Game play and adversarial search

Some figures are derived from the slides created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley, available at http://ai.berkeley.edu.

Comparison to planning and reflex agents

- We are still in the agent view of AI - we have a goal!
- But the transition function does not depend only on our actions
  - There are other agents who take actions as well
  - Usually, in opposition to our goals
- Planning all actions ahead of time will not work, we need to react to the actions of the other agents.
- Paradoxically: a reflex agent, with a lookup table for every state might work (but would be very inefficient)

State of the art in game play

- **Checkers**: 1994: First computer champion. 2007: Checkers **solved**!
- **Chess**: 1997 Deep Blue defeated human champion Gary Kasparov. Very sophisticated evaluation techniques, and significant computing power. These days: trivial computing power can defeat any human.
- **Go**: 2016, DeepMind AlphaGo defeats Lee Sedol, top Go player.
- **Poker**: Some variants were solved (eg. heads-up limit Texas hold'em).

What does it mean for a game to be **solved**: informally, that we found a strategy (*not* a plan) that is the best possible.

Types of games

- **Deterministic** or **stochastic**?
  - Is there randomness involved? Shuffled cards, dice?
- **Complete** or **partial information** game?
  - Is a part of the information hidden?
- One, two or more **players**?
- **Zero** sum? If yes, the game is fully adversarial
- General games
  - Outcome values might be more complex, they don't add up to zero
  - Eg. Monopoly, Settlers of Catan
  - The player's strategy might include cooperation, indifference, competition, alliances, cliques, contracts etc.

Deterministic games

- States $S = \{s_0, \ldots\}$
- Players $P = \{1 \ldots N\}$, take turns
- Actions A. Not all actions might be available for every player at every state.
- Transition function $T(s, a) \rightarrow s'$
  - The fact that this is not probabilistic, makes this a deterministic game
- Terminal test: $completed(s) \rightarrow \{true, false\}$
  - Eg: checkmate!
  - Eg: golden snitch was catched!
- (Terminal) **utilities**: $U(s, p) \in \mathbb{R}$

Game playing in AI

- Agent view of AI: the AI is one of the players.
- Let us assume players A and B who take actions successively.

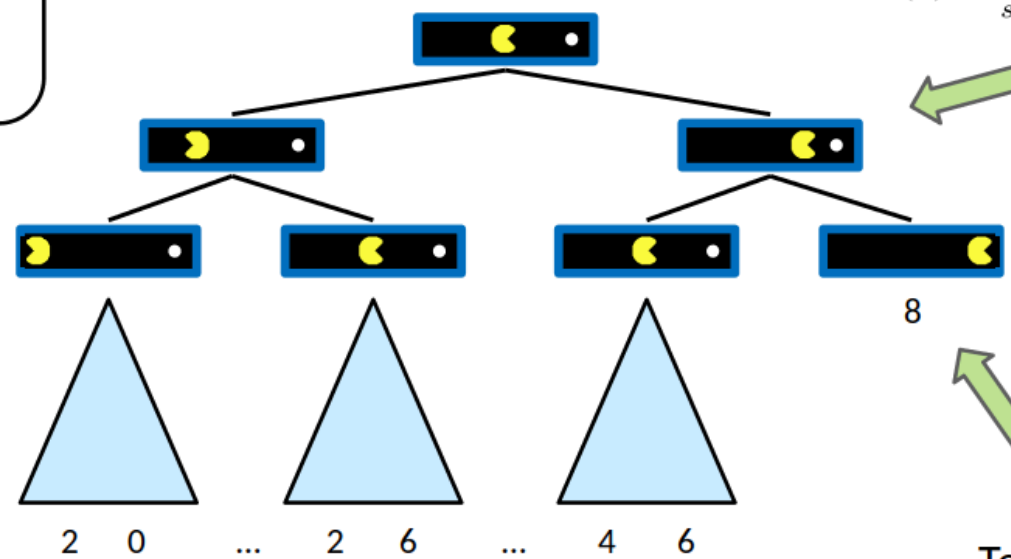
$$s_0 \rightarrow a_{A1} \rightarrow s_1 \rightarrow a_{B1} \rightarrow s_2 \rightarrow a_{A2} \rightarrow s_3$$

- Usually, we cannot search for a **plan**, because the agent's actions are interleaved with the actions of the opponent!
- We will search for a **policy** instead: $\pi(s) \rightarrow a$

Single player, deterministic, complete information game

- Take actions to **maximize** the utility of the terminal state you reach!
- What is value of the *intermediate* states?
  - Depends on where you go from there…
  - But you should go in the direction where you will eventually get better value
  - A perfect player at any choice would choose the one with the maximum value

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:

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

The V value

- The V value of a state s, in many AI contexts, is the value you can achieve starting from s and **acting perfectly from now on**
- In the case of a one player game: just calculate it recursively by max.
  - ...it gets harder later...
- For a terminal state: $V(s) = known$
- For a non-terminal state

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

Example: tic-tic-tic game

- Tic-tic-tic is one person tic-tac-toe, with limit of 3 moves
- m = 3, average b = 8
- How do we calculate the V values?

How to act in a single player, deterministic, complete information game?

- Your policy should be: take the action for which the successor has the largest value.

$$\pi(s) = \underset{a}{argmax} \; V\left(T(s,a)\right)$$

- Is this now gameplay or planning?
- Actually, both! You can calculate a list of actions to the end of the game.

Two player, deterministic, zero-sum games

- Agents have opposite utilities: for each terminal state they add up to zero: $U(s, p_1) = -U(s, p_2)$
  - Eg. chess, go, etc.
- We can think of a single value that one of the agents maximizes and the other minimizes.
- Purely adversarial

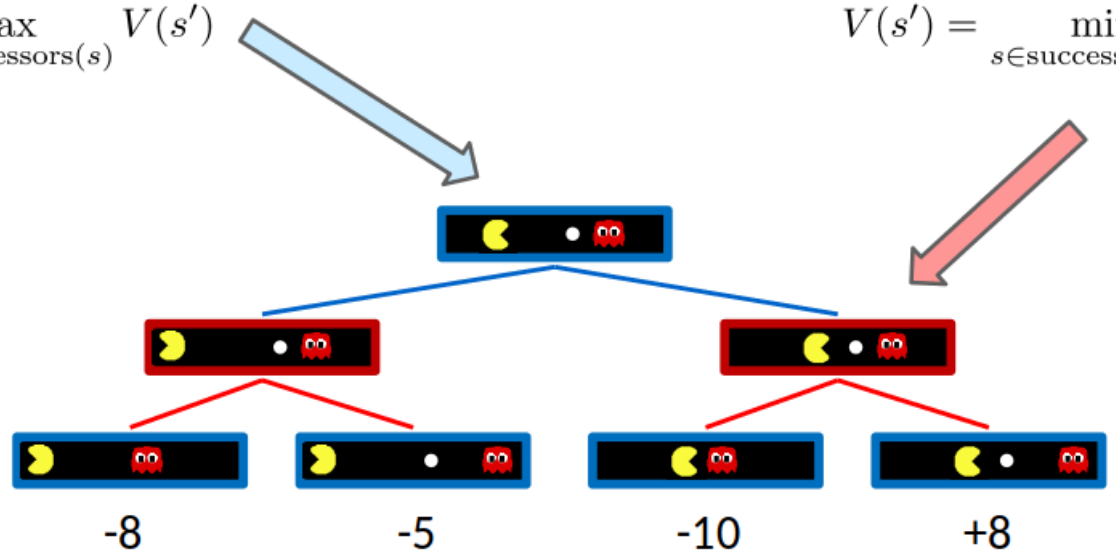
Zero sum game

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:

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

Adversarial search (minimax)

- Assume deterministic, zero sum games
- Player one maximizes the result, the other one minimizes it
  - We call it a maximizing player Δ and minimizing player ∇
- Minimax search tree
  - State-space search tree, with a V value
  - Players alternate turns, correspond to vertical layers in the tree

Minimax algorithm

```
def maxvalue(s)
  if s terminal return val(s)
  v = −∞
  for s' in succ(s)
    v = max (v, minvalue(s'))
  return v

def minvalue(s)
  if s terminal return val(s)
  v = ∞
  for s' in succ(s)
    v = min (v, maxvalue(s'))
  return v
```

Minimax example

- Tic-tac-toe - what is the value of this position?

```
  |  x  |  o
_________
  |  o  |
_________
x  |  x  |  o
```

Performance of minimax

- Similar to exhaustive DFS
  - Time $O(b^m)$
  - Space $O(bm)$
- It can solve any adversarial game, just not very efficiently
  - Chess: $b \approx 35, m \approx 100 \rightarrow 35^{100}$
  - Go: $b \approx 250, m \approx 210 \rightarrow 250^{210}$

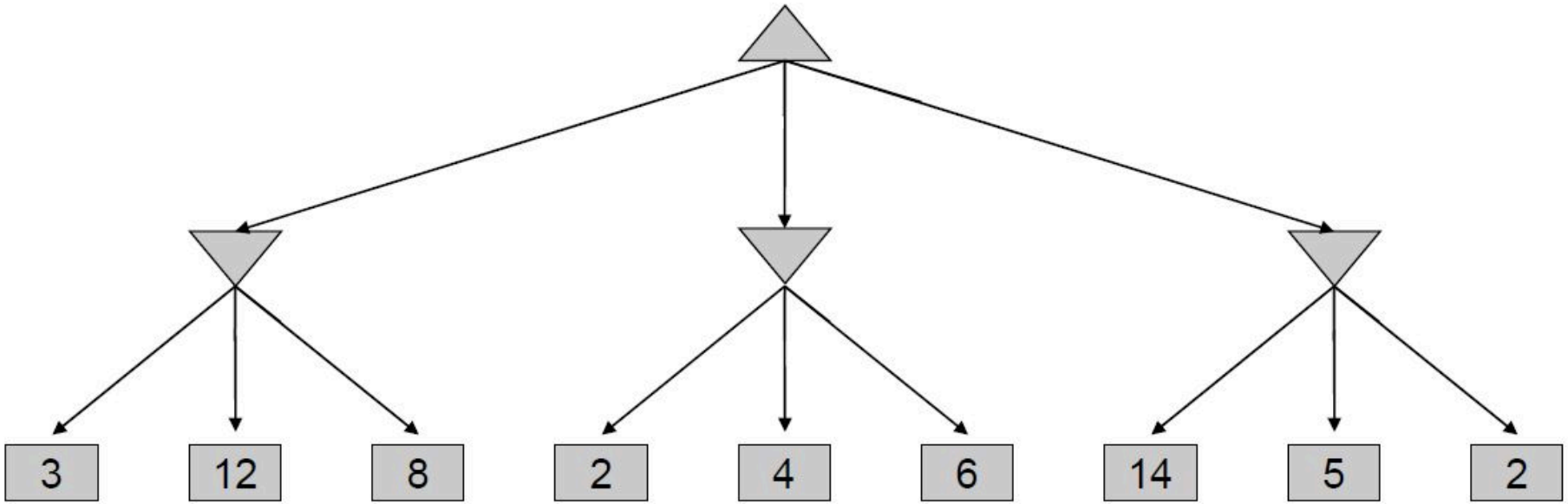
Game style of minimax

- It works perfectly against a perfect player.
- It also works perfectly against a non-perfect opponent
    - But this means that sometimes is too cautious

Alpha-beta pruning

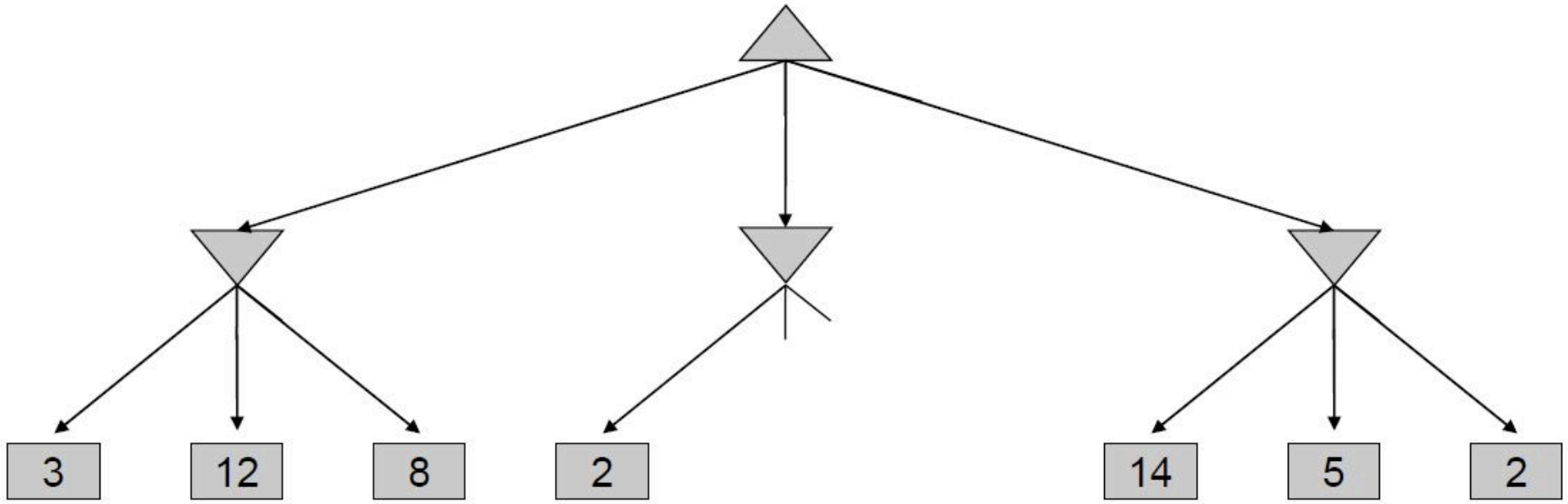
- Can we improve on the performance of minimax?
- For instance, do we always need to search the whole tree, down to all the leaves?
- **It helps us to know that the other player is our adversary:**
  - If we find that in a branch there is at least one very bad outcome, we can stop searching it.
  - Even if there are better outcomes in that branch, the opponent will not choose it!

Alpha-beta pruning: full minimax tree



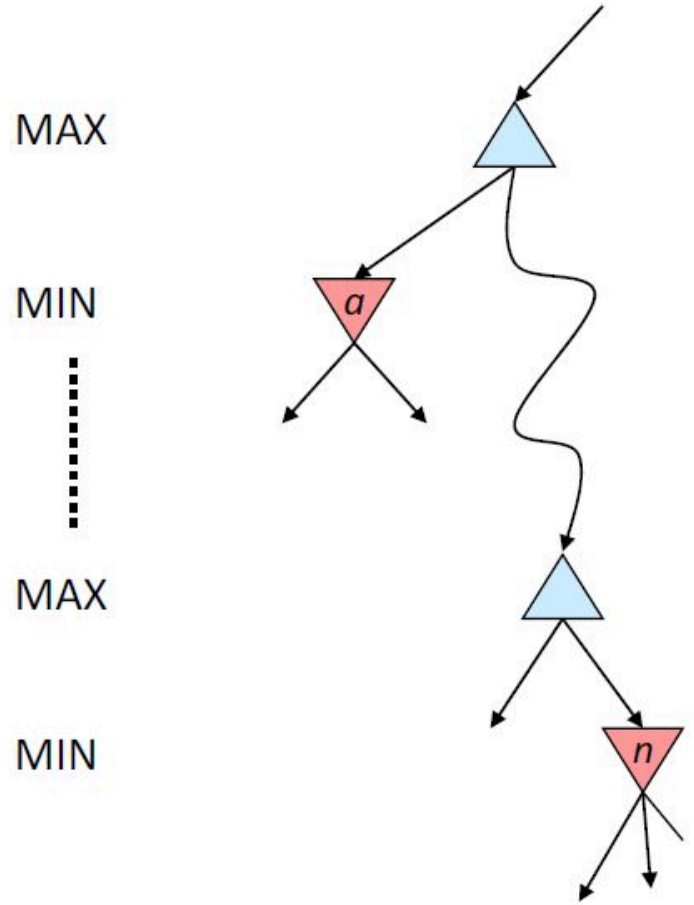
- There are situations when Max already knows that it will not choose a node!

Alpha-beta pruning: pruned tree



Alpha-beta pruning

- MIN version
  - We are computing the min-value at n
  - We are looping over n's children
  - n's estimate of the children's 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 n becomes worse than α, MAX will avoid it, so we can stop considering n's other children
- MAX version: symmetric, with β substituted for α



Alpha-beta implementation

```
def maxvalue(s, alpha, beta)
  if s terminal return val(s)
  v = −∞
  for s' in succ(s)
    v = max (v, minvalue(s', alpha, beta))
    if v >= beta return v
    alpha = max(alpha, v)
  return v

def minvalue(s)
  if s terminal return val(s)
  v = +∞
  for s' in succ(s)
    v = min (v, maxvalue(s', alpha, beta))
    if v <= alpha return v
    beta = min(beta, v)
  return v
```

Understanding alpha-beta pruning

- The min or max value of the *root* does not change.
  - But the values of nodes further down can change.
  - Even at the first level! Which is not good, because those values are the ones used for the action selection
  - Solution: start alpha-beta at the children of the top node.
- How much we prune depends on the order in which we investigate the children
  - You want to get the bad news to come soon, because then you can prune the rest.
  - Rich area for heuristics

What can be achieved with alpha-beta pruning

- If the ordering is perfect, time complexity drops from $O(b^m)$ to $O(b^{m/2})$
- Doubles the solvable depth!
    - From looking ahead 4 moves to looking ahead 8...
- Complexity remains exponential, complete search of chess or go are still hopeless

Resource limited search for minimax

- In practice, you can only search to a limited depth (*plies*)
  - one ply == one move by one of the players
  - Eg. 4 plies ahead in chess
  - More plies, better performance
- **Evaluation function**
  - This is what you return when you hit the limit, cannot search further.
  - It is an **informed guess** of the V value of the current state

Evaluation functions and cost

- The ideal evaluation function is the actual minimax value
- An evaluation function is always imperfect
  - If we can made an efficient and perfect evaluation function for a game, it is not much of a game.
- We can sometimes make evaluation functions better by expending more computation.
  - Cheap evaluation function in chess: add up the nominal piece values and return the difference between the white and black pieces
  - More expensive one: calculate the positional values of the pieces.
  - Very expensive one: look up the position in a library of famous games

Evaluation functions and depth

- It turns out that the deeper in the tree the evaluation function is, the less its quality matters.
- Tradeoff:
  - Cheap but weak evaluation function, go 8 plies deep?
  - Expensive but good evaluation function, go 2 plies deep?

How to build an evaluation function?

- You can use just about any function $eval(S) \rightarrow \mathbb{R}$
- Historically: ask an expert for a formula that explains the evaluation of a game state
- Example evalution function for chess:

$$Evaluation = (9 \times Queens) + (5 \times Rooks) + (3 \times Bishops\ and\ Knights) + (1 \times Pawns) + PositionalAdjustments$$

- This process is called **knowledge elicitation**
  - Very work-intensive, expensive and prone to major errors

Feature-based evaluation function

- New idea: don't ask the expert for a formula, just important **features**.
- A feature $f(s) \in \{0, 1\}$ or $f(s) \in [0, 1]$ is a property of a state which might or might not hold
  - $f(s)$ = is the black king checked?
  - $f(s)$ = damage incurred by a unit
- We assume that the evaluation is a weighted linear sum of features

$$eval(s) = w_1 \cdot f_1(s) + \cdots + w_n \cdot f_n(s)$$

Feature-based evaluation function (cont'd)

- What did we gain:
  - Easier for the expert to *list features* that matter, rather than *provide a formula*
  - Once we are done with the features, we can ask the expert for the weights
    - Can be positive or negative, small or large in absolute value

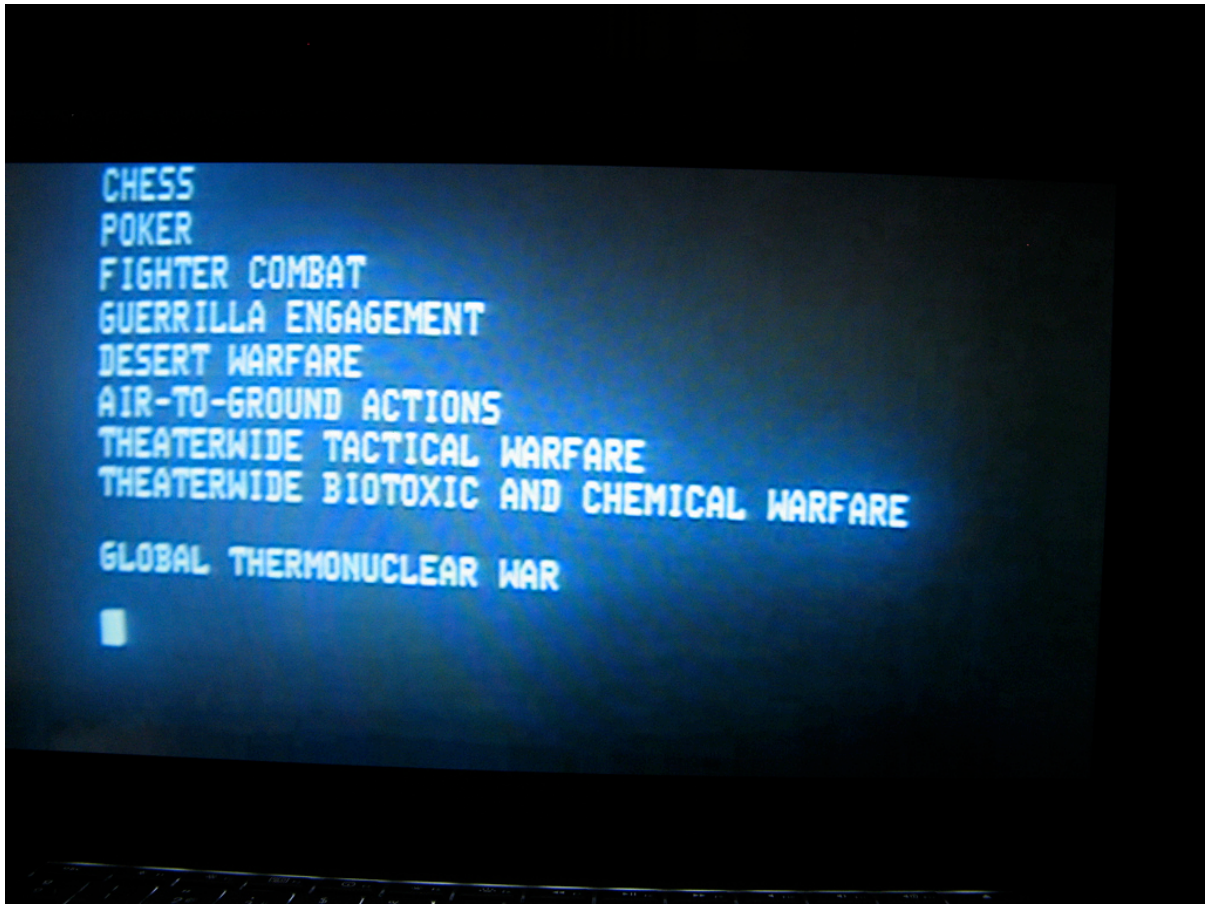
Learning feature weights

- **We can learn the weights!**
  - The features are the **input x** of a machine learning system
  - The **real V(s) value** is the **output y**
- It is not exactly trivial to get the real V(s):
  - we can try to run the search to completion… expensive in computer time, cannot be done for chess or go
  - ask the expert to evaluate the board… expensive in human time
  - use library of games by expert players… not a bad idea, for games where such records exist, like chess and go
  - we can try to play the game to the end… players might not be optimal

More about features

- The idea of features had been / is very influential in AI.
  - We will meet them again in reinforcement learning.
  - They had been very important in computer vision, speech recognition etc.
- General consensus circa 2010: *engineer* f, *learn* w
- Since 2012: *learn* f, *learn* w
- It is sometimes not easy to find the f in a large neural network, even if we know they should be there.
- **Explainable AI**: one way to explain what a system does is to know what features it takes into account
  - Sometimes, we don't want some features to matter: eg. race, gender, immigration status

Adversarial games are not only played on boards!




SHALL WE PLAY A GAME