Game play and adversarial search

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).

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, settles of Catan
  - Players relative strategy can be of 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.

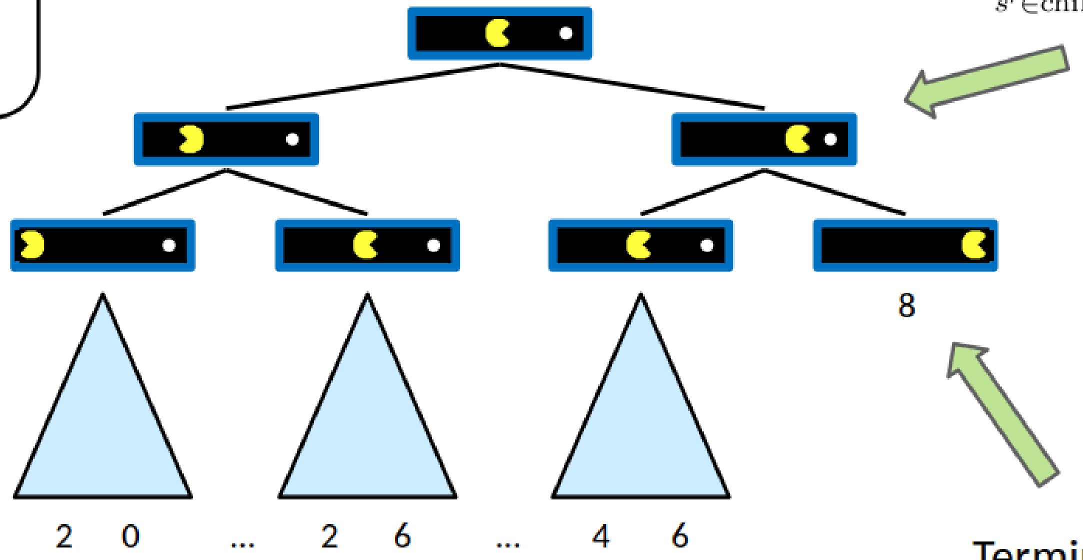
$$s_0 \rightarrow a_{A1} \rightarrow s_1 \rightarrow a_{B1} \rightarrow s_2 \rightarrow a_{A2} \rightarrow s_3$$

- Usually, we cannot search for a **plan**, because the agents' 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, such that you **maximize** the value 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 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.

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

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

Zero-sum games

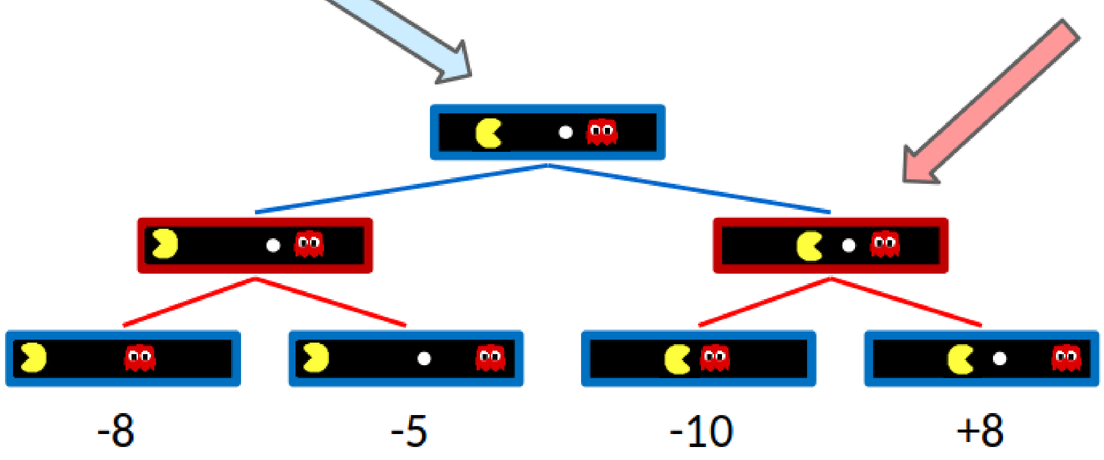
- Agents have opposite utilities: for each terminal state they add up to zero: $U(s, p_1) = -U(s, p_2)$
  - Eg. chess, go, etc.
- We can think of a single value that one of the agents maximizes and the other minimizes.
- Purely adversarial

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

Minmax algorithm

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

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

Minmax example

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

```
  | x | o
----------
  | o |
----------
x | x | o
```

Performance of minmax

- 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 minmax

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

Resource limited search for minimax

- In practice, you can only search to a limited depth (*plies*) - 1 ply == 1 move by one of the players
  - Eg. 4 plies ahead in chess
  - More plies, better performance
- When you reach the limit, you still have to return something, without searching further.
  - Return the value of an **evaluation function**
  - It is a way to evaluate the current state of the game without rolling out a search, for instance, by adding up the strenghts of the piece.

Evaluation functions and depth

- 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 (queen 9pts, rook 5 pts, bishop and knight 3 pts, pawn 1pt) and return the difference.
    - Cheap, not necessarily perfect
  - More expensive one: calculate the positional values of the pieces.
  - Very expensive one: look up the positions 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?

- Ideal function: actual minimax value.
- A convenient way to think about it: weighted linear sum of features

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

- $f()$ - hand engineered **features**
  - Eg. is the black king checked?
- w - weights, that can be manually set, or learned

Alpha-beta pruning