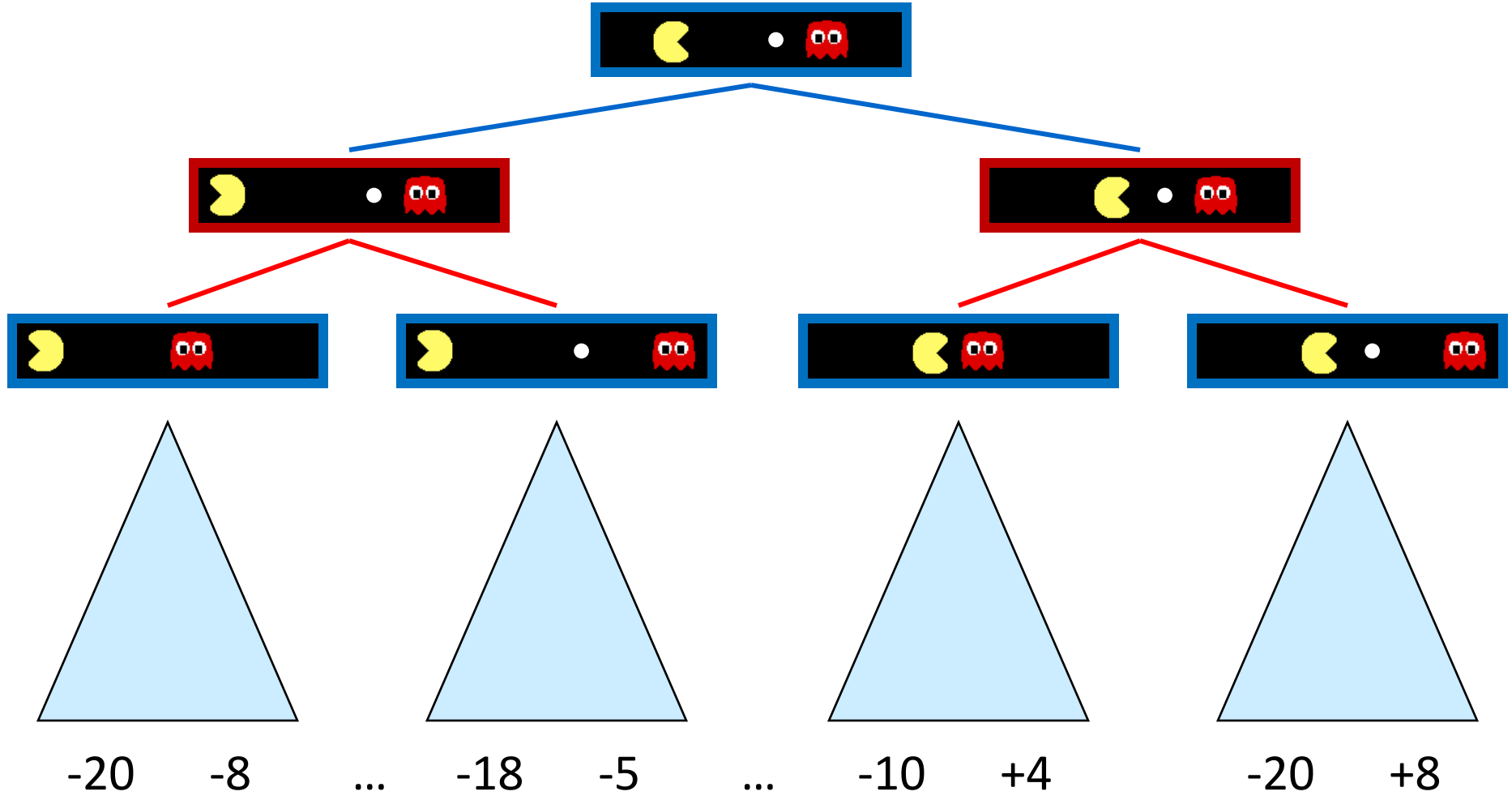
Non-Terminal States:

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

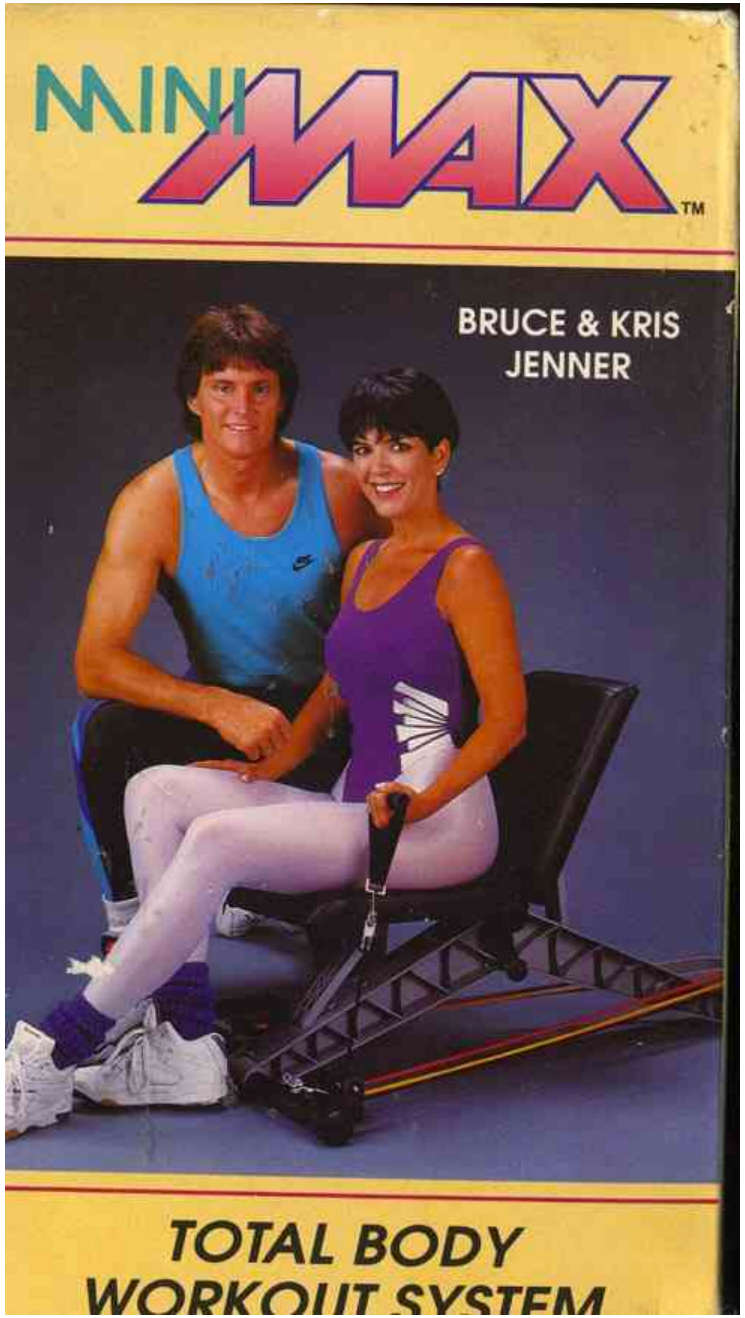
Terminal States:

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

Adversarial game trees



Minimax

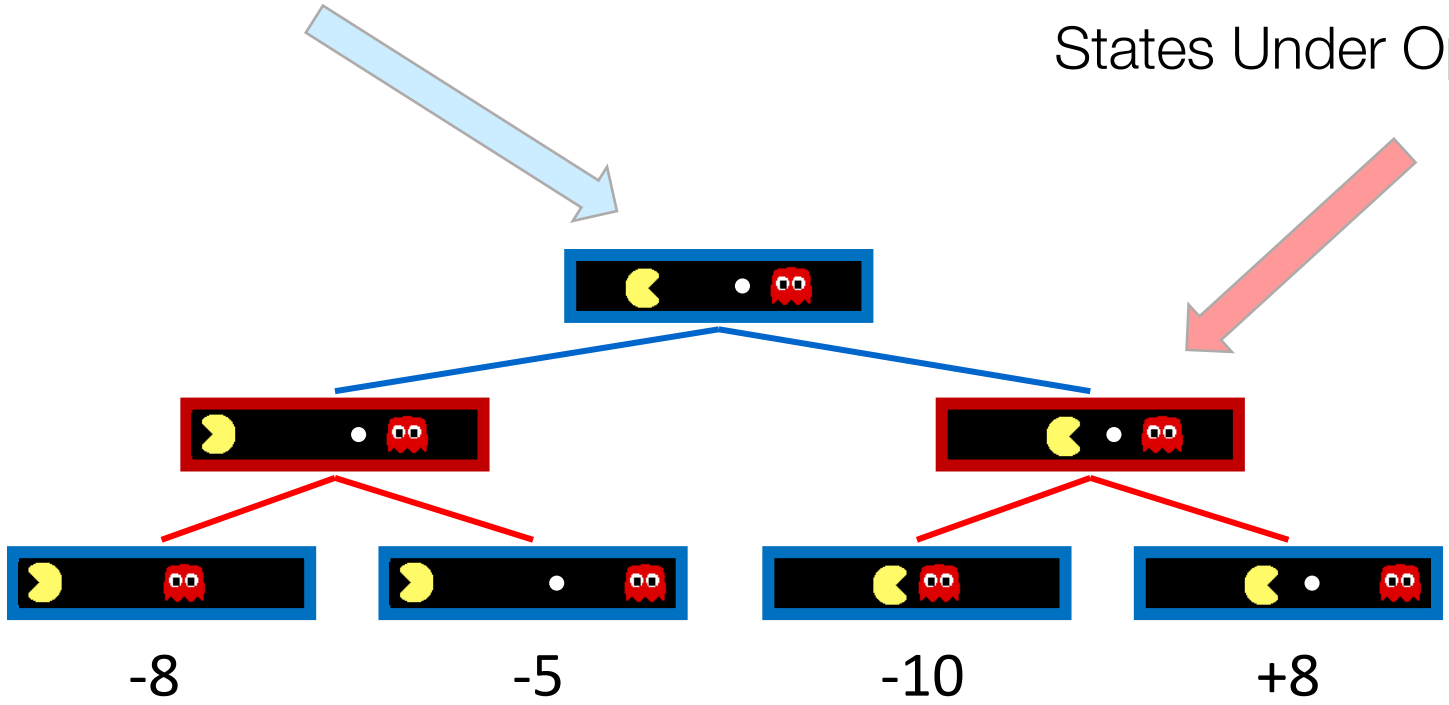


Minimax values

States Under Agent's Control:

Which way should pacman go???

States Under Opponent's Control:



Terminal States:

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

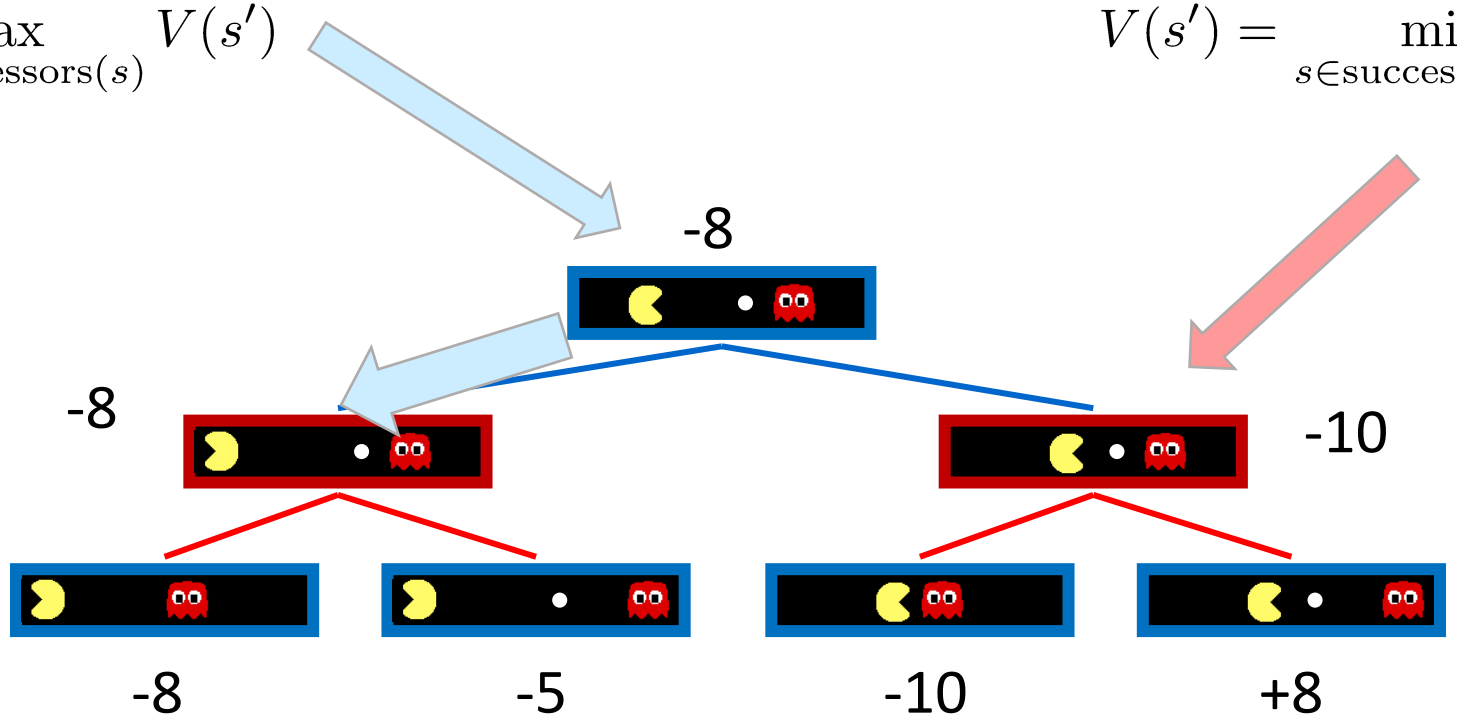
Minimax values

States Under Agent's Control:

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

States Under Opponent's Control:

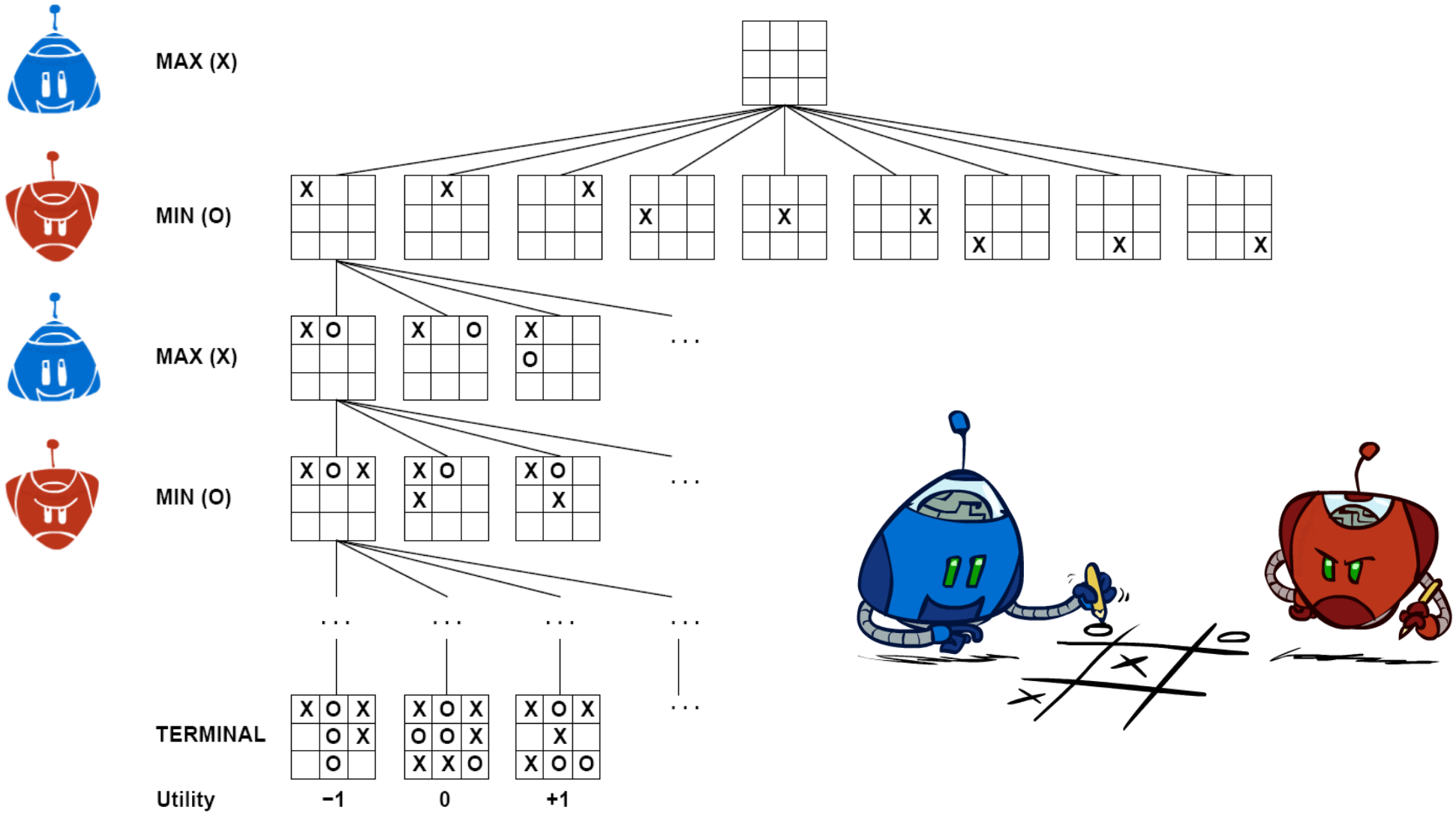
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

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

Tic-Tac-Toe game tree



Adversarial search (Minimax)

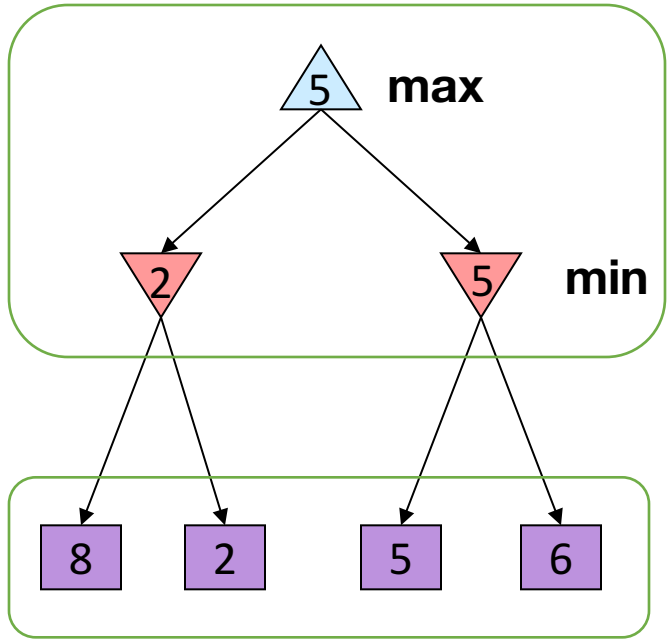
Deterministic, zero-sum games:

- Tic-tac-toe, chess, checkers
- One player maximizes result
- The other minimizes result

Minimax search:

- A state-space search tree
- Players alternate turns
- Compute each node's **minimax value:** the best achievable utility against a rational (optimal) adversary

Minimax values:
computed recursively

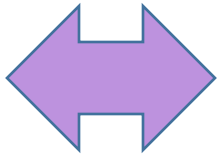


Terminal values:
part of the game

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

Minimax implementation

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



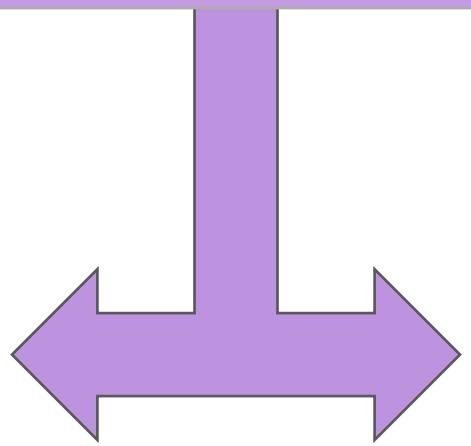
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

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

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

Minimax implementation (dispatch)

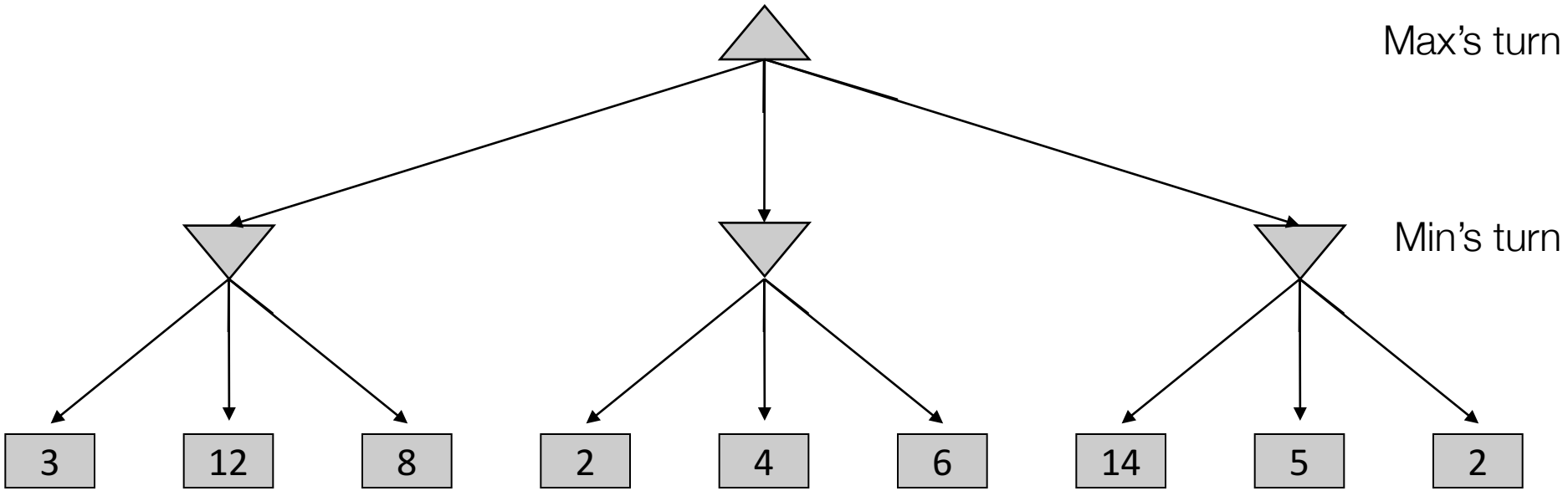
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```



```
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

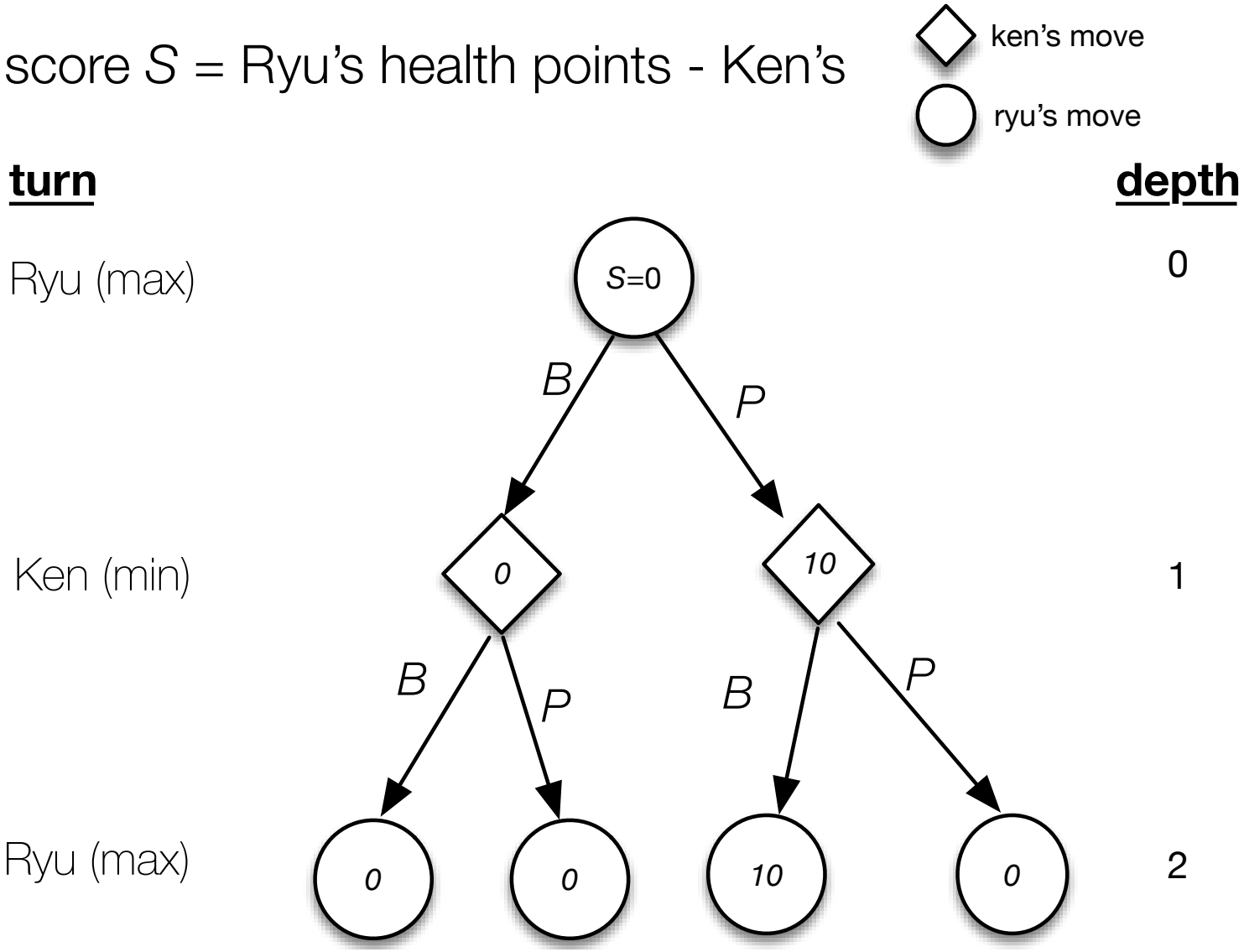


In class exercise on minimax

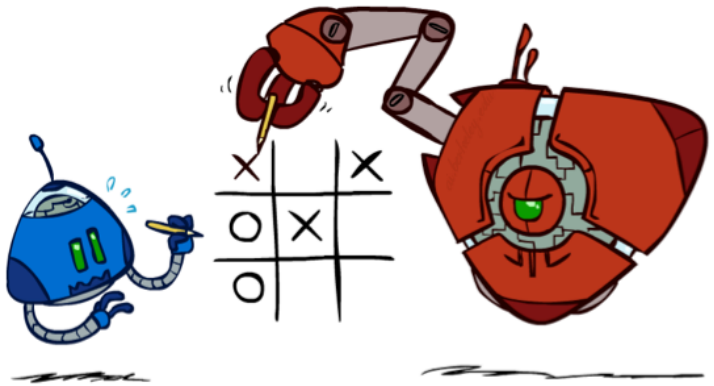
Note: assume Ryu gets first move!

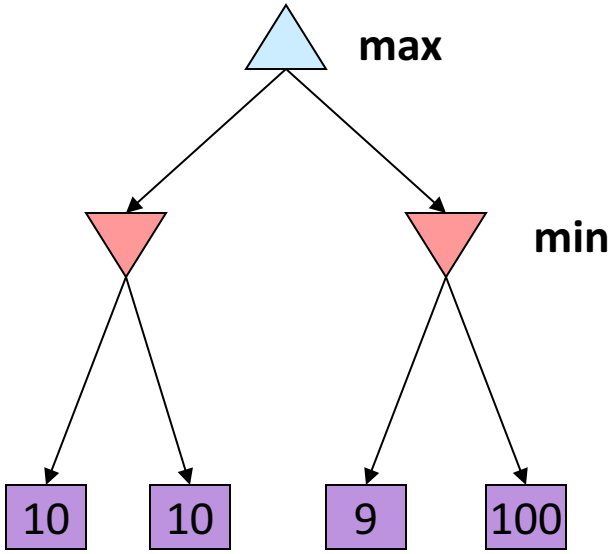
Shortcoming of minimax?

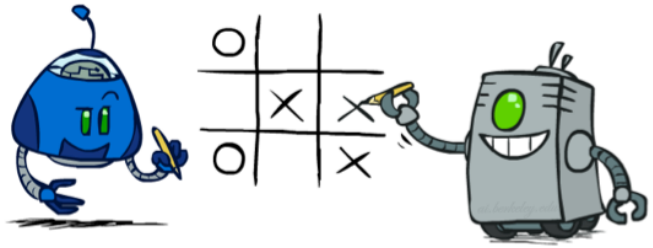
score *S* = Ryu's health points - Ken's



Minimax properties







Optimal against a perfect player. **Otherwise?**

We’ll address this issue next class!

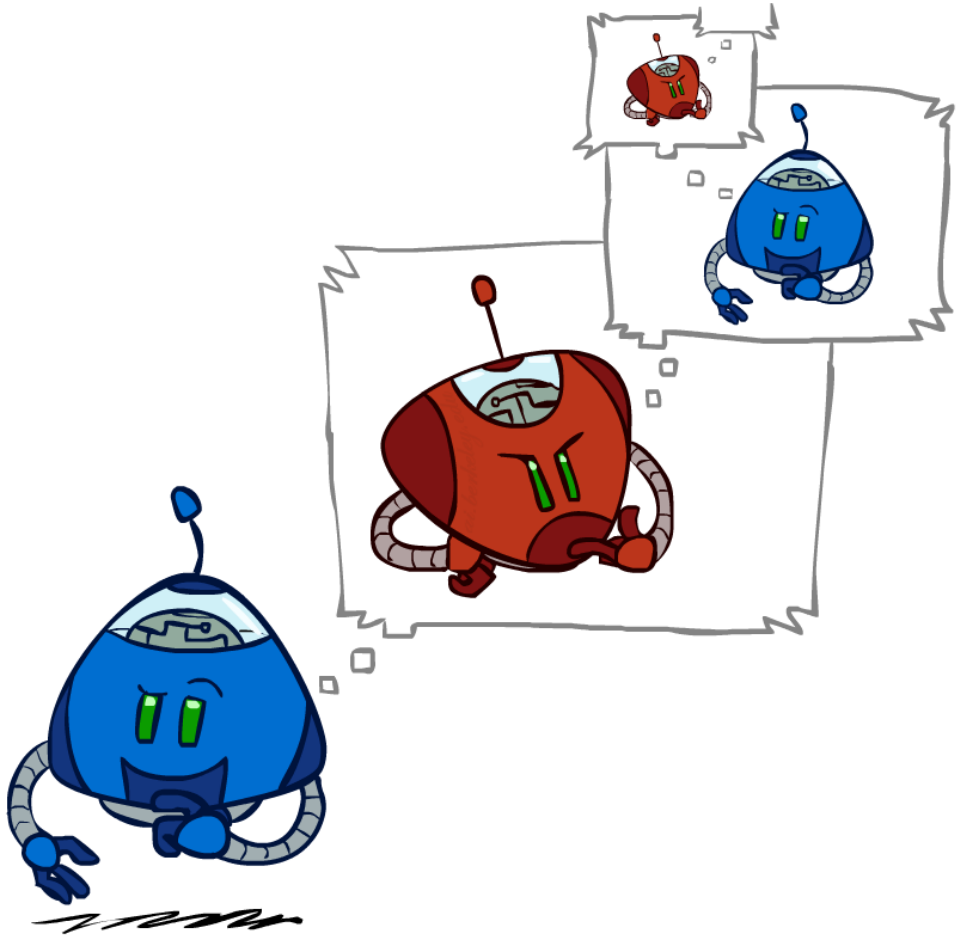
Minimax efficiency

How efficient is minimax?

- Just like (exhaustive) DFS
- Time: $O(b^m)$
- Space: O(bm)

Example: For chess, $b \approx 35$, $m \approx 100$

- Exact solution is completely infeasible
- But, do we need to explore the whole tree?



Resource limits

Resource limits

Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search

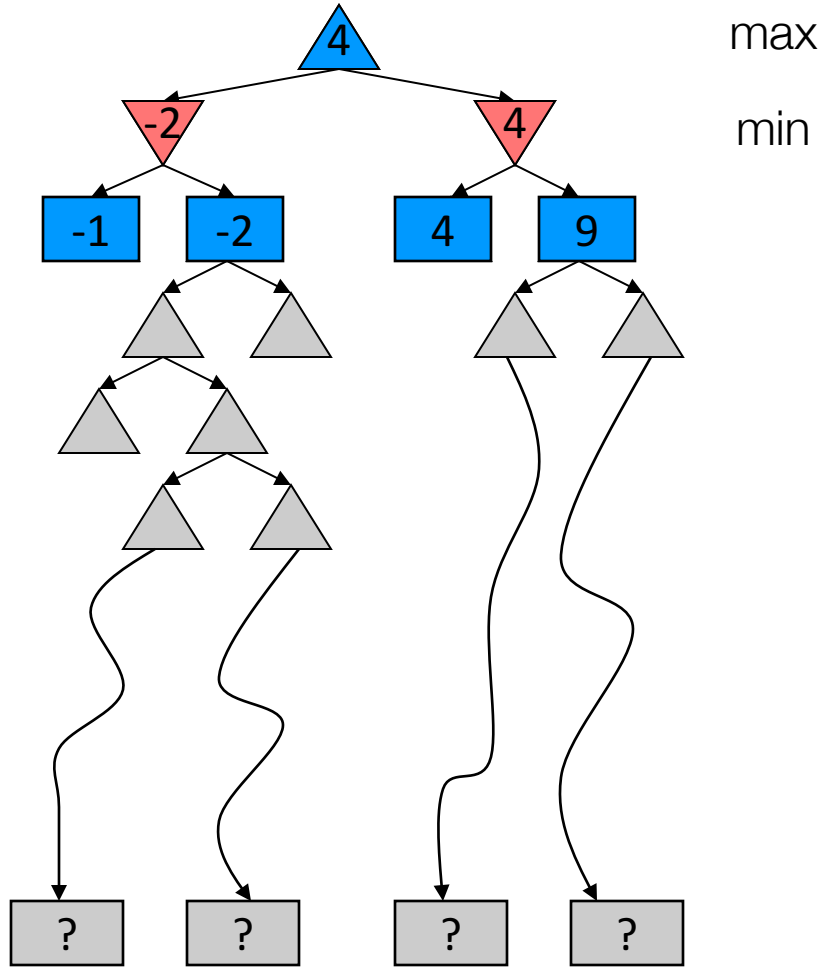
- Instead, search only to a limited depth in the tree
- Replace terminal utilities with an **evaluation function** for non-terminal positions

Example:

- Suppose we have 100 seconds, can explore 10K nodes / sec
- So can check 1M nodes per move
- α-β (we will see in a bit) reaches about depth 8 – decent chess program

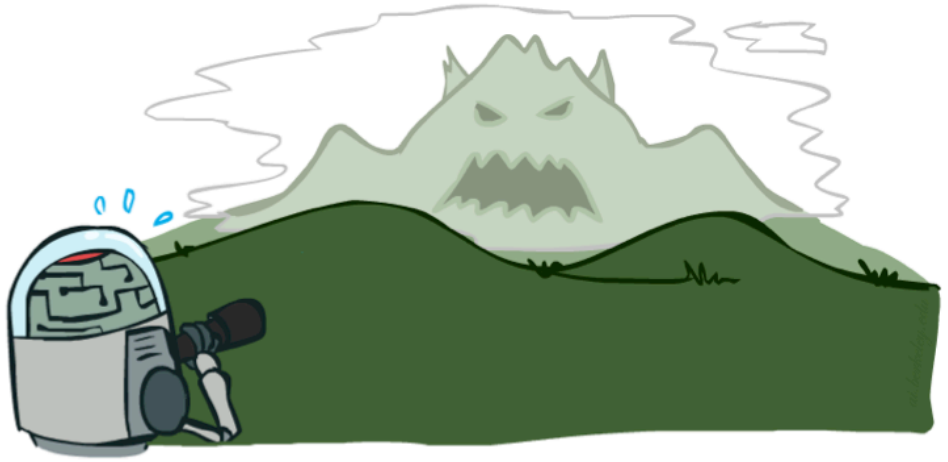
Guarantee of optimal play is gone

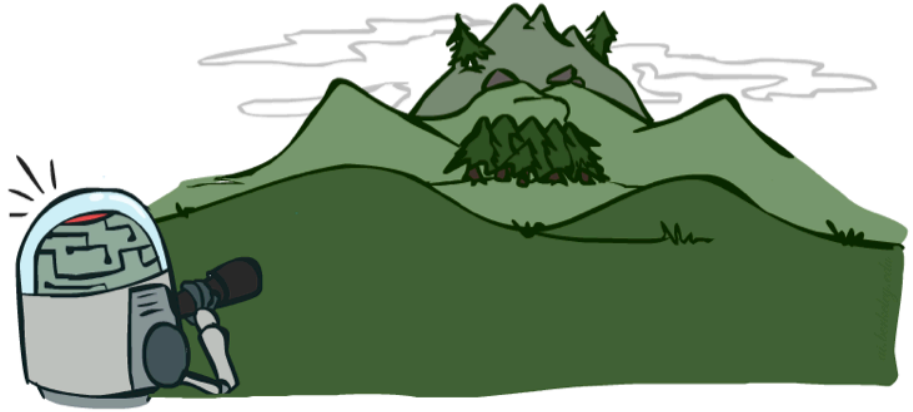
More plies makes a BIG difference



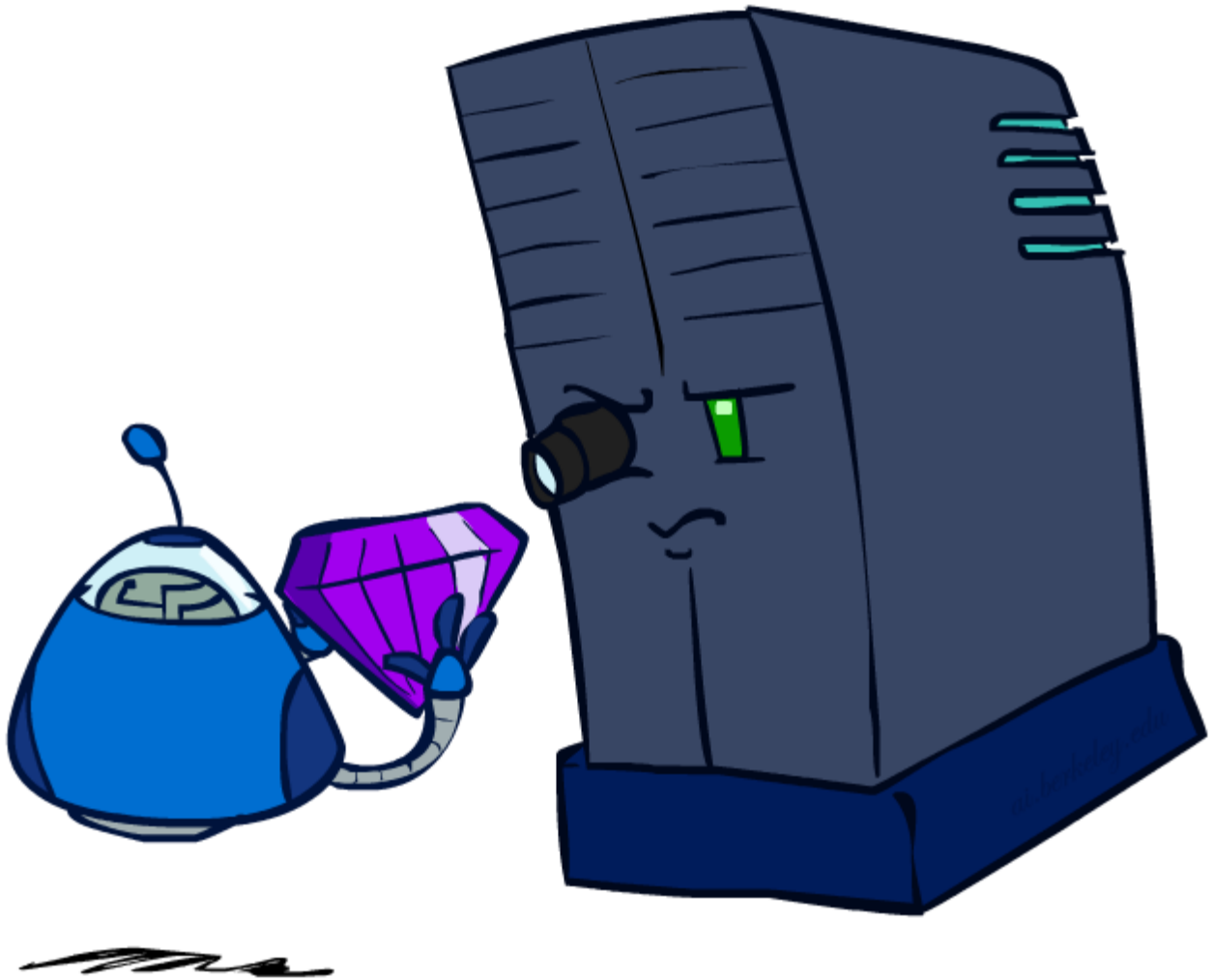
Depth matters

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



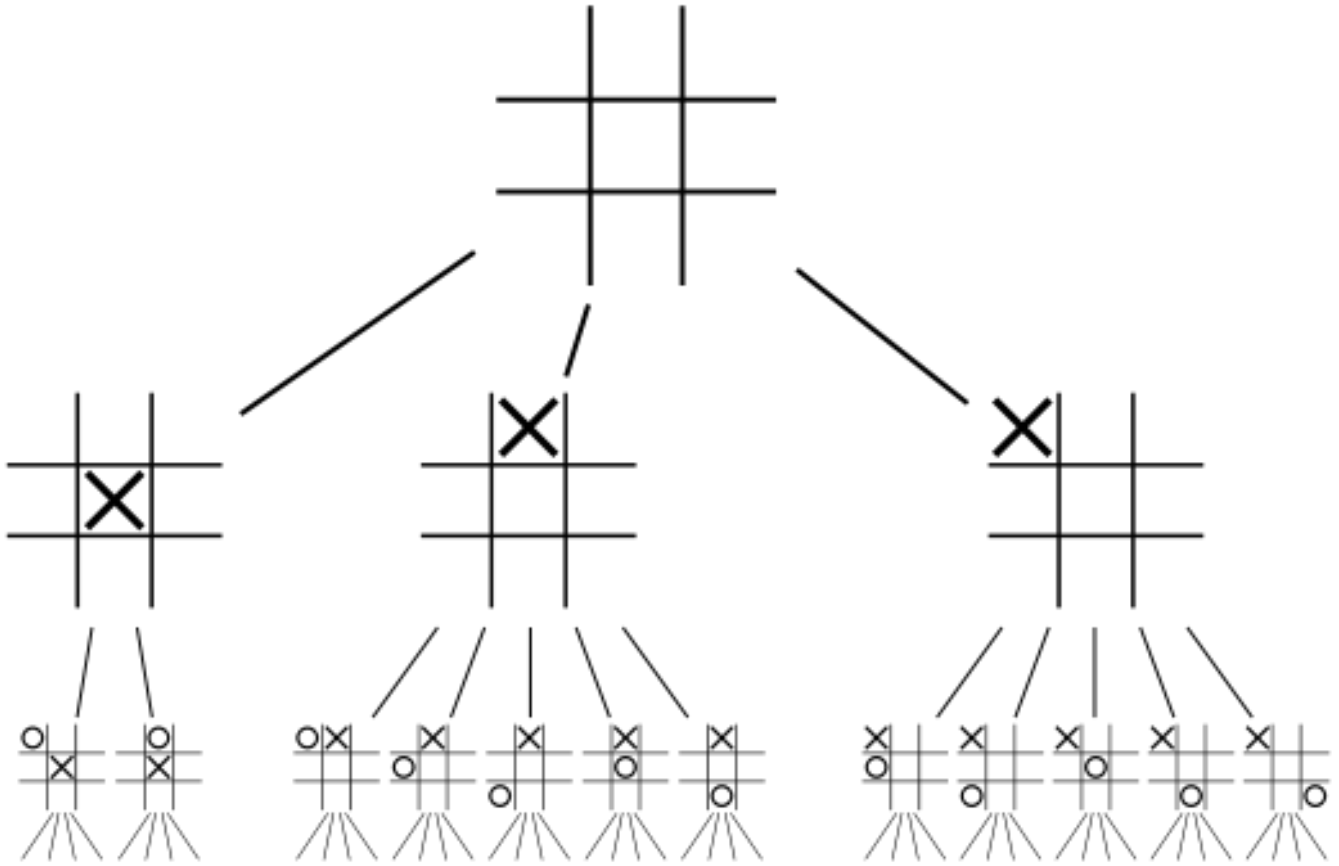


Evaluation functions



Tic-Tac-Toe

- Reasonable evaluation function?



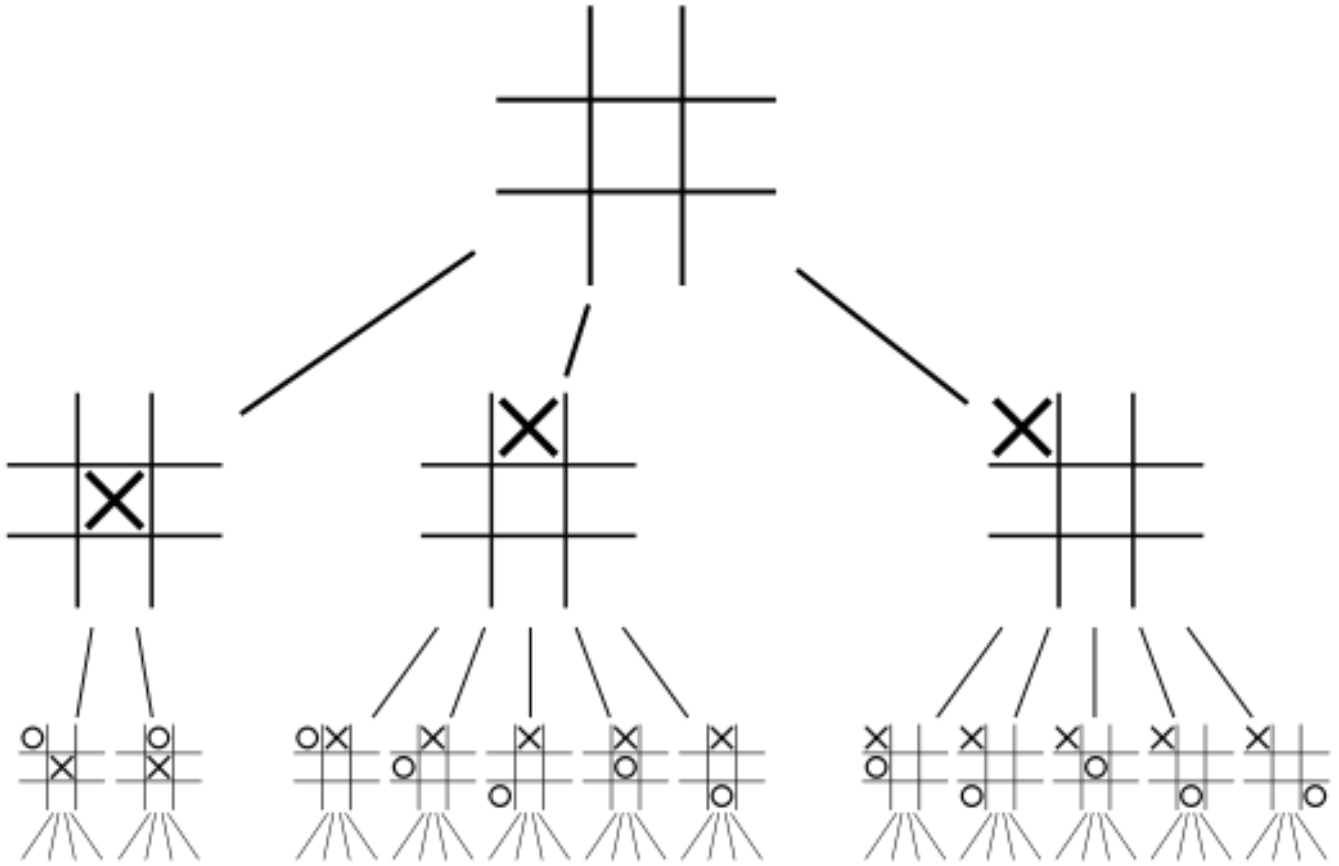
Tic-Tac-Toe

- Reasonable evaluation function?

Perhaps:

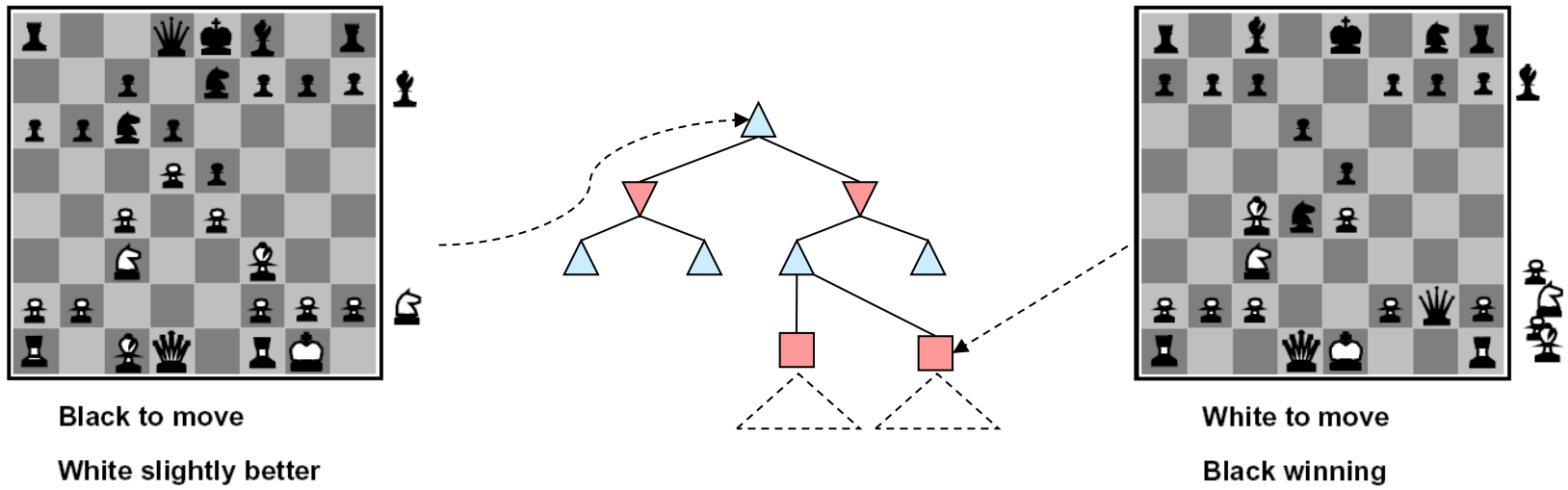
[# of 3-lengths open for me] –

[# of 3-lengths open for you]



Evaluation functions

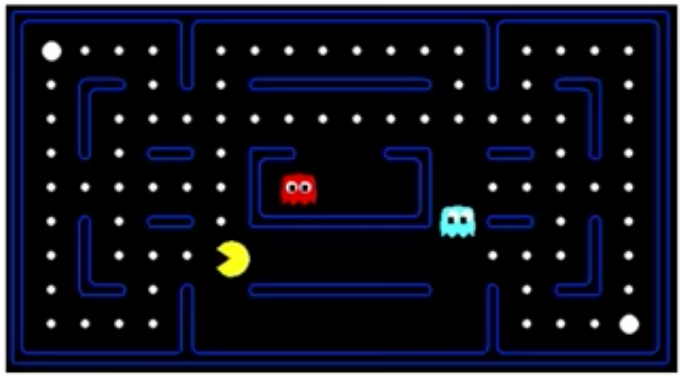
Evaluation functions score non-terminals in depth-limited search



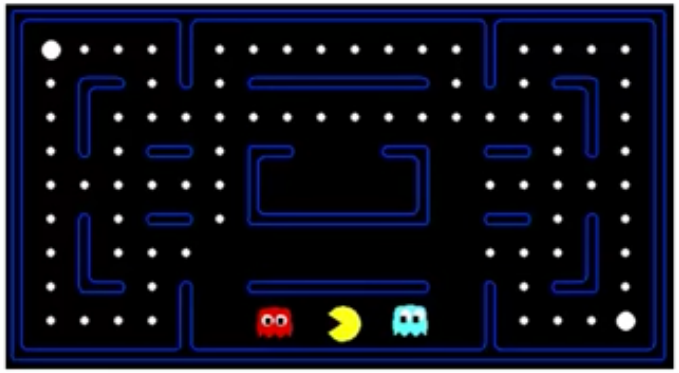
Ideal function: returns the actual minimax value of the position
In practice: typically weighted linear sum of features:

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

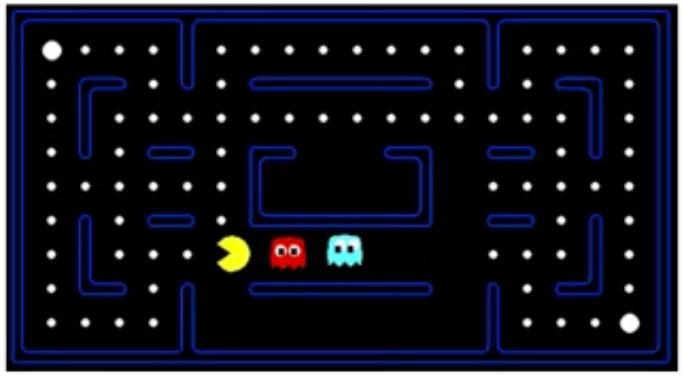
e.g. $f_1(s)$ = (num white queens – num black queens), etc.



A

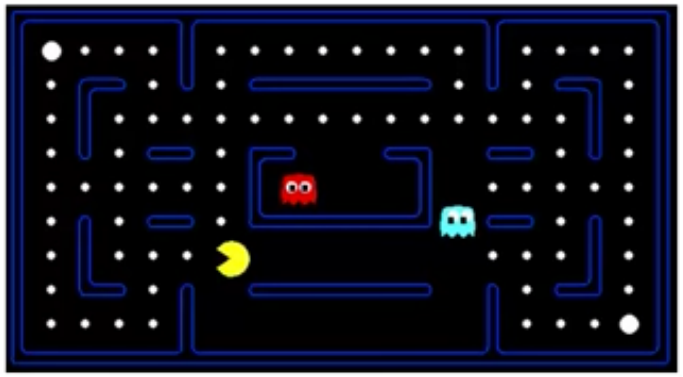


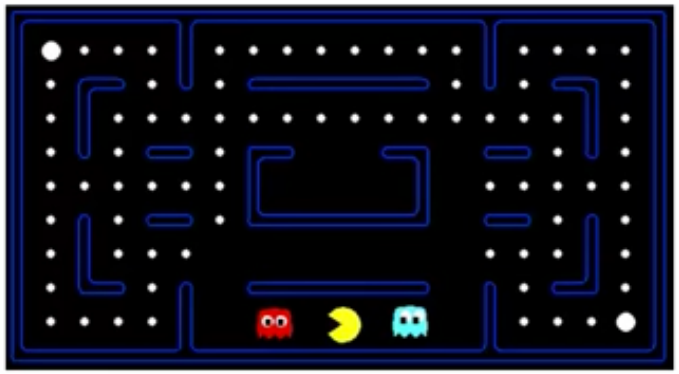
B

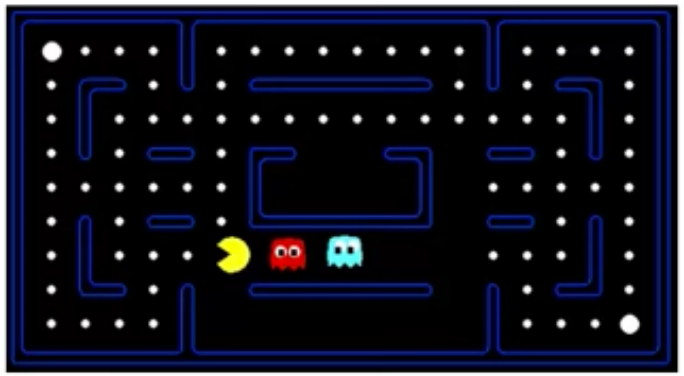


C

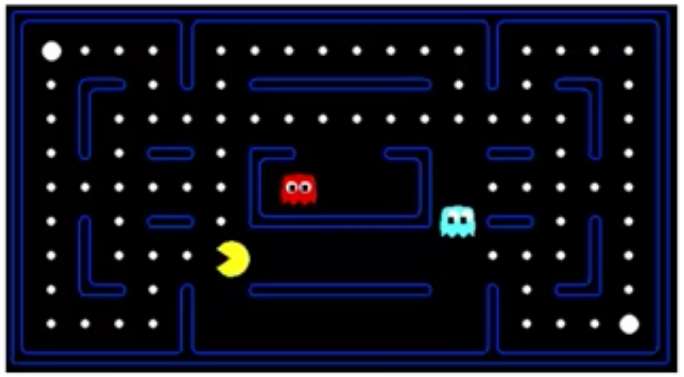
Board snapshots: need scalars from eval function that effectively rank these

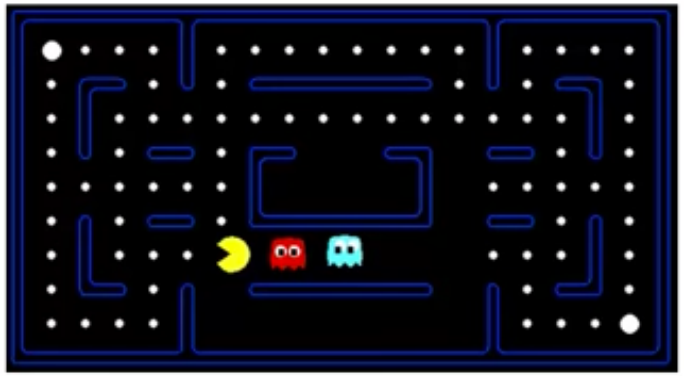


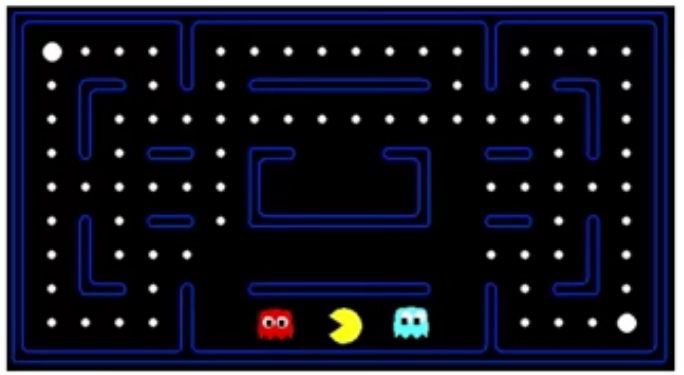




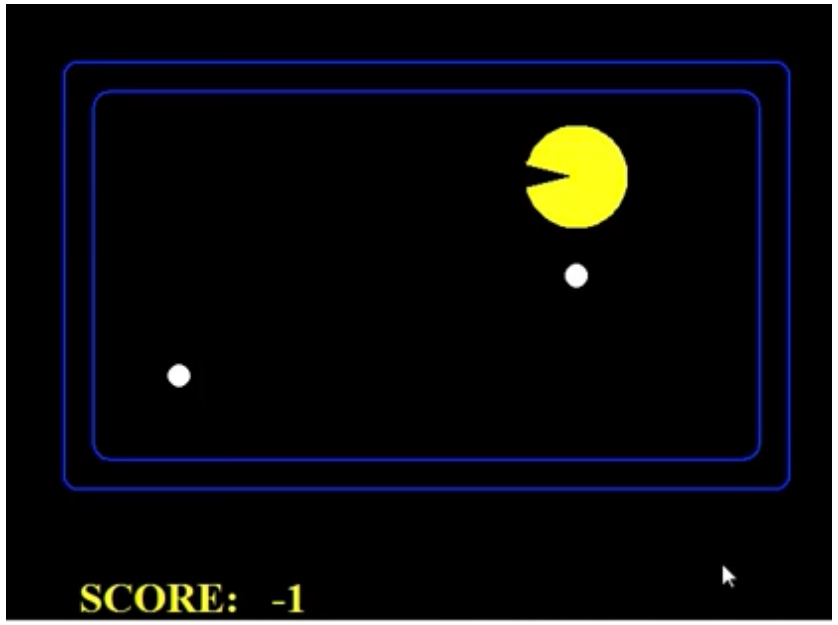
Eval(state)

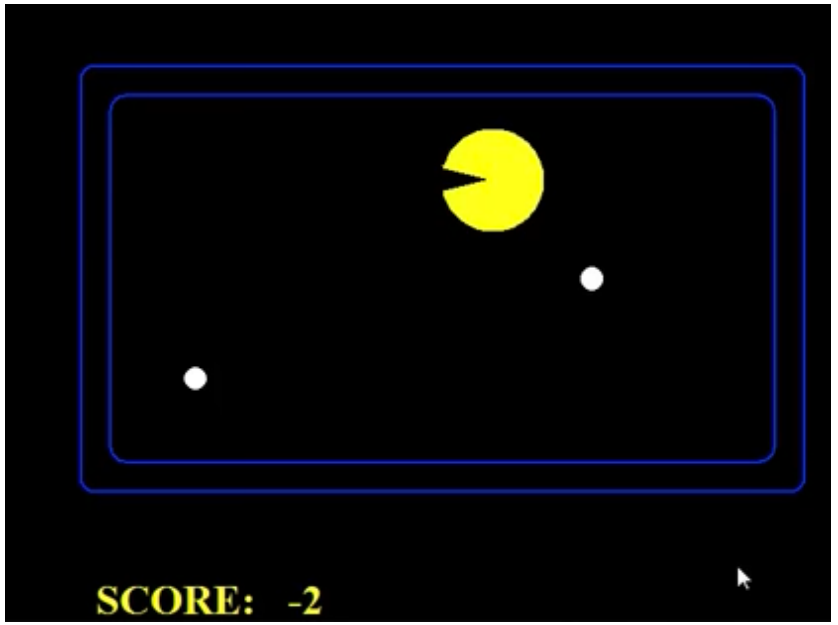


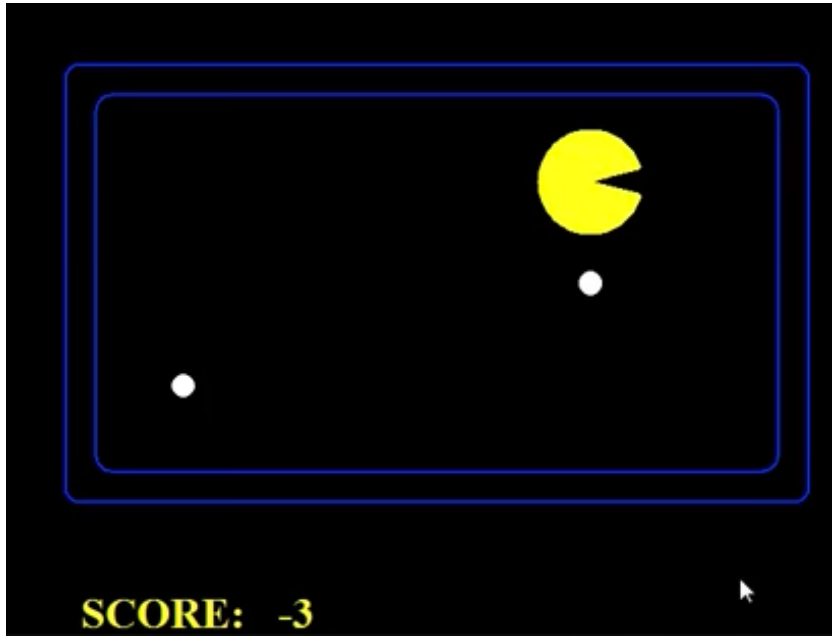


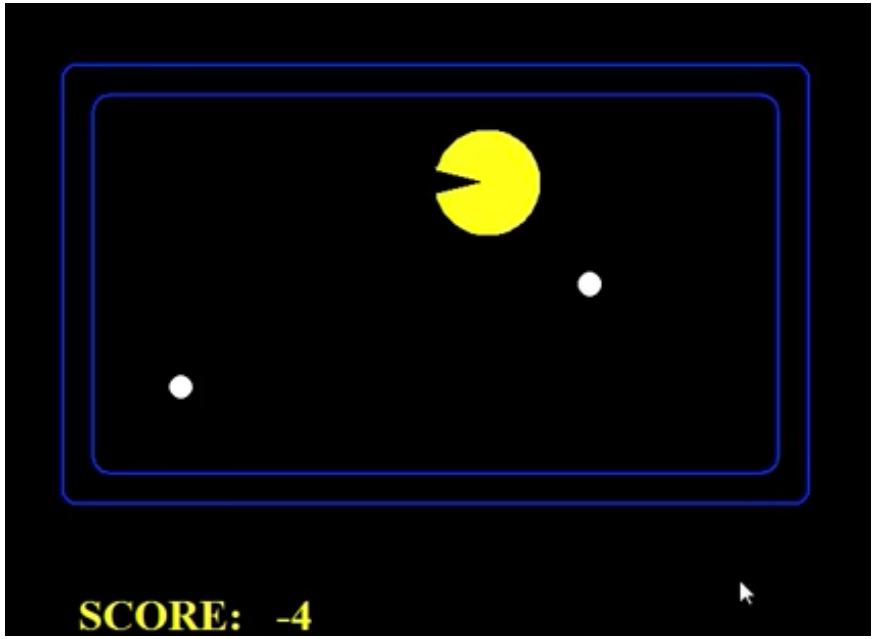


Board snapshots: need scalars from eval function that effectively rank these

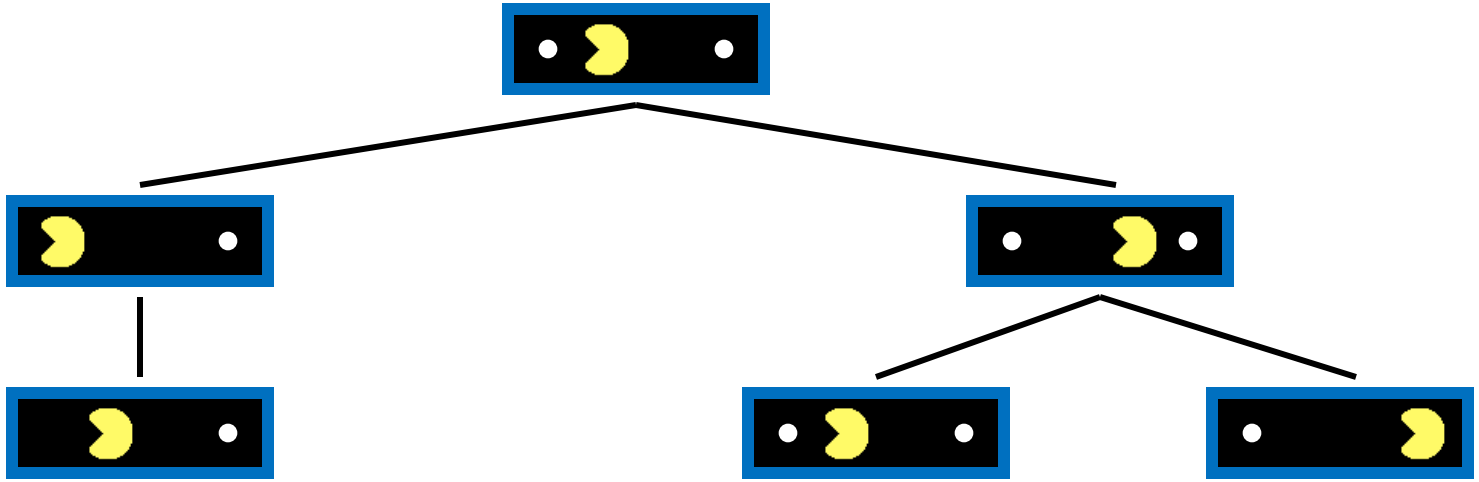

SCORE: -1


SCORE: -2


SCORE: -3


SCORE: -4

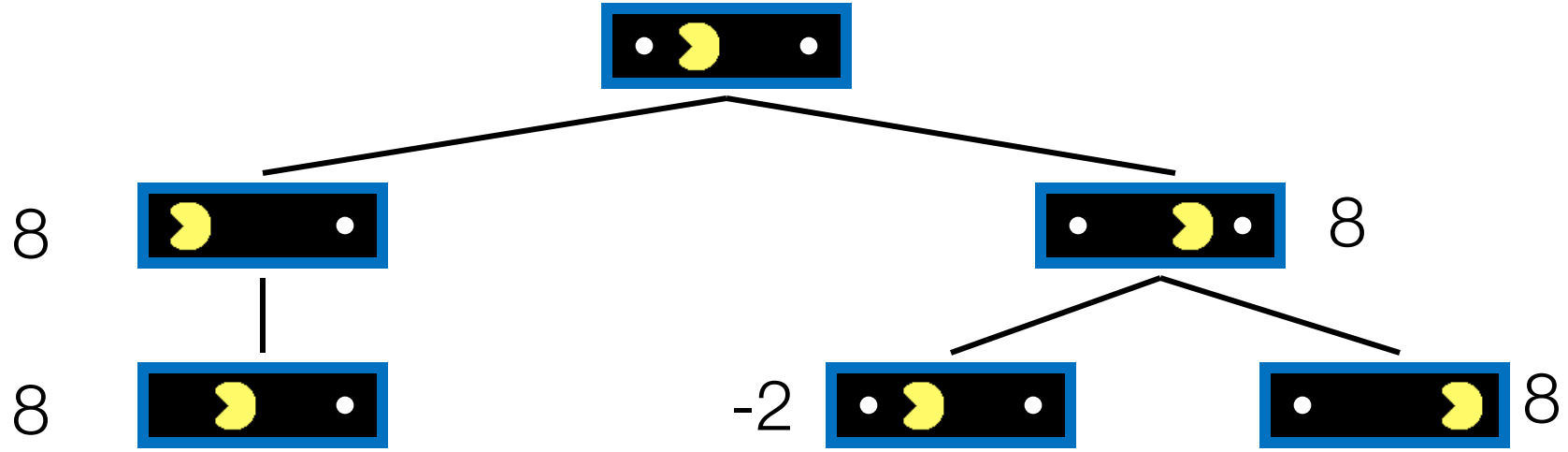
Thrashing (starving PacMan)



A 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!

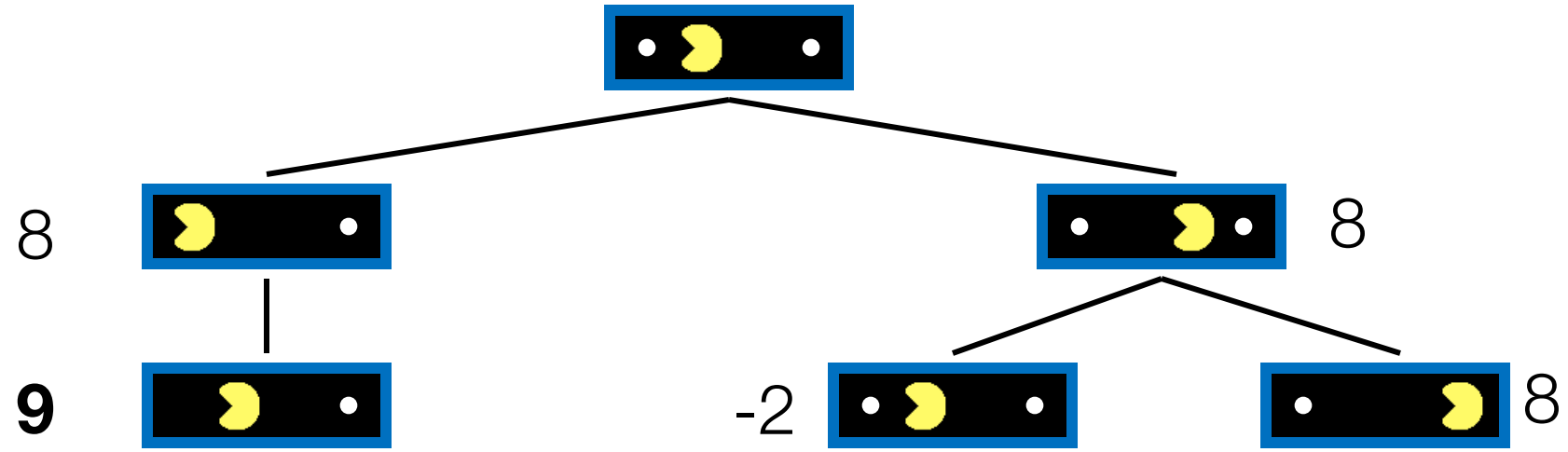
Thrashing (starving PacMan)



A 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!

Thrashing (starving PacMan)



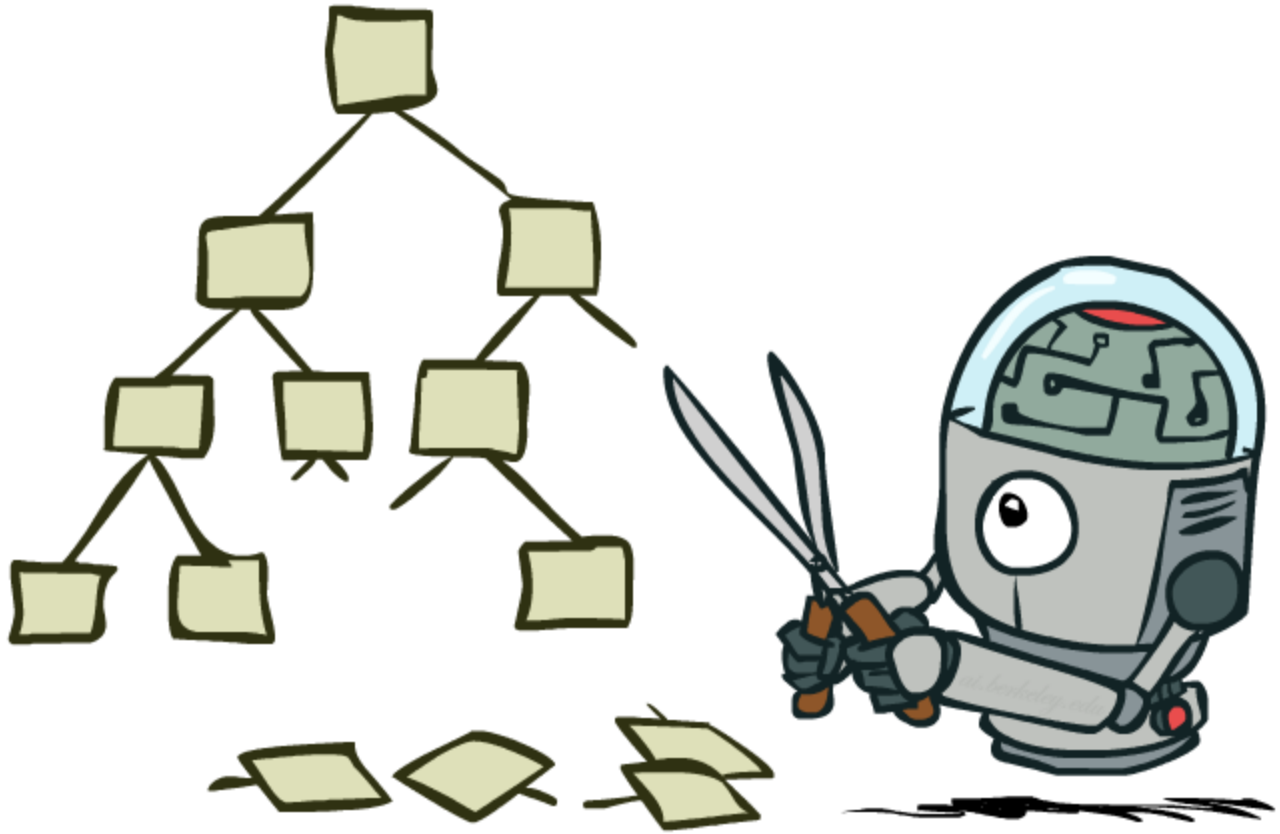
A danger of replanning agents!

- 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!

When you see thrashing…

… You probably need to revisit your evaluation function

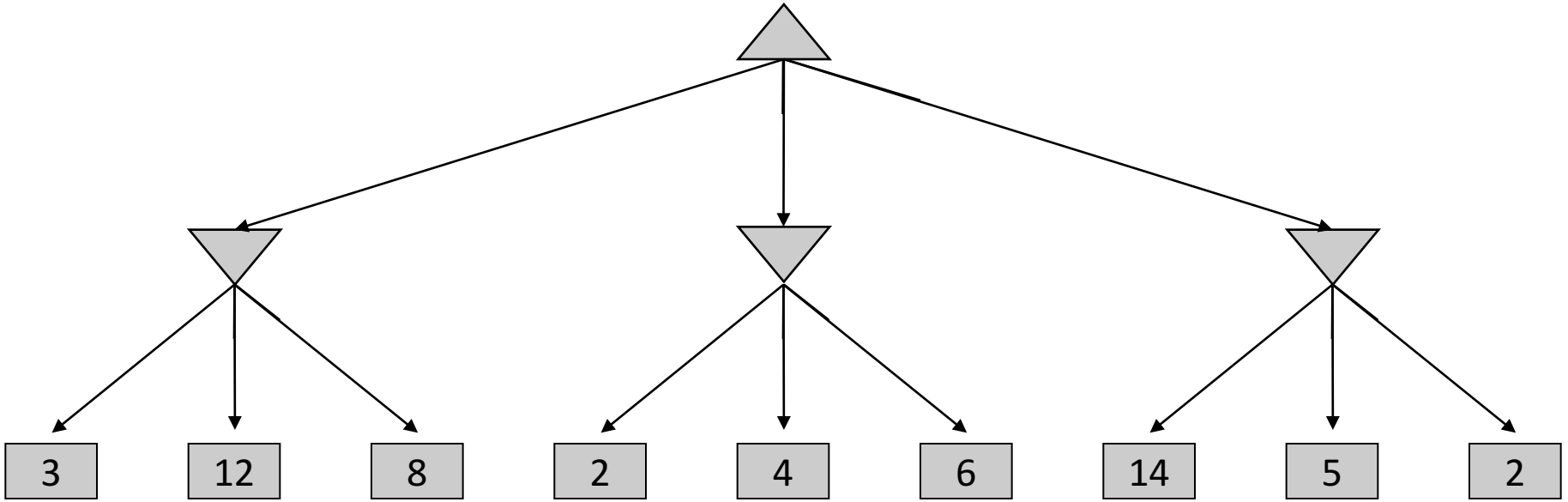
Game tree pruning



Returning to our minimax example

max

min

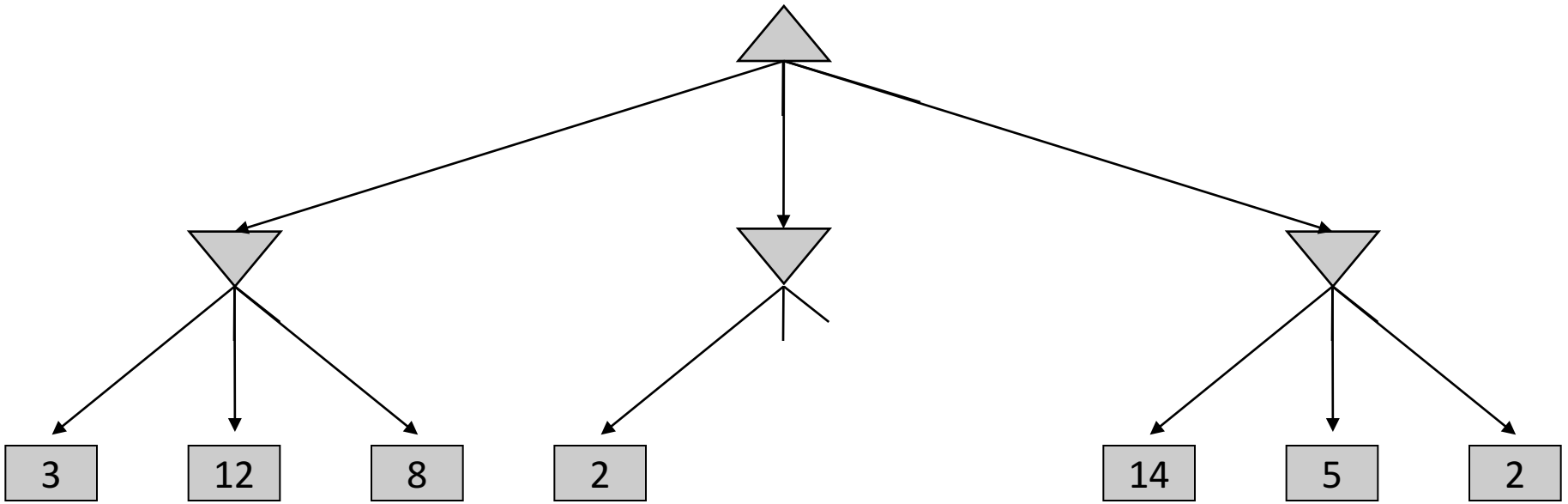


What did we do that was inefficient here?

Minimax pruning

max

min

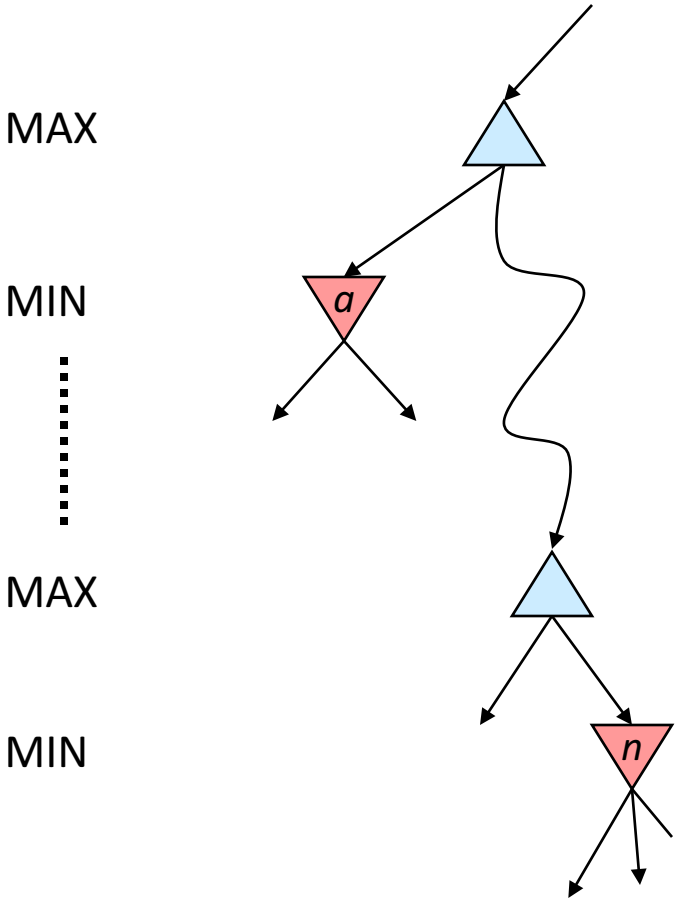


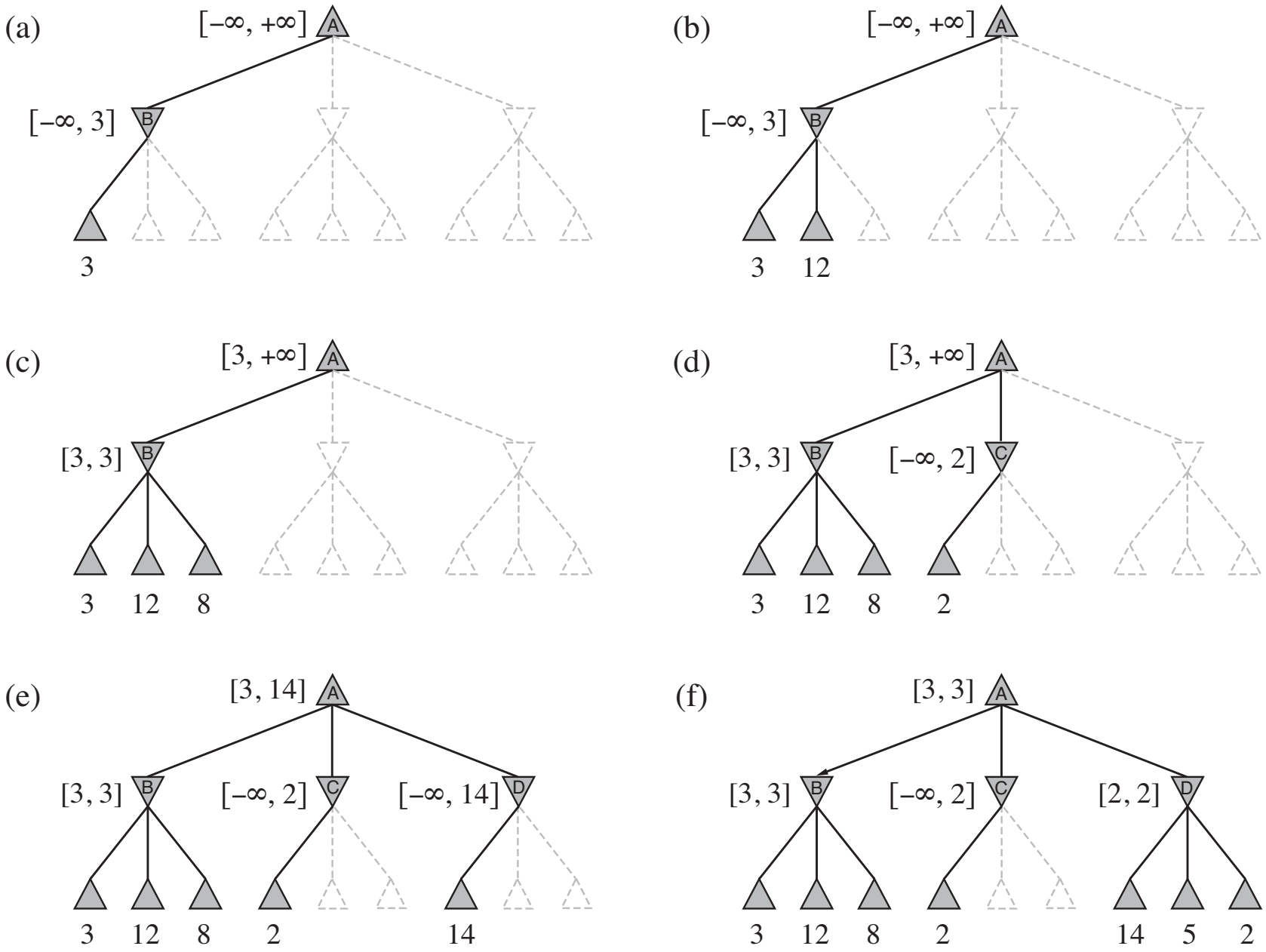
Alpha-Beta pruning

General configuration (MIN version)

- We're computing the MIN-VALUE at some node *n*
- We're looping over *n*'s children
- *n*'s estimate of the childrens' min is dropping
- Who cares about *n*'s value? MAX
- Let *a* be the best value that MAX can get at any choice point along the current path from the root
- If *n* becomes worse than *a (n<=a),* MAX will avoid it, so we can stop considering *n*'s other children (it's already bad enough that it won't be played)

MAX version is symmetric




(a)
[−∞, +∞]
A
[−∞, 3]
B
3
(b)
[−∞, +∞]
A
[−∞, 3]
B
3
12
(c)
[3, +∞]
A
[3, 3]
B
3
12
8
(d)
[3, +∞]
A
[3, 3]
B
[−∞, 2]
C
3
12
8
2
(e)
[3, 14]
A
[3, 3]
B
[−∞, 2]
C
[−∞, 14]
D
3
12
8
2
14
(f)
[3, 3]
A
[3, 3]
B
[−∞, 2]
C
[2, 2]
D
3
12
8
2
14
5
2

Alpha-Beta implementation

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

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

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

Alpha-Beta implementation

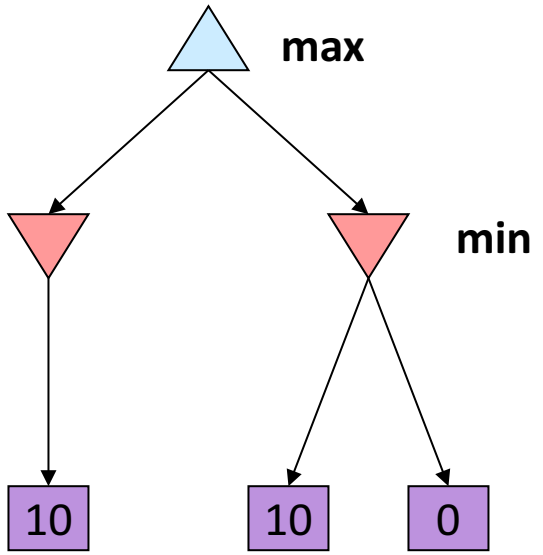
α: MAX's best option on path to root
β: MIN's best option on path to root

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

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

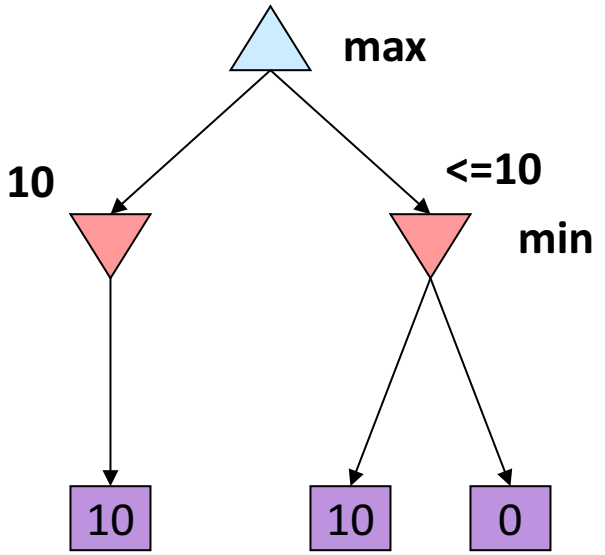
Can we use this for action planning?

What should max do?
What does alpha-beta do?



Can we use this for action planning?

What should max do?
What does alpha-beta do?



From the root it looks like Max could pick either branch, but this is wrong, although the value is correct.

Alpha-Beta pruning properties

This pruning has **no effect** on minimax value computed for the root!

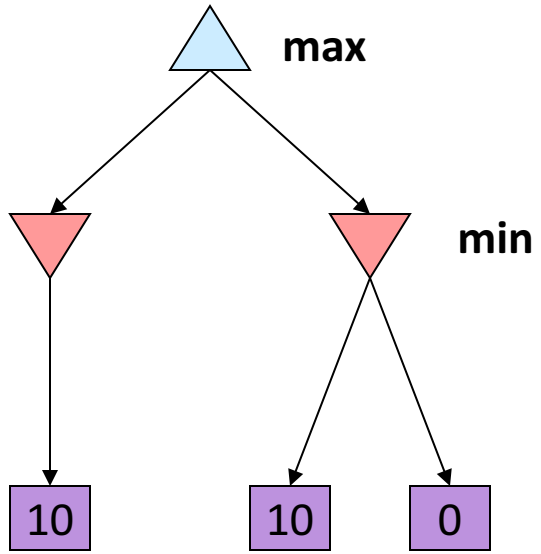
Values of intermediate nodes might be wrong

- Important: children of the root may have the wrong value
- So the most naïve version won't let you do action selection

Good child ordering improves effectiveness of pruning

With "perfect ordering":

- Time complexity drops to $O(b^{m/2})$
- Doubles solvable depth (in best case)!
- Full search of, e.g. chess, is still hopeless…

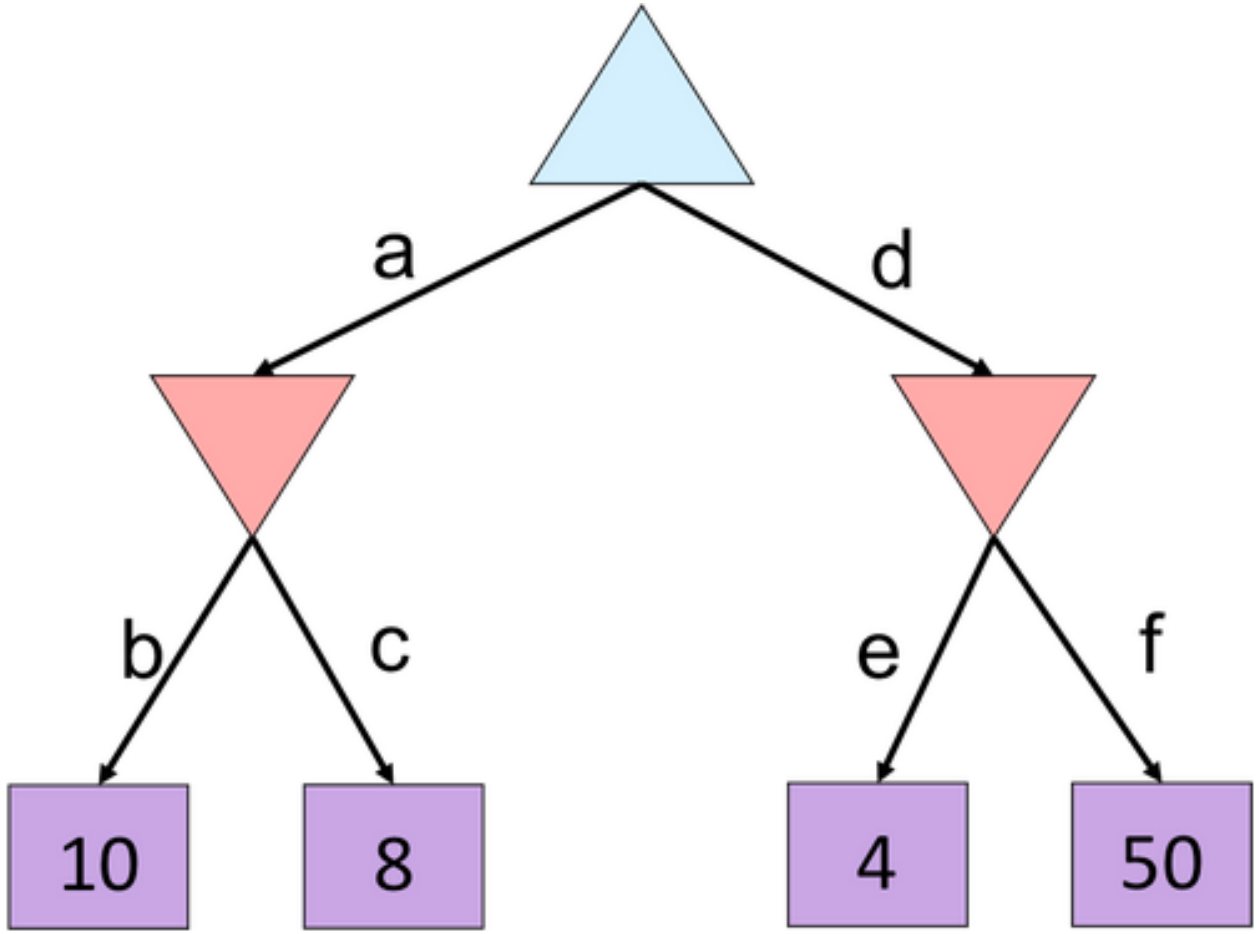


This is a simple example of **metareasoning** (computing about what to compute)

That’s all for today

- Next time: uncertain outcomes – playing with expectations and utilities

Alpha-Beta example



Example 2