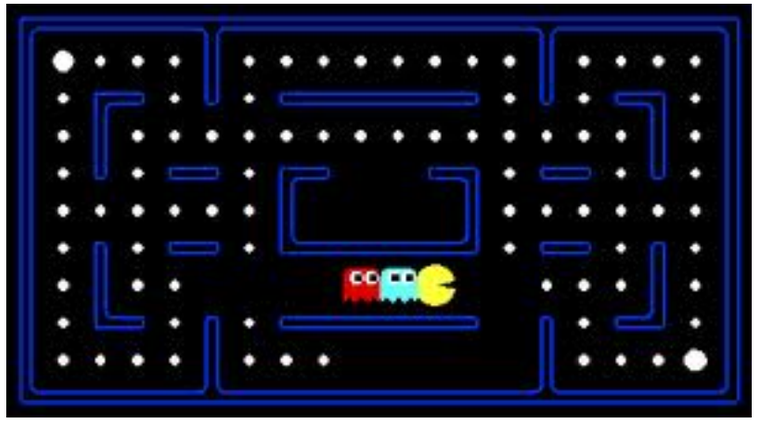
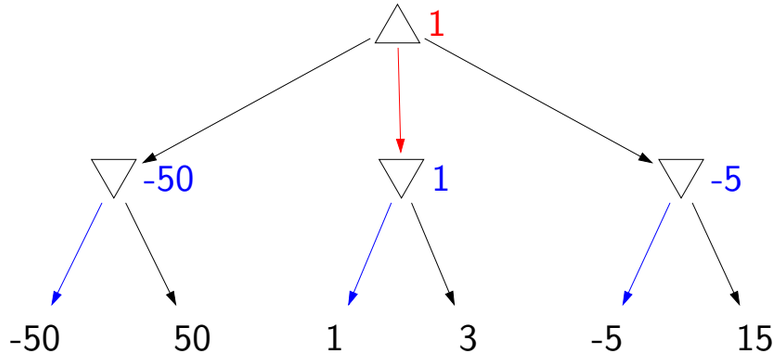

LELAND STANFORD JUNIOR UNIVERSITY
DIE LUFT DER FREIHEIT WEHT
1891

Games: recap



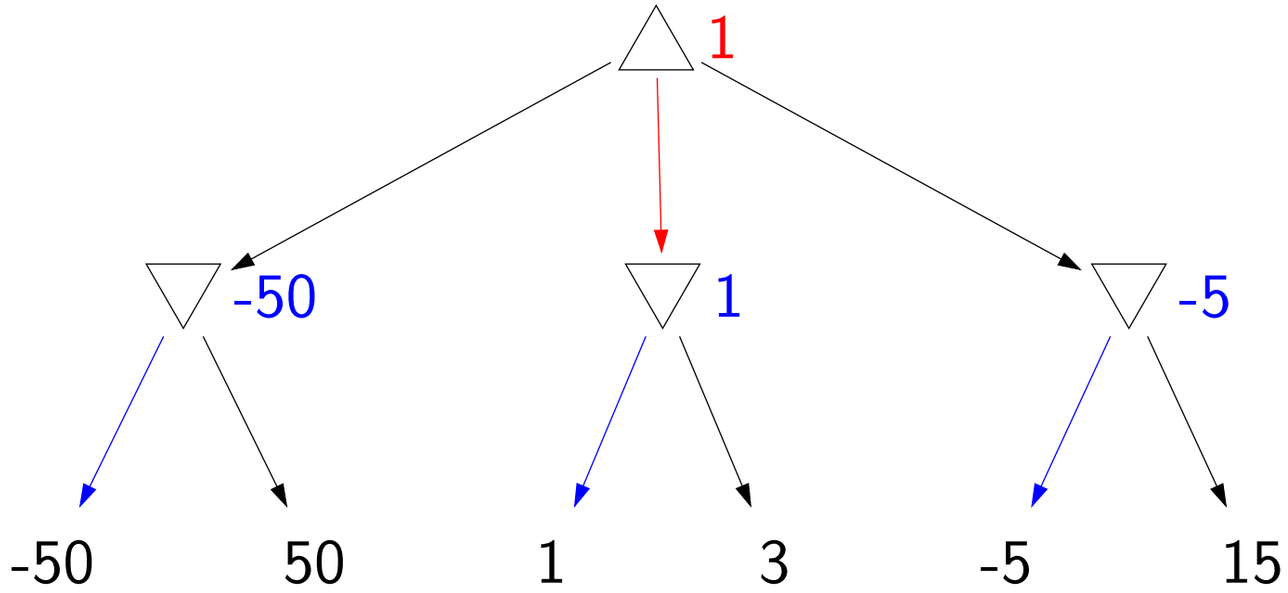
Summary



- Game trees: model opponents, randomness
- Minimax: find optimal policy against an adversary
- Evaluation functions: domain-specific, approximate
- Alpha-beta pruning: domain-general, exact

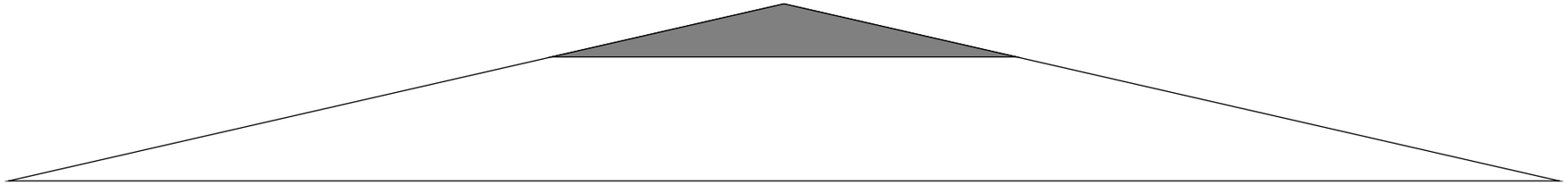
Review: minimax

agent (max) versus opponent (min)



- Recall that the central object of study is the game tree. Game play starts at the root (starting state) and descends to a leaf (end state), where at each node s (state), the player whose turn it is (Player(s)) chooses an action $a \in$ Actions(s), which leads to one of the children Succ(s, a).
- The **minimax principle** provides one way for the agent (your computer program) to compute a pair of minimax policies for both the agent and the opponent $(\pi^*_{\text{agent}}, \pi^*_{\text{opp}})$.
- For each node s, we have the minimax value of the game $V_{\text{minmax}}(s)$, representing the expected utility if both the agent and the opponent play optimally. Each node where it's the agent's turn is a max node (right-side up triangle), and its value is the maximum over the children's values. Each node where it's the opponent's turn is a min node (upside-down triangle), and its value is the minimum over the children's values.
- Important properties of the minimax policies: The agent can only decrease the game value (do worse) by changing his/her strategy, and the opponent can only increase the game value (do worse) by changing his/her strategy.

Review: depth-limited search



$$V_{\text{minmax}}(s, d) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a), d) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a), d - 1) & \text{Player}(s) = \text{opp} \end{cases}$$

Use: at state s, choose action resulting in $V_{\text{minmax}}(s, d_{\text{max}})$

- In order to approximately compute the minimax value, we used a **depth-limited search**, where we compute $V_{\mathsf{minmax}}(s, d_{\mathsf{max}})$, the approximate value of s if we are only allowed to search to at most depth d_{max}.
- Each time we hit $d = 0$, we invoke an evaluation function $\mathsf{Eval}(s)$, which provides a fast reflex way to assess the value of the game at state s.

Summary

- Main challenge: not just one objective
- Minimax principle: guard against adversary in turn-based games
- Simultaneous non-zero-sum games: mixed strategies, Nash equilibria
- Strategy: **search game tree + learned evaluation function**

- Games are an extraordinary rich topic of study, and we have only seen the tip of the iceberg. Beyond simultaneous non-zero-sum games, which are already complex, there are also games involving partial information (e.g., poker).
- But even if we just focus on two-player zero-sum games, things are quite interesting. To build a good game-playing agent involves integrating the two main thrusts of AI: search and learning, which are really symbiotic. We can't possibly search an exponentially large number of possible futures, which means we fall back to an evaluation function. But in order to learn an evaluation function, we need to search over enough possible futures to build an accurate model of the likely outcome of the game.

Chess

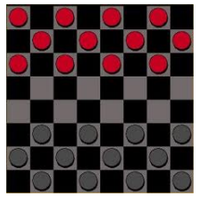
1997: IBM's Deep Blue defeated world champion Gary Kasparov

Fast computers:

- Alpha-beta search over 30 billion positions, depth 14
- Singular extensions up to depth 20

Domain knowledge:

- Evaluation function: 8000 features
- 4000 "opening book" moves, all endgames with 5 pieces
- 700,000 grandmaster games
- Null move heuristic: opponent gets to move twice



Checkers

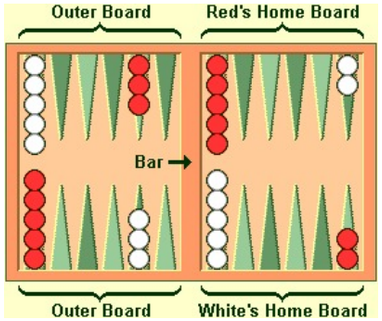
1990: Jonathan Schaeffer's **Chinook** defeated human champion; ran on standard PC

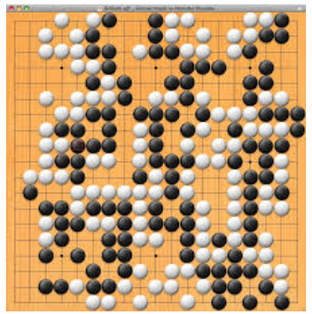
Closure:

- 2007: Checkers solved in the minimax sense (outcome is draw), but doesn't mean you can't win
- Alpha-beta search + 39 trillion endgame positions

Backgammon and Go

Alpha-beta search isn't enough...





Challenge: large branching factor

- Backgammon: randomness from dice (can't prune!)
- Go: large board size (361 positions)

Solution: learning

- For games such as checkers and chess with a manageable branching factor, one can rely heavily on minimax search along with alpha-beta pruning and a lot of computation power. A good amount of domain knowledge can be employed as to attain or surpass human-level performance.
- However, games such as Backgammon and Go require more due to the large branching factor. Backgammon does not intrinsically have a larger branching factor, but much of this branching is due to the randomness from the dice, which cannot be pruned (it doesn't make sense to talk about the most promising dice move).
- As a result, programs for these games have relied a lot on TD learning to produce good evaluation functions without searching the entire space.

AlphaGo

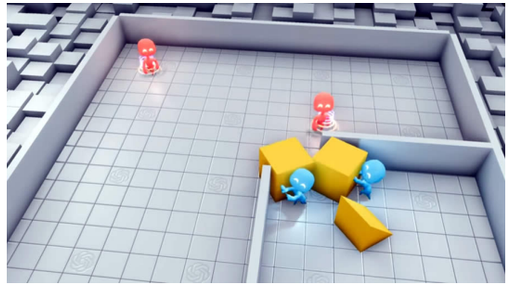
- Supervised learning: on human games
- Reinforcement learning: on self-play games
- Evaluation function: convolutional neural network (value network)
- Policy: convolutional neural network (policy network)
- Monte Carlo Tree Search: search / lookahead

- The most recent visible advance in game playing was March 2016, when Google DeepMind's AlphaGo program defeated Le Sedol, one of the best professional Go players 4-1.
- AlphaGo took the best ideas from game playing and machine learning. DeepMind executed these ideas well with lots of computational resources, but these ideas should already be familiar to you.
- The learning algorithm consisted of two phases: a supervised learning phase, where a policy was trained on games played by humans (30 million positions) from the KGS Go server; and a reinforcement learning phase, where the algorithm played itself in attempt to improve, similar to what we say with Backgammon.
- The model consists of two pieces: a value network, which is used to evaluate board positions (the evaluation function); and a policy network, which predicts which move to make from any given board position (the policy). Both are based on convolutional neural networks.
- Finally, the policy network is not used directly to select a move, but rather to guide the search over possible moves in an algorithm similar to Monte Carlo Tree Search.

Coordination games

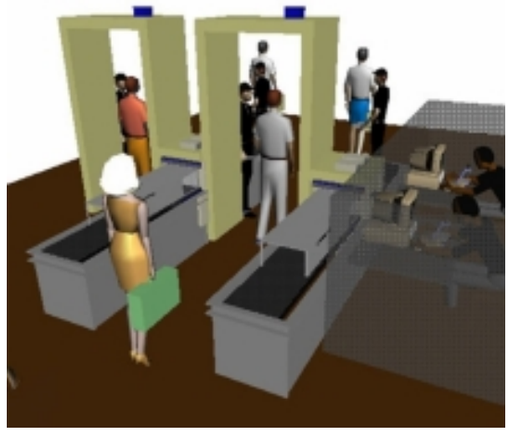
Hanabi: players need to signal to each other and coordinate in a decentralized fashion to collaboratively win.

Hide-and-Seek: OpenAI has developed agents with emergent behaviors to play hide and seek.



Other games

Security games: allocate limited resources to protect a valuable target. Used by TSA security, Coast Guard, protect wildlife against poachers, etc.

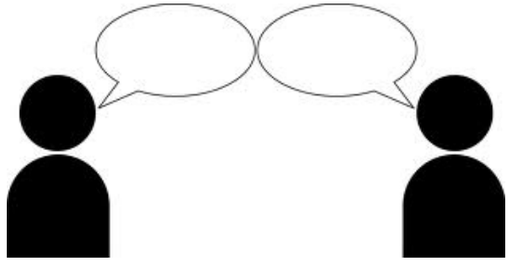


- The techniques that we've developed for game playing go far beyond recreational uses. Whenever there are multiple parties involved with conflicting interests, game theory can be employed to model the situation.
- For example, in a security game a defender needs to protect a valuable target from a malicious attacker. Game theory can be used to model these scenarios and devise optimal (randomized) strategies. Some of these techniques are used by TSA security at airports, to schedule patrol routes by the Coast Guard, and even to protect wildlife from poachers.

Other games

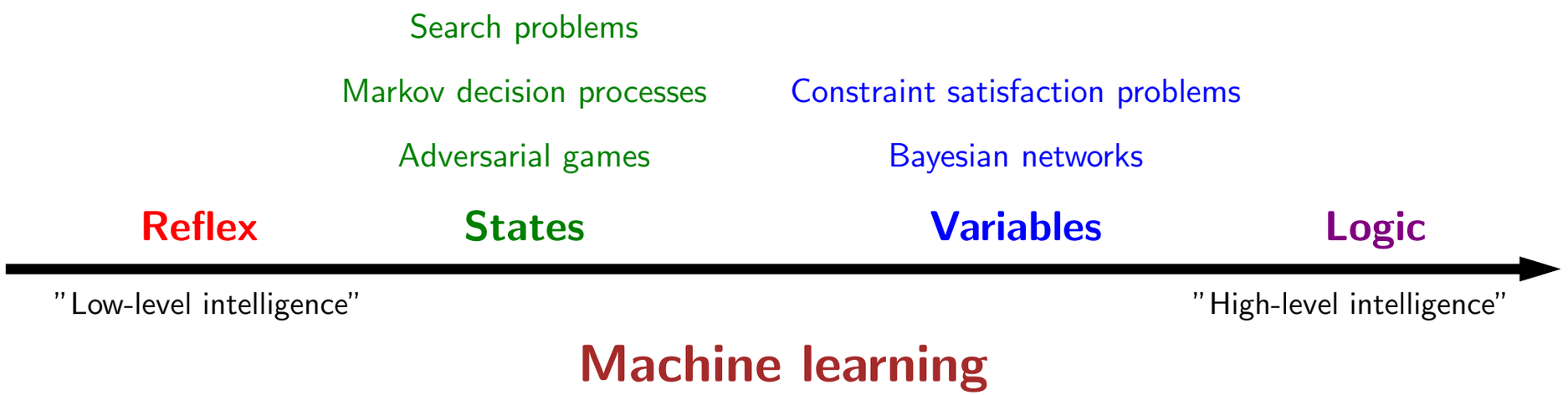
Resource allocation: users share a resource (e.g., network bandwidth); selfish interests leads to volunteer's dilemma

Language: people have speaking and listening strategies, mostly collaborative, applied to dialog systems



- For example, in resource allocation, we might have n people wanting to access some Internet resource. If all of them access the resource, then all of them suffer because of congestion. Suppose that if $n-1$ connect, then those people can access the resource and are happy, but the one person left out suffers. Who should volunteer to step out (this is the volunteer's dilemma)?
- Another interesting application is modeling communication. There are two players, the speaker and the listener, and the speaker's actions are to choose what words to use to convey a message. Usually, it's a collaborative game where utility is high when communication is successful and efficient. These game-theoretic techniques have been applied to building dialog systems.

Course plan


Search problems
Markov decision processes
Adversarial games
Constraint satisfaction problems
Bayesian networks
Reflex
States
Variables
Logic
"Low-level intelligence"
"High-level intelligence"
Machine learning