game theory

- classically, searching involves one agent trying to reach its objective
- games involve searching in the presence of an opponent
- games are represented like search problems:
    - states: game board configurations
    - operators/actions: allowed moves
    - initial state: *current* game state
    - end state: winning game state

sequential games

- players move one at a time
- have an heuristic $h(S)$ for how good a state is
- assuming a zero-sum game, we can use a single evaluation function for both players: f
    - the higher, the more advantageous to the computer
    - the lower (negative!), the more advantageous to the player
    - this evaluation function can be tricky to find

minimax

- a minimax tree involves nodes in a tree where MIN and MAX levels alternate
- computer tries to maximize MAX levels, opponent tries to minimize MIN levels
- select a depth limit and an eval function, and generate the tree up to n deep
- evaluate the leaves, propagate their value up
    - select minimum value for MIN
    - select maximum value for MAX
- for trees with chance-nodes (e.g.: dice throws), use the *expected values* for the chance-nodes
    - chance nodes before a MIN layer $$expectimin(A) = \sum_i(P(d_i) * minvalue(i))$$
    - chance nodes before a MAX layer $$expectimax(A) = \sum_i(P(d_i) * maxvalue(i))$$
    - basically, weighted sums

alpha-beta pruning

- an optimization for minimax
- cancel generation for parts of the tree that aren't needed
- the idea is to keep an estimate for each node based on the currently explored nodes below
    - α for MAX nodes, updated as descendants are explored
    - β for MIN nodes, updated as descendants are explored
    - if `parent.alpha >= self.beta`, stop exploring `self`, since `self.beta` only goes down
    - if `parent.beta <= self.alpha`, stop exploring `self`, since `self.alpha` only goes up
- for trees deeper than 4 levels, alpha-beta pruning applies to deeper levels too
    - look up the `alpha` or `beta` of not only the node above, but the nodes 3, 5, etc. above too - all of the `alpha`s or `beta`s on the way to the root
- other optimizations include:
    - *forward-pruning* - ignore sub-trees if they don't make sense to explore

        - too similar to ones encountered before?
        - have seemingly irrational moves?
        - only recommended at big depths, not near the root
    - searching with a time limit: return the best move *so far* when the time runs out

monte carlo tree search

- apply the following algorithm:

1. **selection**:
    - all nodes have a value: the ratio of wins passing through that node vs. total times passed through that node ($wins / passes$)
    - select the max ratio on each level to reach a node where statistics don't exist for all children
    - *upper confidence bound* (UCB1) selection maximizes the following expression instead: $$\frac{w_i}{n_i} + \sqrt{\frac{2\ln{n}}{n_i}}$$ *w_i = wins for child i, n_i = passes for child i, n = passes for self*
    - selection can pick a node that's already a terminal state
        - if it's a loss, give it a very small value to prevent its choice next time
        - if it's a win, give it a very large value, and give its parent a very small value

2. **expansion**:
    - applied when selection cannot continue
    - select a random unvisited leaf node and add a new record for each of its sons (0/0)
    - nodes with 0/0 will be selected first every time using UCB1, as they have infinite value

3. **simulation**:
    - after expansion, start simulation
    - do random moves until you get to a terminal state (win/loss)
    - this kind of simulation is also called *rollout* or *playout*
    - sometimes, instead of a completely random search, you can use weighting heuristics to choose "better" moves

4. **actualization** (or *retro-propagation*)
    - after simulation, increment passes (and wins) for all visited nodes
    - a win is incremented only on the nodes corresponding to the winning player
    - don't update down, just up from the node added in the expansion step to the root

- after repatedly applying the above algorithm, choose the move with the most passes, as its value is the best estimated
    - since it was explored most, its value must be great too
- after the computer and the opponent move, the corresponding sub-tree can be reused as initial values for the next iteration

MCTS vs minimax

- MCTS doesn't need heuristics
- MCTS is asymmetrical: exploration converges towards better moves

- MCTS is *anytime*: it can produce an estimate of the best move at any time

strategy games

- different from sequential games
- these games represent strategic, *simultaneous* interactions between rational agents
    - these agents are assumed to pick their actions to maximize their winnings
- elements:
    - at least 2 players or **agents**
    - each agent has a number of different **strategies** to use
    - the chosen strategies of each agent determine the **outcome** of the game
    - for every outcome, there is a numeric **payoff** for each player
- these elements can be represented by a *payoff matrix*, as seen below

famous example: prisonier's dillema

- 2 agents: 2 detained people
- both have the same strategies: confess to the crime, or deny involvement
- the agents choose their actions simultaneously, without knowing what the other agent is going to choose
- outcome: years of prison time
- payoff: less years $\rightarrow$ bigger payoff
- example payoff matrix:

	deny	confess
deny	-1, -1	-5, 0
confess	0, -5	-3, -3

- the first payoff in each cell is for *agent 1* (blue) and the second is for *agent 2* (red)

zero-sum games

- for any result of the game, the sum of agent payouts will be 0. example:

	C	D
A	2, -2	-1, 1
B	1, -1	-3, 3

- two-player zero-sum games can be represented by a simplified payout matrix:

	C	D
A	2	-1

B	1	-3

- the represented winnings are for the first agent (blue), and they are reversed for the second one

domination

- a strategy S dominates a strategy T of the same agent if any result of S is at least as good as the corresponding result in T
- a rational agent should never play a dominated strategy
- if every agent has a dominating strategy and plays it, their combination and the respective payoffs is called the game's **dominant strategy equilibrium**
- example:

	C	D
A	1, 3	4, 2
B	2, 4	7, 1

- B dominates A for agent 1, and C dominates D for agent 2
- this means that (B, C) will be the dominant strategy equilibrium
- not all dominations will be apparent at first
  - the **principle of higher order dominance**
  - cross out the dominated strategies
  - excluding the crossed-out strategies, more dominations will appear
  - repeat until equilibrium found
  - example:
    - D dominates E, cross out E
    - B now dominates A, cross out A
    - D now dominates C, cross out C
    - what's left is the equilibrium, (B, D)

	C	D	E
A	1, 1	2, 0	3, -1
B	2, 1	4, 3	2, 0

pure Nash equilibrium

- a combination of strategies is a **Nash equilibrium** if each player maximizes their winnings, given other player's strategies
- it identifies stable strategy combinations, in the sense that switching strategies will not yield better results unless the other agents do the same
- idk what the $u_i(s_i^, *s_{-i}*^) \ge u_i(s_i, s_{-i}^*)$ stuff is about but thats a nash equilibrium
  - it's strict if $\gt$ instead
- to find pure nash equilibriums:
  - write ${$ before the payoff for the first player if it's the best on the column
  - write $}$ after the payoff for the second player if it's the best on the line

- pure nash equilibriums will be between curly braces
- example:

	deny	confess
deny	-1, -1	-5, 0 }
confess	{ 0, -5	{ -3, -3 }