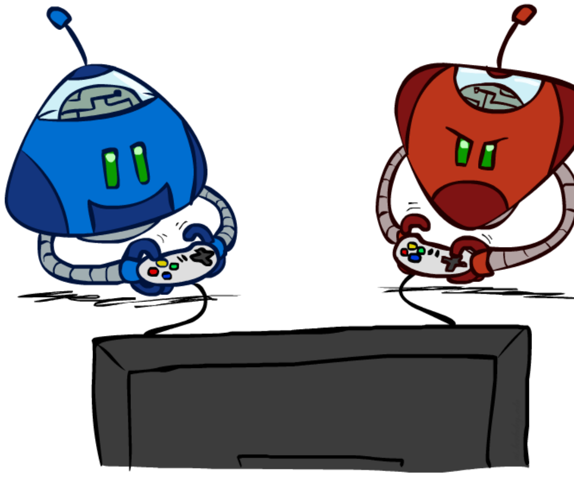
Artificial Intelligence

Chapter 6: Adversarial Search



Updates and Additions: Dr. Siamak Sarmady
By: Dan Klein and Pieter Abbeel
University of California, Berkeley

Game Playing

- If someone or a computer can play a game with you and can win, it shows smartness and intelligence.
- So researchers have used games to make computers smart and show that they are intelligent…

Checkers:

1950: First computer player.

1994: First computer champion, Chinook, ended 40-year dominance of human champion Marion Tinsley using complete 8-piece endgame.

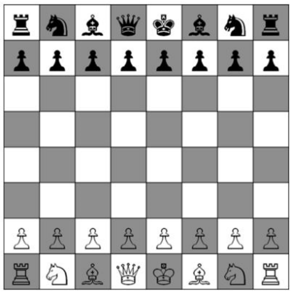
It means, it can understand all possible scenarios until next 8 moves, if there is a guaranteed possibility of win, enforce a win.

2007: Checkers solved!

It means your program can understand all possible scenarios from the beginning of the game (e.g. if you are the beginner), whether it is possible to enforce a win or a tie or you will lose …

In checkers, a tie will happen if both play correct (tic-tac-toe is same). If the opponent make a mistake, you can enforce a win

Game Playing



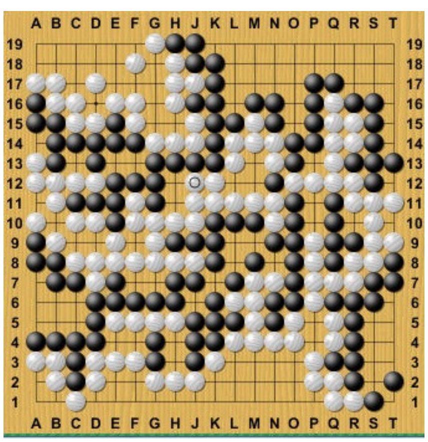
Chess:

1997: Deep Blue defeats human champion Gary Kasparov in a six-game match.

Deep Blue examined 200M positions per second, used methods that searched up to 40 steps ahead. Current programs are even better.

We still cannot check the whole game (whether there is a guaranteed way to enforce a win or tie)

Game Playing



Go:

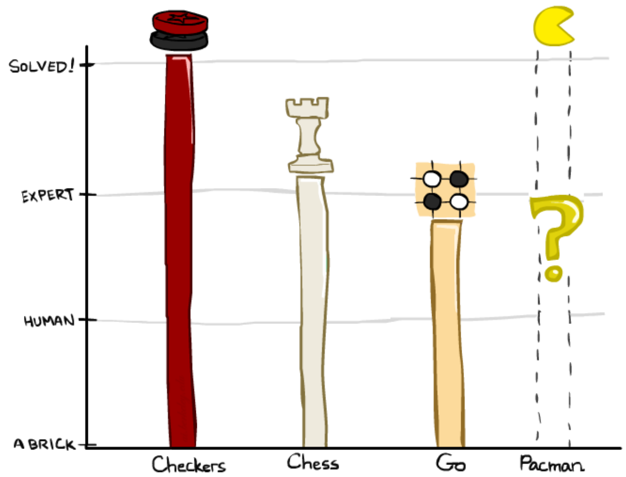
Human champions are now starting to be challenged by machines, though the best humans still beat the best machines.

In go, branching factor, b > 300! (i.e. moves in each step). So Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.

That is in this game computer software do not look everywhere, they just search some of the search space…

These programs are not in champion level, but in expert level.

Game Playing State-of-the-Art



- **Pacman?**

  We don't know…

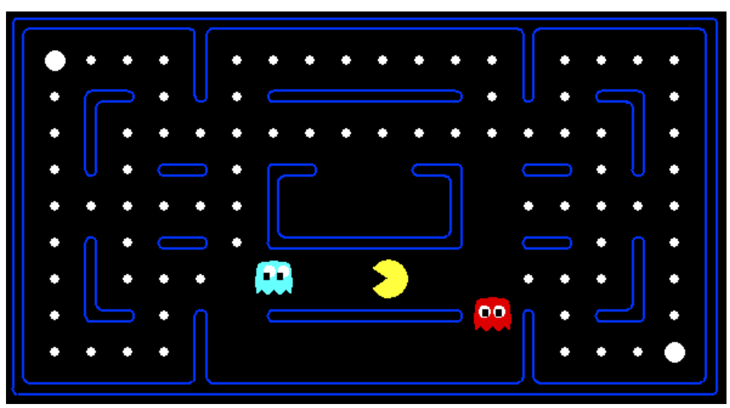
Behavior from Computation

Try to guess what kind of computations happen in the software to produce these behaviors you see?

It is NOT bunch of IF…Then commands!

- Each player (Eater and Ghosts) are searching for options at each step

- One player plays at each step and then the others decide what to do to increase the chance of win…



[Demo: mystery pacman (L6D1)]

Video of Demo Mystery Pacman

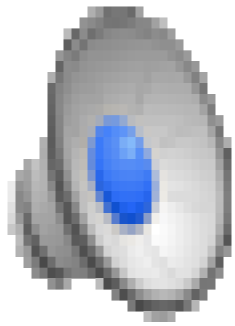
Try to guess what kind of computations happen in the software to produce these behaviors you see?

It is NOT bunch of IF…Then commands!

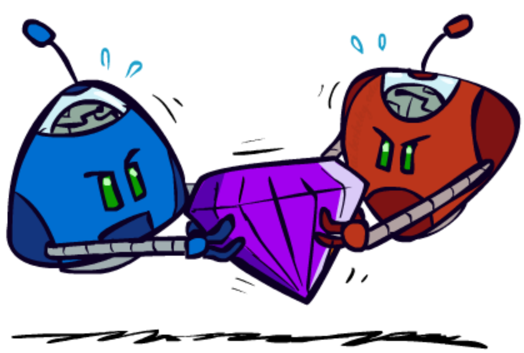
- Each player (Eater and Ghosts) are searching for options at each step

- One player plays at each step and then the others decide what to do to increase the chance of win…

Wasn't perfect?



Adversarial Games



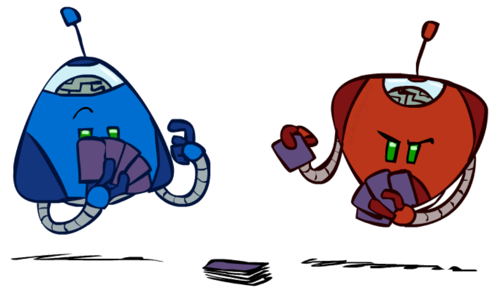
Two or more agents play, each trying to maximize its own utility…

Sometimes, one player's win, is the other's loss

Types of Games

- Many different kinds of games!

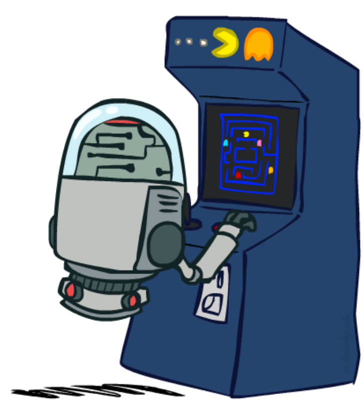
- Possibilities:
  - Deterministic vs. stochastic
  - One, two, or more players
  - Zero sum vs. cooperative
  - Perfect information (Chess), Partial Info (Cards)



- We want algorithms for calculating a strategy (policy) which suggests a move in each state…
  - Strategy means series of actions
  - Since it is not single player, we don't know what is going to happen. In each situation we want to know what the agent should do…

Deterministic Games

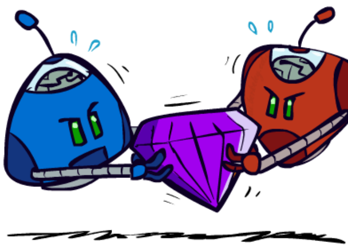
- Formalization:
  - **States:** S is the set of states in the game (start at s_0)
  - **Players:** P={1...N} (usually take turns)
  - **Actions:** set of A (may depend on player / state)
  - **Transition Function:** $SxA \rightarrow S$
    - Maps an s_n state to a resulting state S_{n+1} because of an action
    - We used to call it a successor function in single player…
  - **Terminal Test:** $S \rightarrow \{t,f\}$
    - A test that in last state, will tell us whether the agent won
  - **Terminal Utilities:** $SxP \rightarrow R$
    - Determines the utility for each player in the last state
    - In games we normally calculate the utility at the end of the game

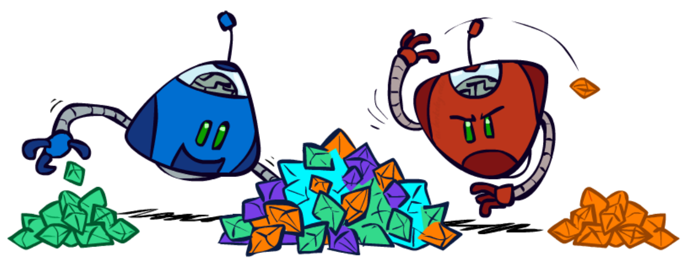


- Solution for a player is a policy: $S \rightarrow A$

Zero-Sum Games

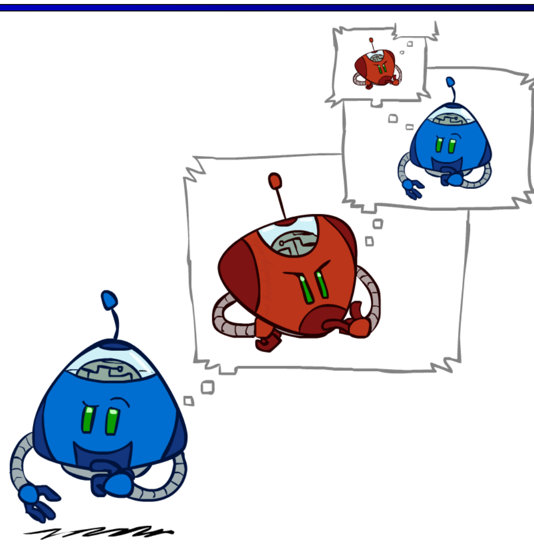
Each one seeks higher utility for itself, but they simplify the job for the other as they sort, not a full collaboration though





- 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

Adversarial Search



Decisions are taken based on what each agent thinks the other agent will do…

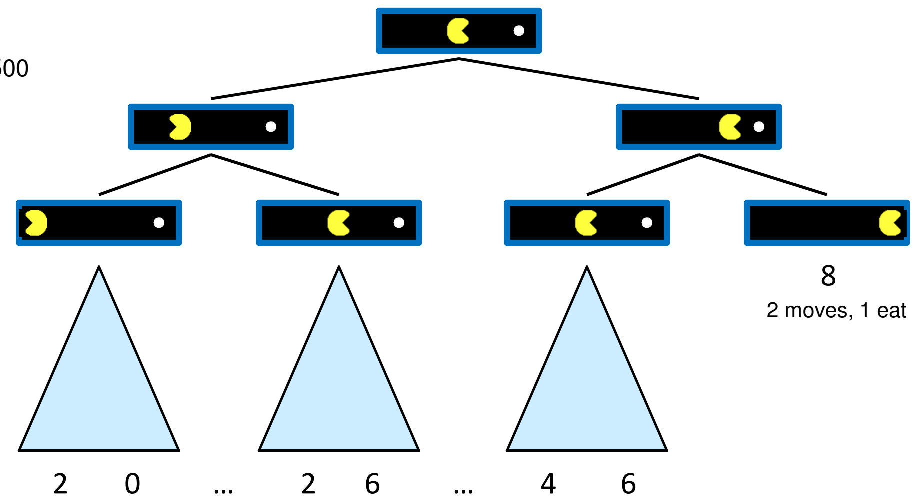
So, assuming we perform the first move, we think what actions the other agent might perform, and in answer to each of those actions what we should do …

In some games it is possible to have a search tree

Single-Agent Trees

One move: -1
Eat food pallet: +10
Sometimes game win: +500

To which side do you think the agent will go, if it has these utilities?



Value(potential) of a State (Single Player)

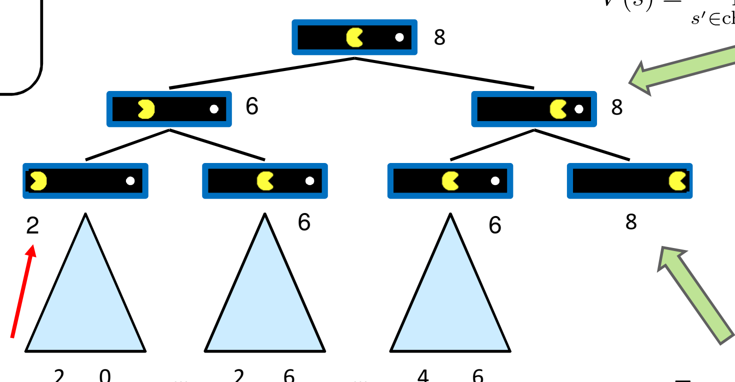
Value of a state:
The best achievable outcome (utility) from that state

We do maximum propagation, because the agent has the full decision and it will naturally want to select the node with higher utility…

So we build "maximum utility direction" information for it

Non-Terminal States:

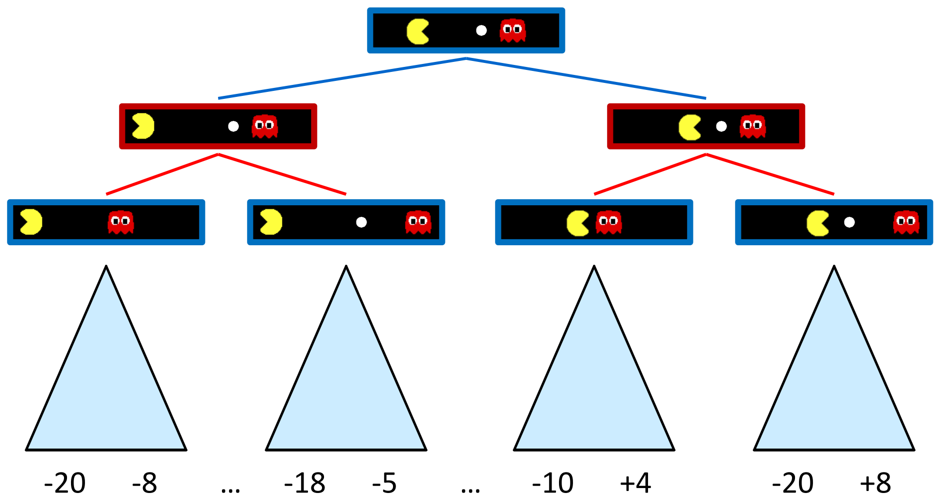
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Terminal States:
$V_{(s)}$ = known

Propagate the best outcomes towards up…

Adversarial Game Trees



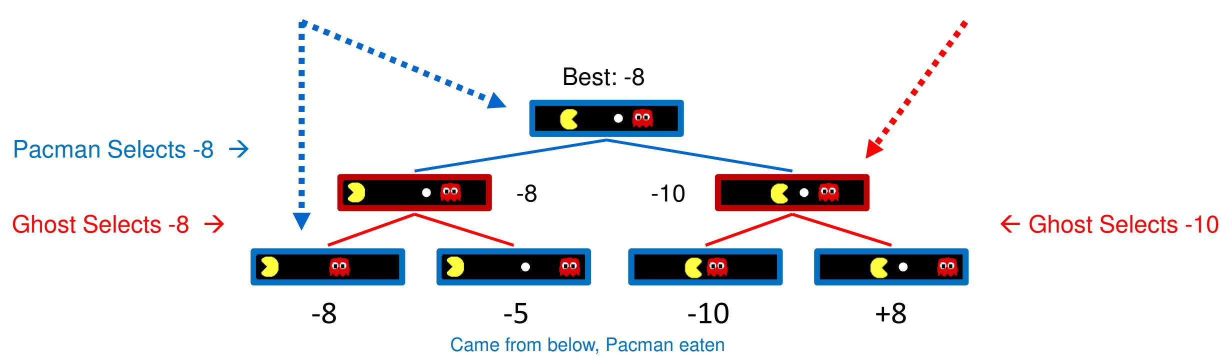
Here, if both players play carefully, it is guaranteed loss for Pacman

(track moves to see)

Minimax Values

Selection of Next States Under Agent's Control: $V(s) = \max_{s' \in \text{successors}(s)} V(s')$

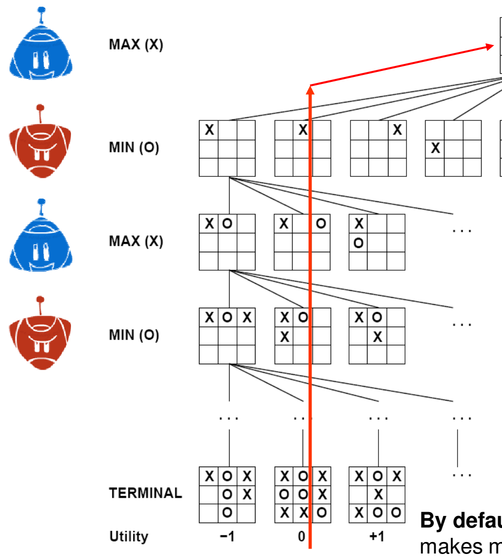
Selection of Next States Under Opponent's Control: $V(s') = \min_{s \in \text{successors}(s')} V(s)$



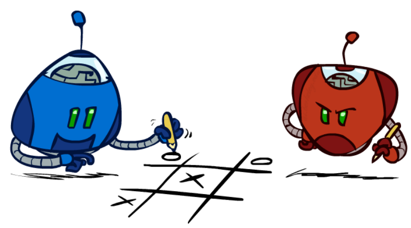
Minimizing Node: Ghost selects the **next move**
Maximizing Node: Pacman selects the **next move**
Propagate up, for maximizing nodes select max, for minimizing nodes take min

Terminal States:
$V(s) = \text{known}$

Tic-Tac-Toe Game Tree



The score here always traces back to a terminal state with a utility of 0

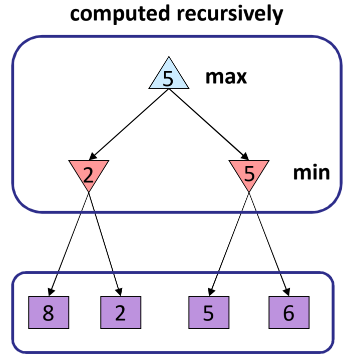


By default, If both players play carefully in this game, result will be a tie (unless one makes mistake). That is you will end up in a 0 terminal utility (many might exist)… If one makes mistake, we may go to some sub-trees that allow to force a win or loss.

Adversarial Search (with Minimax) Summary

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers, Pacman
  - **One of the players** maximizes result
    selects the node with Max utility
  - **The other** minimizes result
    selects the node with Min utility of opponent, i.e. max of his own
- Minimax search algorithm:
  - A state-space search tree
  - Players alternate turns
  - **Compute every node's minimax value:**
    1. Visit all branches and terminal states (DFS). Propagate the values upward and calculate the Minimax values.
    2. Try to go a path with the best achievable utility **against** a rational (optimal) adversary

Minimax values: computed recursively



Terminal values: part of the game

Max is not 8, … You cannot force 8 in left branch to your opponent. Max that you can force is 5. **You can still hope for 8** (if opponent makes mistake)

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
```

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

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

Initialize with worse possible score…

And then use recursive calls to min-value and max-value in turns

Just works if we have two agents which play in turns

Minimax Implementation (Dispatch Function)

Better implementation, decides based on the node type i.e. maximizer or minimizer…

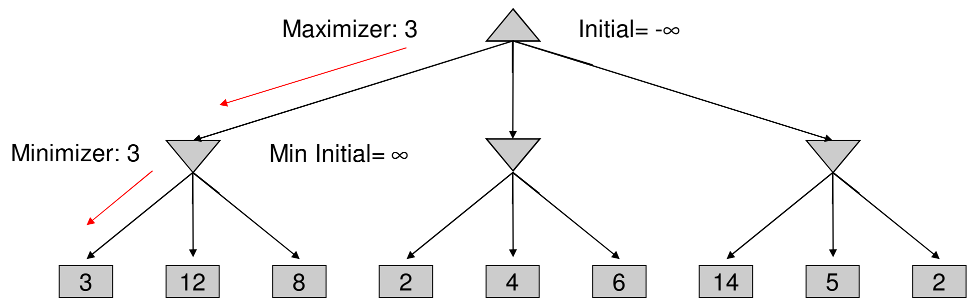
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

Because we don't always have a maximizer and a minimizer, e.g. if we have 3 players… i.e. pacman and 2 ghosts

```
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

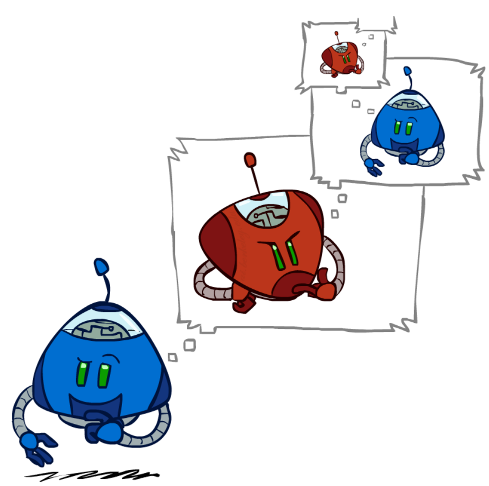


If both play optimally, this is the outcome of the game

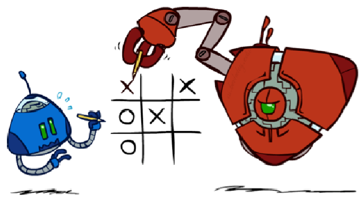
In exam you will propagate from bottom to top, but in programs you don't have/create the whole tree… you use DFS etc.

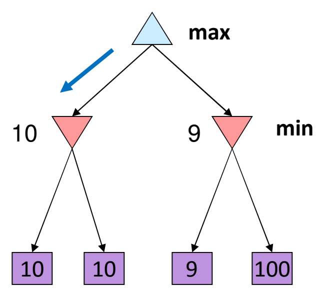
Minimax Efficiency

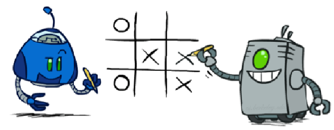
- How efficient is minimax?
  - Just like (exhaustive) DFS
  - **Time:** $O(b^m)$, b: branching m: depth
  - **Space:** O(bm)
- Example: For chess e.g. $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree? Can we get away with partial search?



Minimax Properties







Optimal against a perfect player. Otherwise?

If we suspect that the minimizer may make mistake, perhaps we go to right? (9 and 10 are not that far, and we have chance of wining 100!)

[Demo: min vs exp (L6D2, L6D3)]

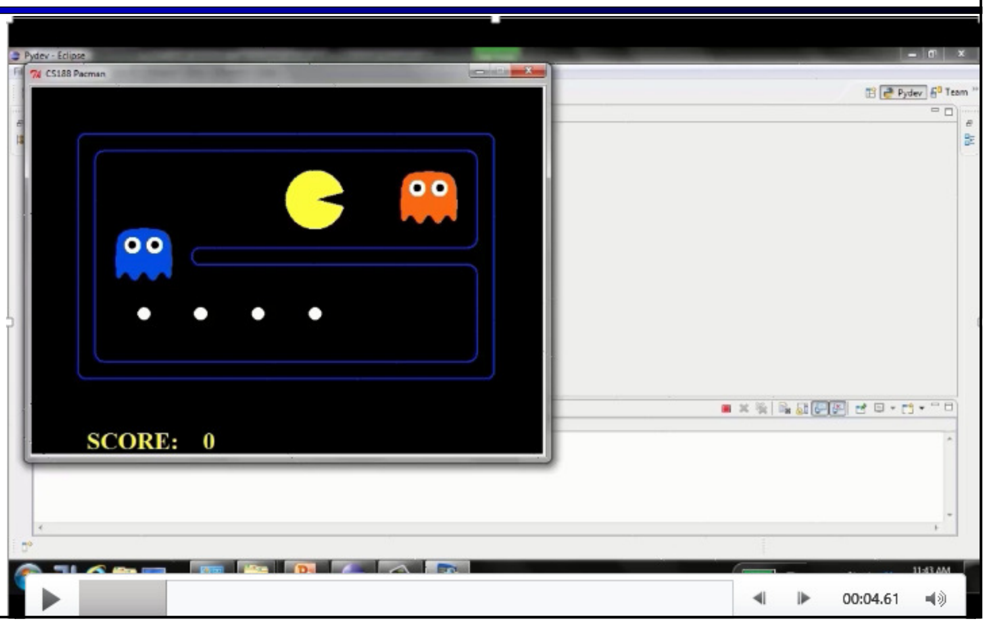
Video of Demo vs. Expert (Minimizer)

If both ghosts are experts, then we already know that there is no chance….

In that case what should packman do to maximize its utility?

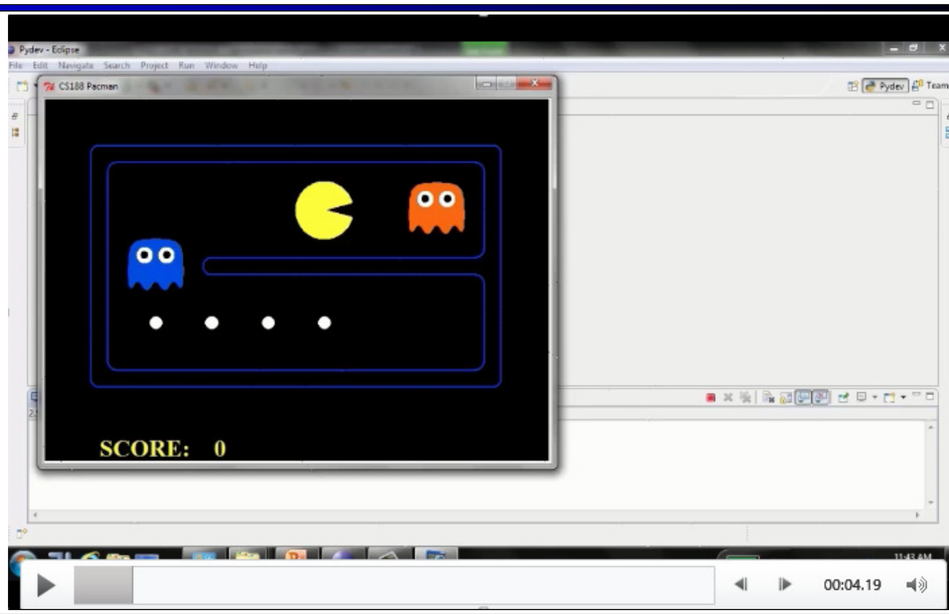
If living means getting more and more negative score, better die and stop the negative scores..!

Since we lose 1 by each move, better suicide and lose just 1 than wasting more scores…



Video of Demo vs. Non-Expert Ghosts

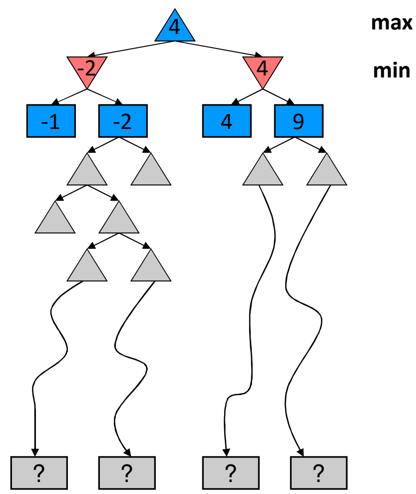
If the ghosts play randomly, then we have a 50% chance that the blue moves down (or up)… so we take our chance…



Resource Limits

Resource Limits

- **Problem:** In realistic games, cannot search to leaves!
- **Solution:** Make up values!
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions…
  - We wish that those non-terminal value represents the rest of the branch (exactly like heuristic). But we cannot be sure of that…
  - Quality of the evaluation function is important
  - Now we just have estimates… not real MinMax values.
- **Example:**
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - With α-β pruning, can reach about depth 8 – decent chess program
- Guarantee of optimal play is gone
- Checking more levels makes a BIG difference in accuracy
  - Goes near to actual end game scores (otherwise we would only evaluate 2 highest nodes)
- Use iterative deepening for real time algorithm (until time runs up)



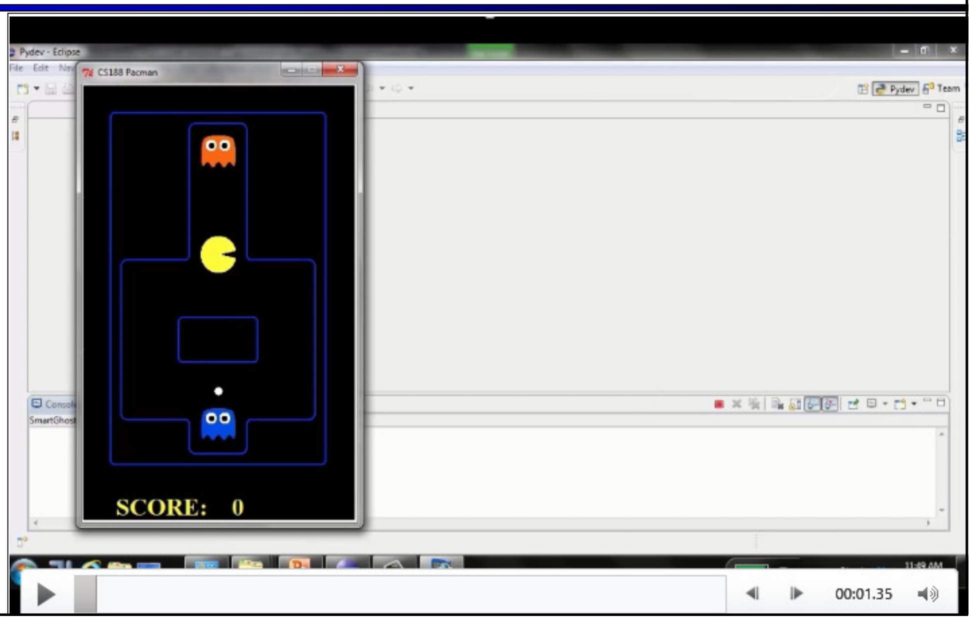
Depth Matters

- Evaluation functions are always imperfect
- The deeper in the tree you go, your results will be more reliable and the quality of the evaluation function matters less

[Demo: depth limited (L6D4, L6D5)]

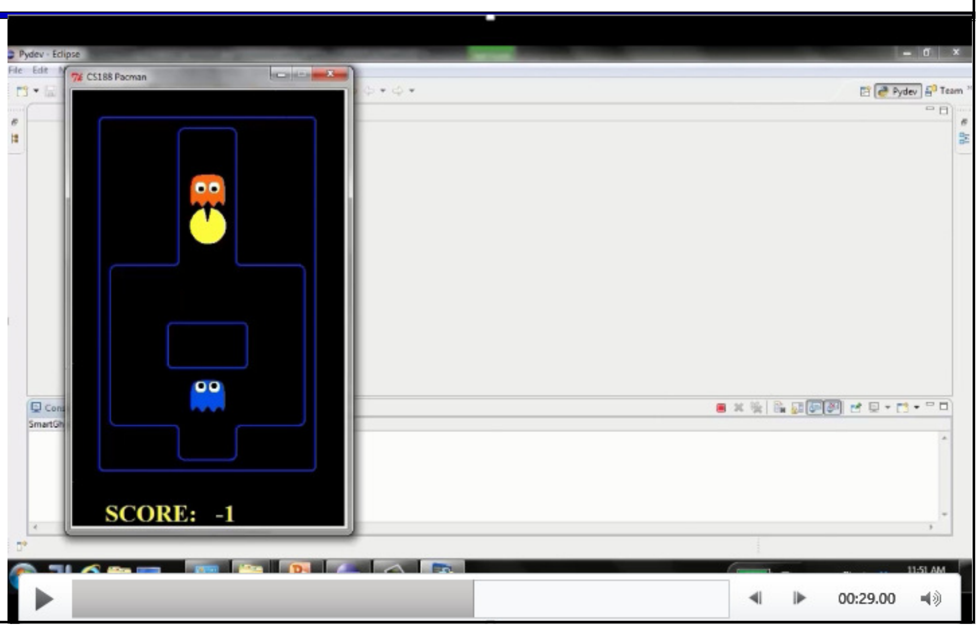
Video of Demo Limited Depth (2)

- In this demo we only search 2 levels deep
- Players cannot stop, they should move
- Pacman can either go up or down, if down it does not know which side will blue ghost go
- Pacman decides that it should not go up, but does not know which side to go after it comes down …
- so blue closes the road to it, while orange closes its back

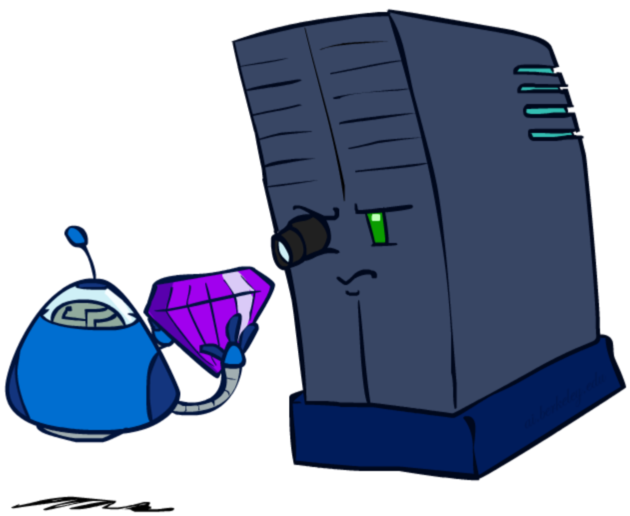


Video of Demo Limited Depth (10)

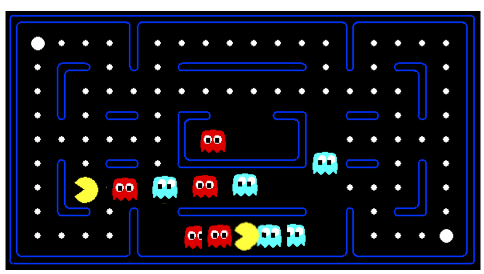
- This time we look ahead 10 moves
- This time pacman finds that the blue ghost should decide to go left or right after coming up
- It goes up one step to give time for itself to see what blue is going to do (it knows it can still be ahead of orange ghost)…
- Notice that the evaluation function is the same, only we search more steps ahead



Evaluation Functions



Evaluation for Pacman



- The evaluation function could be just a running score of the game … so in Pacman it could be something like f(x) = Eaten Palets * 10 – Movements
- But what happens if our evaluation function is not good or sufficient
- It should be able to differentiate between states and gives better value for better states

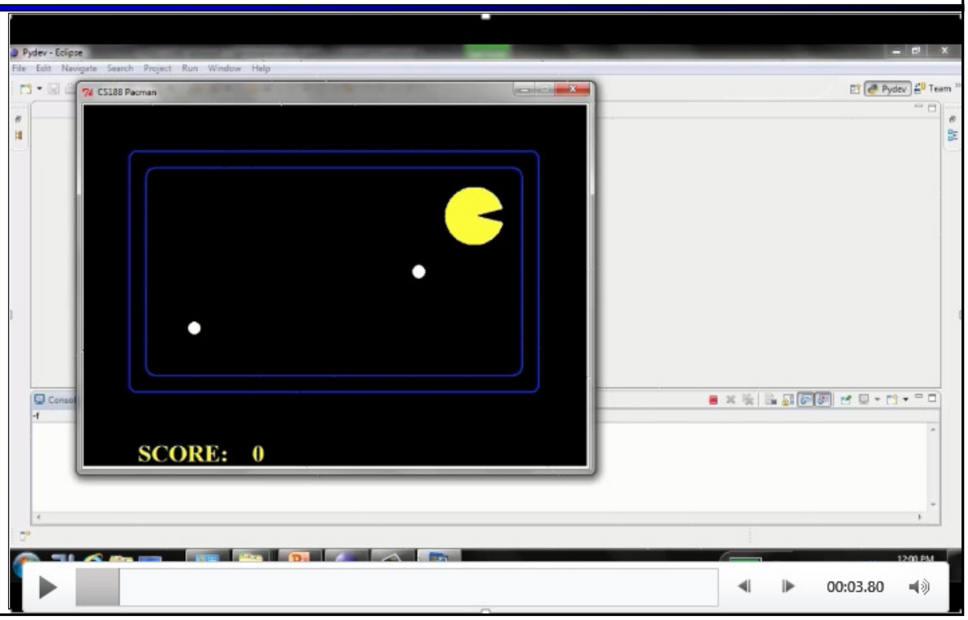
[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function), smart ghosts coordinate (L6D6,7,8,10)]

Video of Demo Thrashing (d=2)

- Evaluation function is just score of game

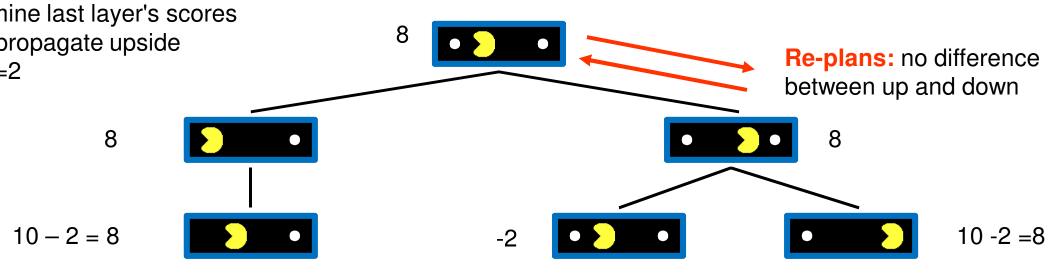
- What happens if we just do two levels of evaluation only…

- there are no ghosts and you think it should be easy



Why Pacman Starves

- determine last layer's scores
- Then propagate upside
- depth=2

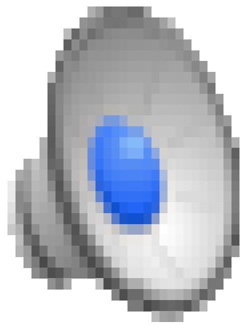


- A danger of re-planning agents!
  1. It knows his score will go up by eating the dot now (west, east)
  2. It knows his score will go up just as much by eating the dot later (east, west)
  3. There are no point-scoring opportunities after eating the dot (within the horizon, two here)
  4. Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of re-planning!

Video of Demo Thrashing -- Fixed (d=2)

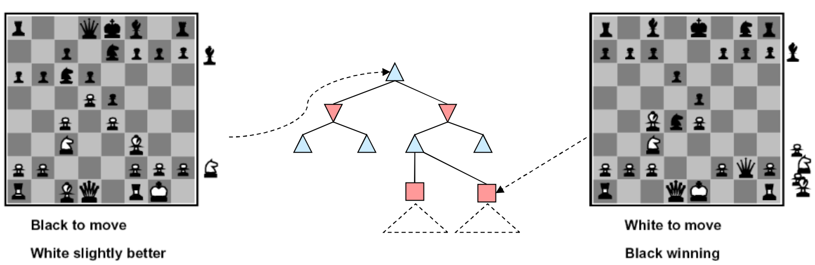
So how we fix it?

- By using a better heuristic, for example adding the distance to food pallets to the function
- In that case the graph we draw will no more produce a tie…
- If you see trashing in your problems, normally it is something in your evaluation function



Evaluation Functions

- Evaluation functions determine scores for non-terminal states in depth-limited search

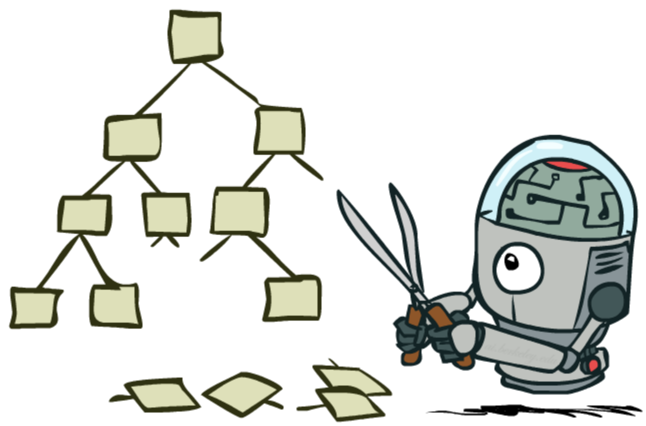


- **Ideal function:** returns the actual minimax value of the position
- **In practice:** approximate it. For example, sum up the scores each wins (might represent how good a branch is)
- **Chess:** typically weighted linear sum of features (produces a real value)

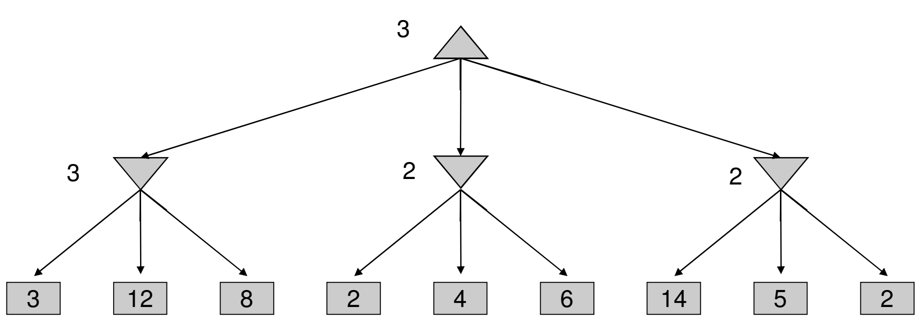
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- Such as: $f_1(s)$ = (num white queens – num black queens) , and f_2 for bishops etc.

Game Tree Pruning

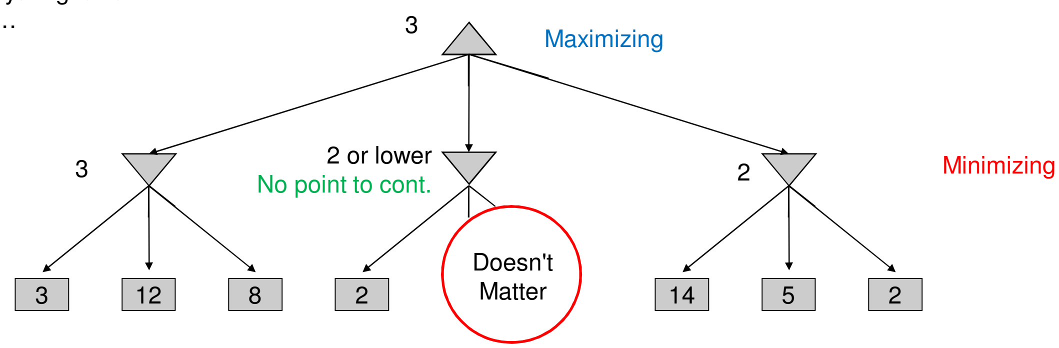


Minimax Example



Minimax Pruning

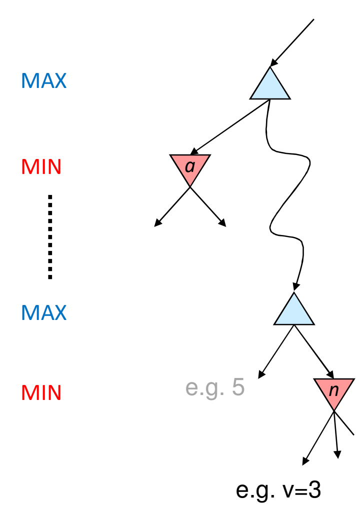
Maximizer will not choose anything lower than 3 so…



We might have pruned 2/9 of the tree

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 *α* be the best value that MAX can get at any choice point along the current path from the root
  - If v ≤ *α* and *n* becomes ≤ *α*, MAX at the top will avoid reaching it, so we can stop considering *n*'s other children (it's already bad enough that it won't be played), so as soon as seeing v=3, (**prune if v ≤ α**)
- **MAX** version is symmetric
  - The same way, if we are finding MAX-VALUE, the value cannot be ≥ b because the MIN in higher levels will avoid reaching the lower part of the tree (**prune if v ≥ β**)



We keep best MAX (α) and MIN (β) value we have seen along the path up until the route

Alpha-Beta Implementation

α Is the biggest value seen
β is the smallest value seen

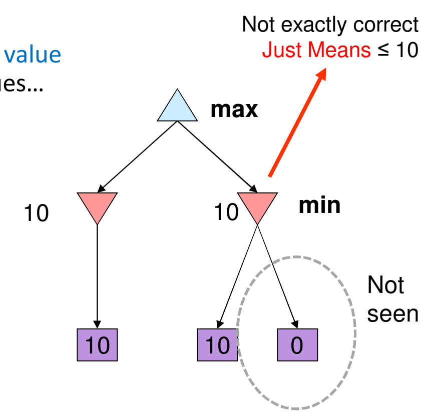
α: MAX's best option on path to root
β: MIN's best option on path to root

```
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    //no need to cont.
        β = min(β, v)
    return v
```

Alpha-Beta Pruning Properties

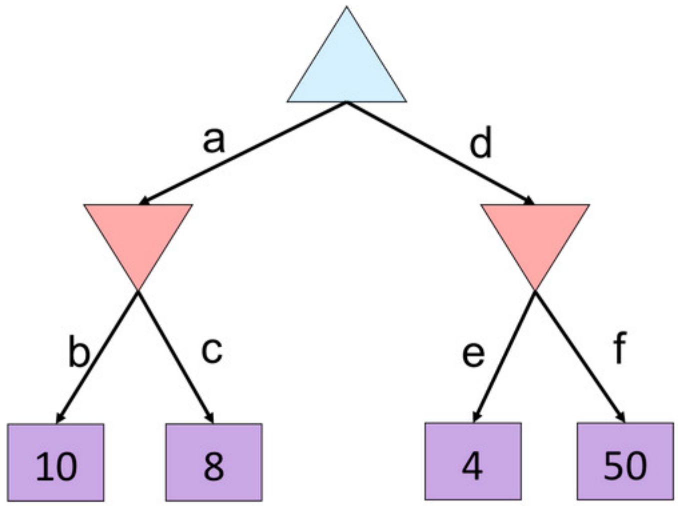
- This pruning has no effect on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - **Important:** children of the root (intermediate) may have the wrong value because we perform pruning and do not always calculate exact values…
  - So the most naïve version won't let you do action selection
    For example, will not help you in breaking ties for 10-10
  - Just passing up the alpha, beta will not be enough…



- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$ … search better Childs first?
  - Halves the depth so it can double the solvable depth!
  - Full search of, e.g. chess, is still hopeless…
- This is a simple example of meta-reasoning (computing about what to compute)

Alpha-Beta Quiz

Where does pruning happen?



Alpha-Beta Quiz 2

Where does pruning happen?