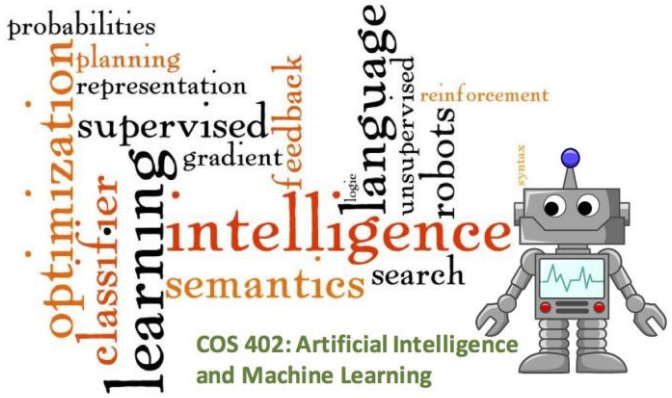


COS 402 – Machine Learning and Artificial Intelligence
Fall 2016

Decision-making (exact and approximate)

Sanjeev Arora Elad Hazan

PRINCETON UNIVERSITY

Plan for today

- Part 1: Rational choice theory. (Major achievement of first half of 20th century; quantifies rational decision-making for a computationally unlimited decision-maker. Important in economics, social sciences,..)

- Part 2: Decision-making when one has limited knowledge, or limited computation power.

(General theme of next few lectures: rational decision making.)

Rational Decision Making

Think about some important decision you took: e.g. which university to attend, which course to take,…

What did “rational decision-making” mean to you in those settings?

Rational Choice Theory: Ingredients

- We have an inbuilt "satisfaction"/"utility"/"reward" which seems to motivate us.
- We take a series of actions, and their consequences continue to unfold far into future.
- Our decision-making seeks to maximise reward/utility over this series of actions.

Reactions to this formalization? Objections?

(Theory runs into well-known controversies when applied to human decision-making, but OK as a framework for designing autonomous agents.)

Example: Cake Eating

- You receive a cake. It will go bad after 5 days.
- On any given day, if you eat x percent of the cake, you get utility $\sqrt{x}$
- How much cake should you eat each day?

If x_i percent eaten on day i (i=1, 2, 3,…), then net utility U = $\sqrt{x_1} + \sqrt{x_2} + .. + \sqrt{x_5}$
Since there is no utility in leaving cake uneaten, the x_i 's must sum to 100%.

Calculus says maximum achieved when $\partial U / \partial x_i$ is the same for all i (else can improve utility by reallocating).

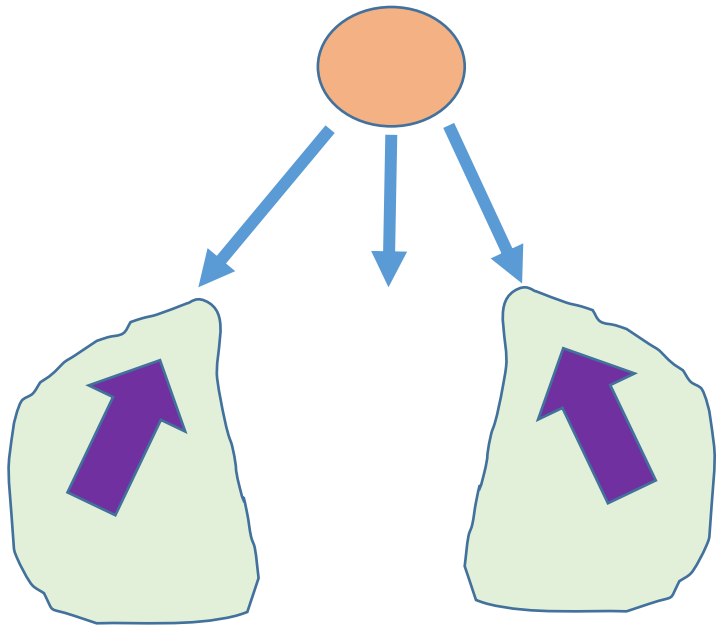
→ for x_i = 20% for all i.

Cake eating with sneaky roommate.

- You receive a cake. It will go bad after 5 days.
- On each day you are allowed to each any multiple of 20%. If you eat x percent of the cake, you get utility $\sqrt{x}$.
- Each night, with probability ½, your roommate eats 20% of the cake (if there is any left in the fridge).
- What is your optimum cake eating schedule now?

(handwritten slide from ipad)

Evaluating tree of all possibilities: dynamic programming



At each node, need to decide which of available options is the best.

Expand the tree from each child node and compute the utility obtained from optimum decision-making within each subtree.

Choose the decision which leads to best expected utility.

Rational Choice Theory (summary)

You as decision-maker know the following:

- Your utility/reward function.
- Set of possible actions at each step.
- Knowledge of possible events that may happen and affect your net utility; you know the distribution of these events ahead of time.
- You take actions that maximise your expected utility.

In CS often talk about minimizing "cost" instead of maximising utility.
("Cost" = - utility.)

Optimum decision can be computed using tree of all possibilities ("search tree.")
(But if T steps of actions, search tree may have size exponential in T)

Has been used in social science to study human decisions: why do people get married, how they save for retirement etc….

Subcase 1: Graph search.

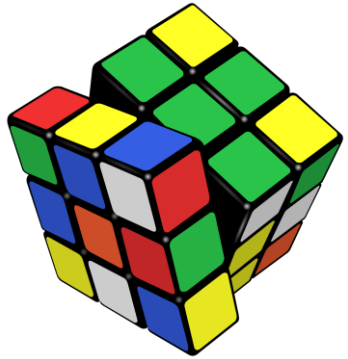
(Deterministic; no probabilistic events)

Input: Directed graph G = (V, E).

Start at some node s.
Looking for a node with a “jackpot.”
Basic set of actions: explore an outgoing edge from the node you happen to be at.

Application: Solving puzzles.

(What is the “graph” here?)



(Recap from Lecture 1)

A farmer must transport a fox, goat and box of beans from one side of a river to another using a boat which can only hold one item.

The fox cannot be left alone with the goat, and the goat cannot be left alone with the beans.

How to reason out a plan?

Ans: Write a search tree. Cost - ∞ if something eats something; cost 1 for each river crossing.

Graph Search (Iterated Depth First Search)

```
DFS-Search(u, t)
 /* u=start node, t=depth of
search*/
{ if u= jackpot, return “success”;
  if t=0; return failure.

 for all neighbors v of u,
  {       If Search(v, t-1);
          returns “success,” return
          success.
  }
Return failure.
}
```

Call DFS-Search(s, t) with t=1, 2, 3,…

Analysis

B = maximum degree of any node
d = minimum distance from s to a jackpot node.

Running time ≤ b^d .

Space ≤ O(d)

Recall: Memory is a more severe constraint than running time; computation must fit in RAM!

Application 2: Game playing

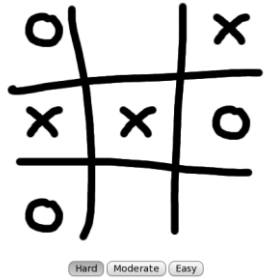
2 players. Take turns making moves.
At the end, win/lose some amount.
(Chess, Checkers, Tennis, …)

Important subcase: Zero-sum game
(what one player wins, the other loses)

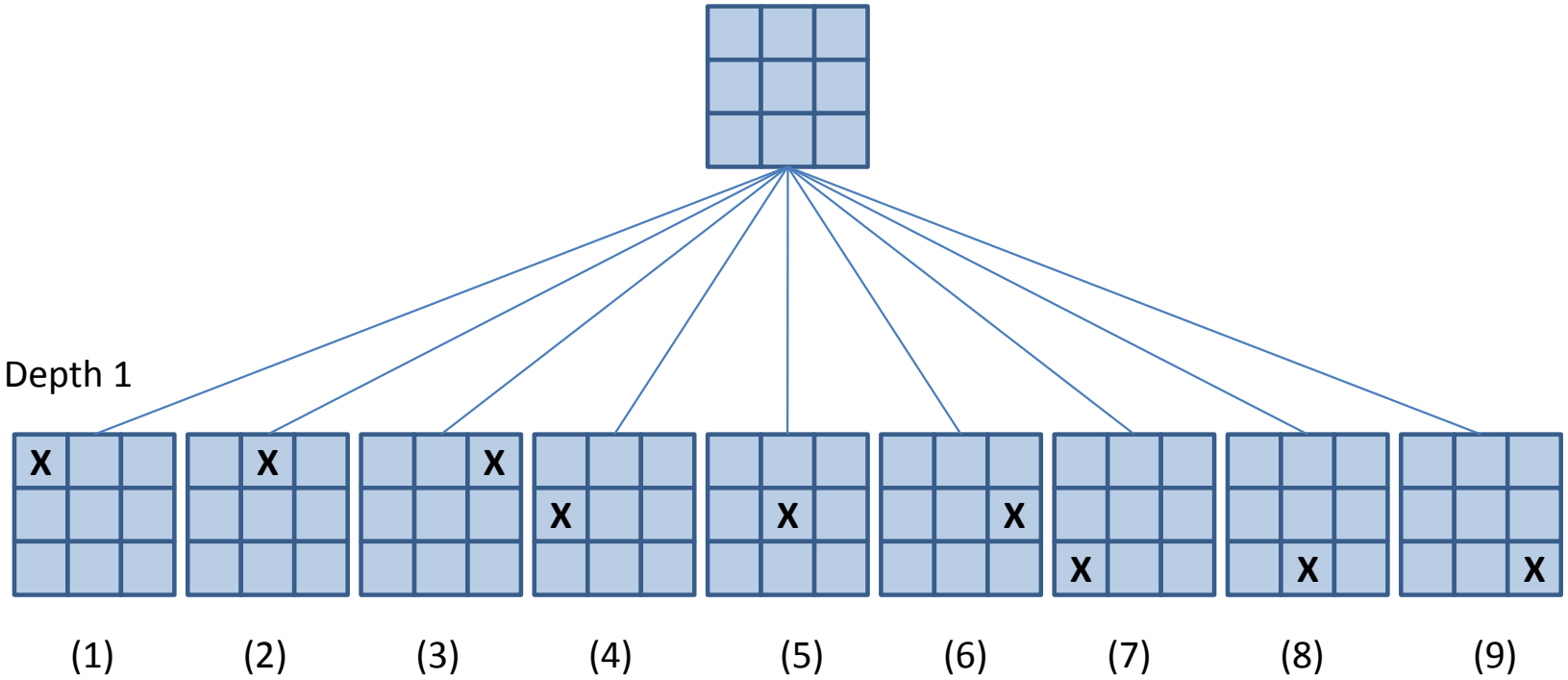
Example: Chess, Checkers, Tic-tac-toe etc.

Example of a game that is not zero sum: War between countries. (Both could be destroyed.)

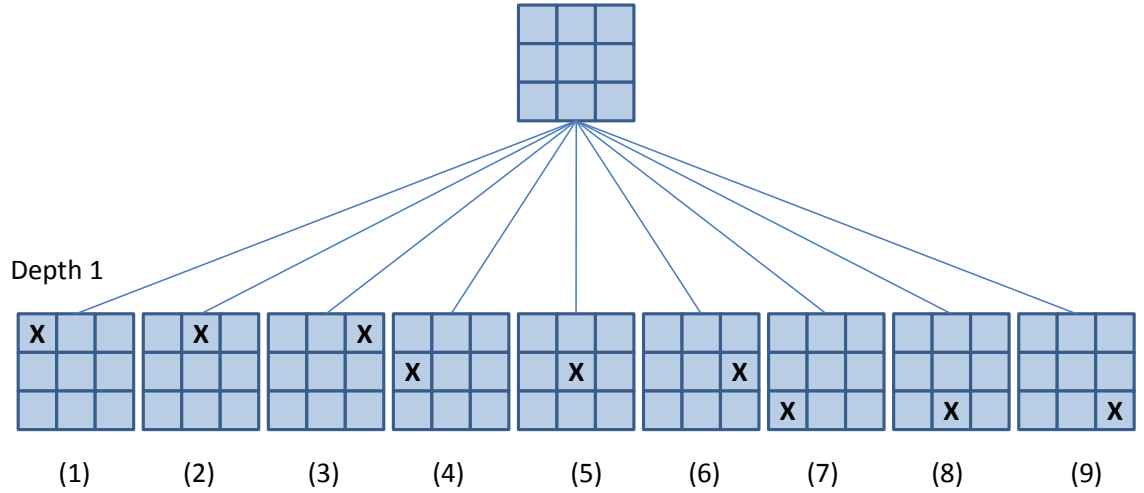
Game tree: tree of all possible moves.

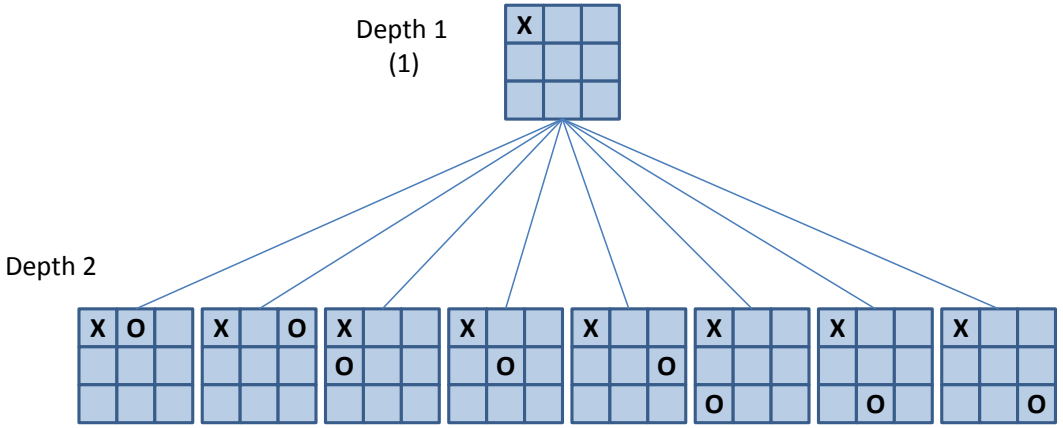


Example: Tic-tac-toe (loser gives $1 to winner; no money exchanged for a draw)



Minimax tree search for zero-sum games



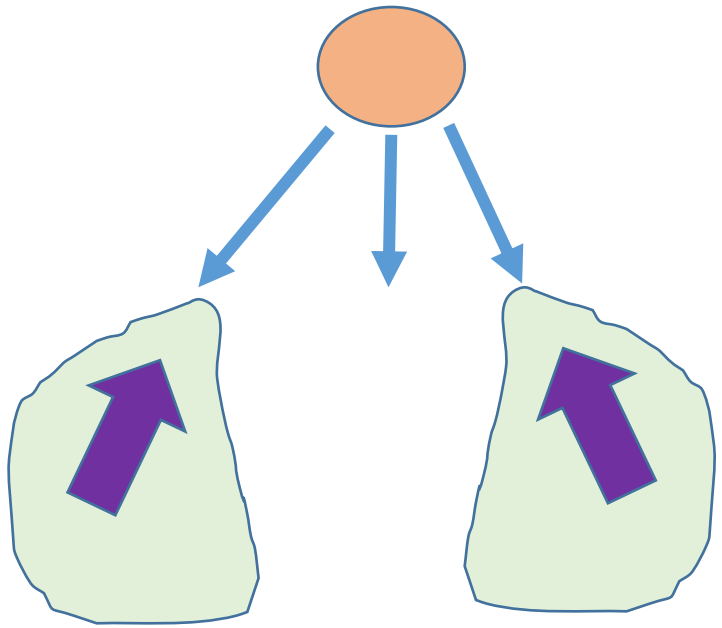


Minimax tree search.

Payoff = net payment by player 1 to player 2.

Player 1 picks tries to make moves that minimize payoff at the end, whereas Player 2 makes moves that maximise.

Minimax tree search (contd)



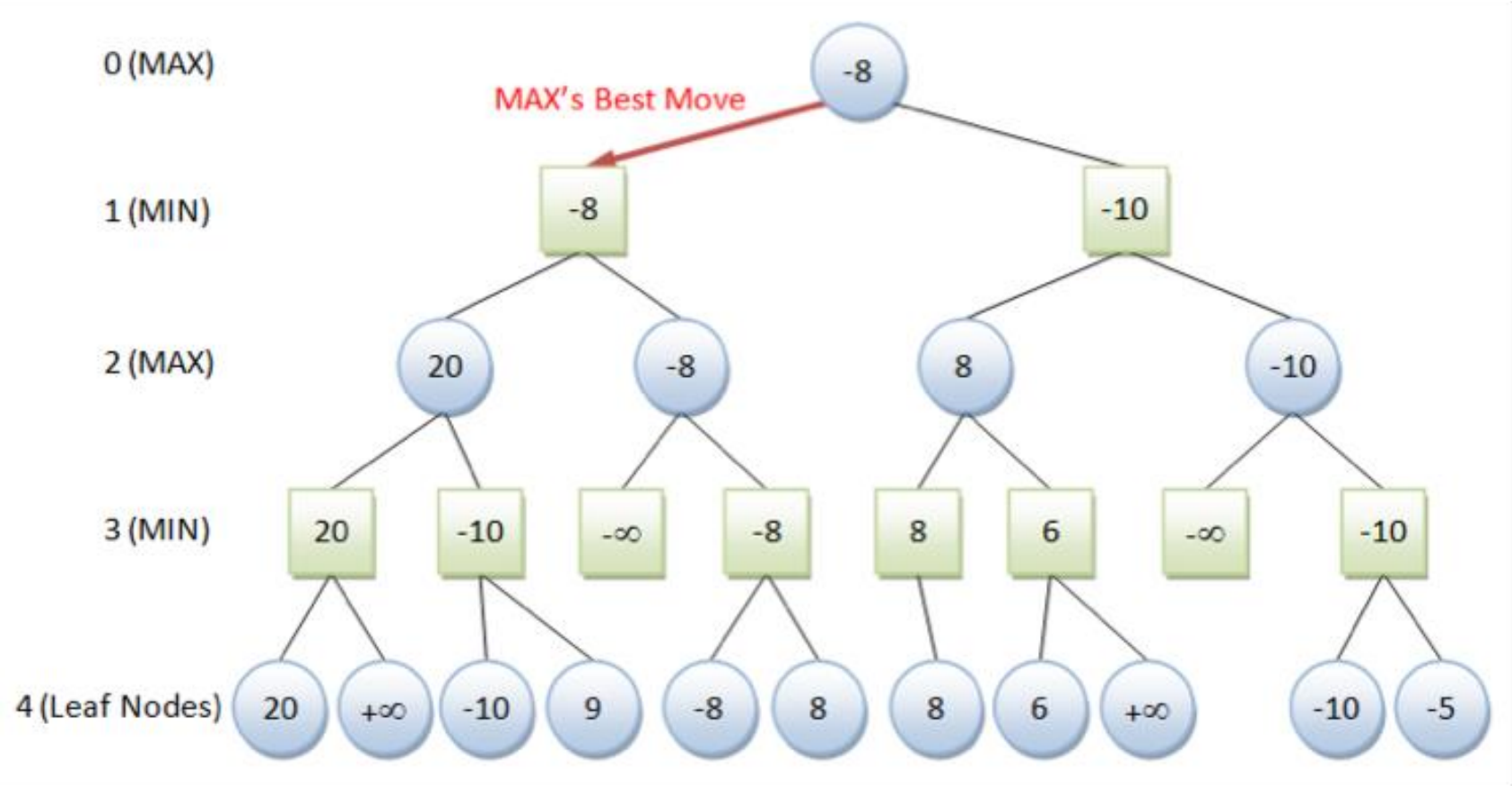
Payoff = net payment by player 1 to player 2.

At each node, need to decide which of available options is the best. Assume opponent plays optimally.

Player 1's turn: Expand tree from each child node and compute payoff obtained from optimum decision-making within each subtree. Choose move that minimizes payoff.

Player 2's turn: Expand tree, compute payoffs of subtrees. Pick move that maximises payoff. Choose move that maximizes payoff.

Example of minimax tree search



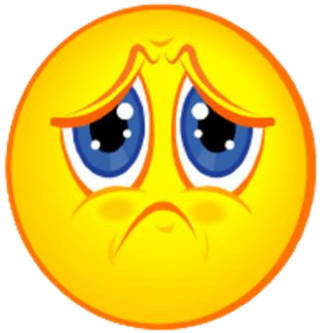
Part 2: Heuristic Search
(Making tree evaluation more efficient)

Size of minimax tree can be exponential in # of moves.

For tic-tac-toe: about 5 actions possible per move;
so 5^9 possible games. (approx 2 million; no big deal)

Chess: About 35 possible moves (estimated) at each time;
100 moves in a chess match.

➔ Search tree has size 35^{100} (> # of atoms in the universe!)



Approximation 1: Limited lookahead

Select next move by looking ahead only k moves. (i.e., truncate search tree)
Use a heuristic "valuation function" to rate the board position after k moves.

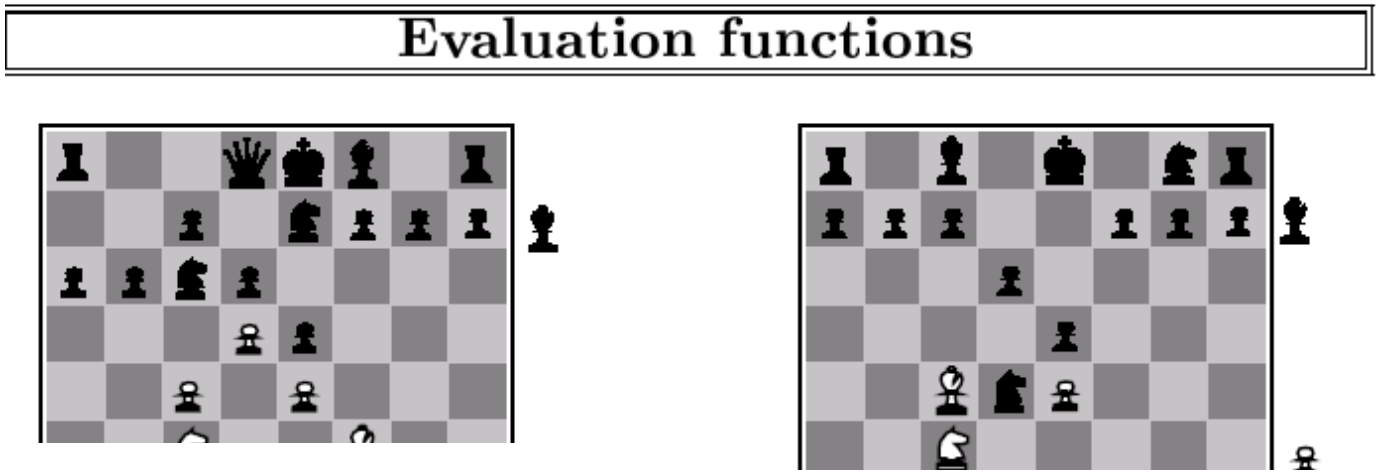
K = 4 (i.e., 2 moves by you and 2 by opponent). Does as well as novice player.
(35^4 = 1.5 million.)

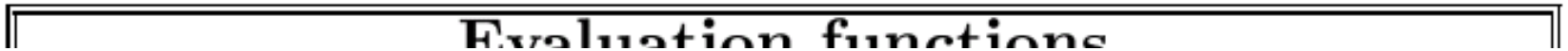
K= 8: Chess expert. (35^8 = 2.25 trillion; takes a few minutes on computer)

K=12: Grandmaster/champion level (35^{12} = infeasible for supercomputers)

Evaluation function

For chess, usually a linear weighted function of features.





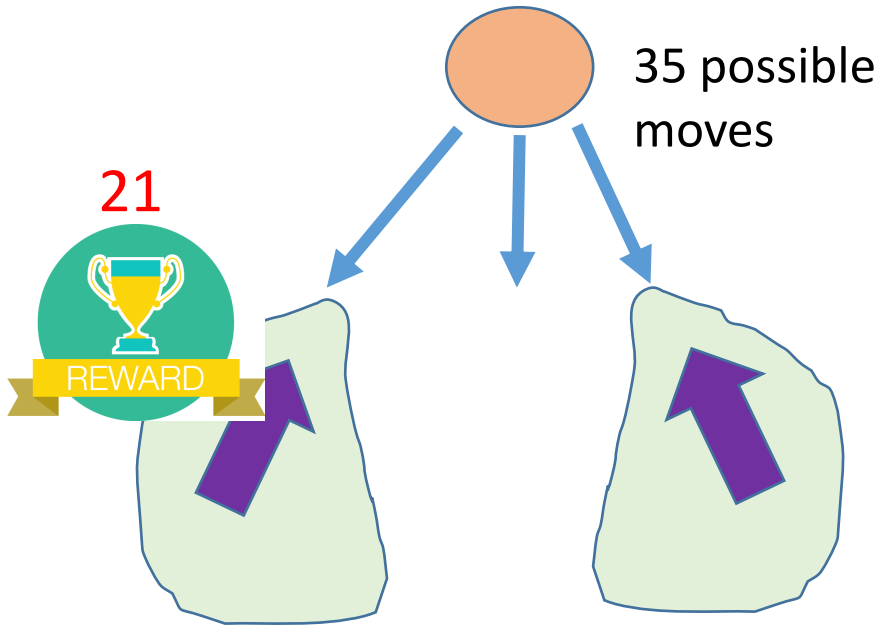
There could be a weighting for each piece remaining, for pawns close to promotion, pieces free to move, etc…

Alpha-Beta Search

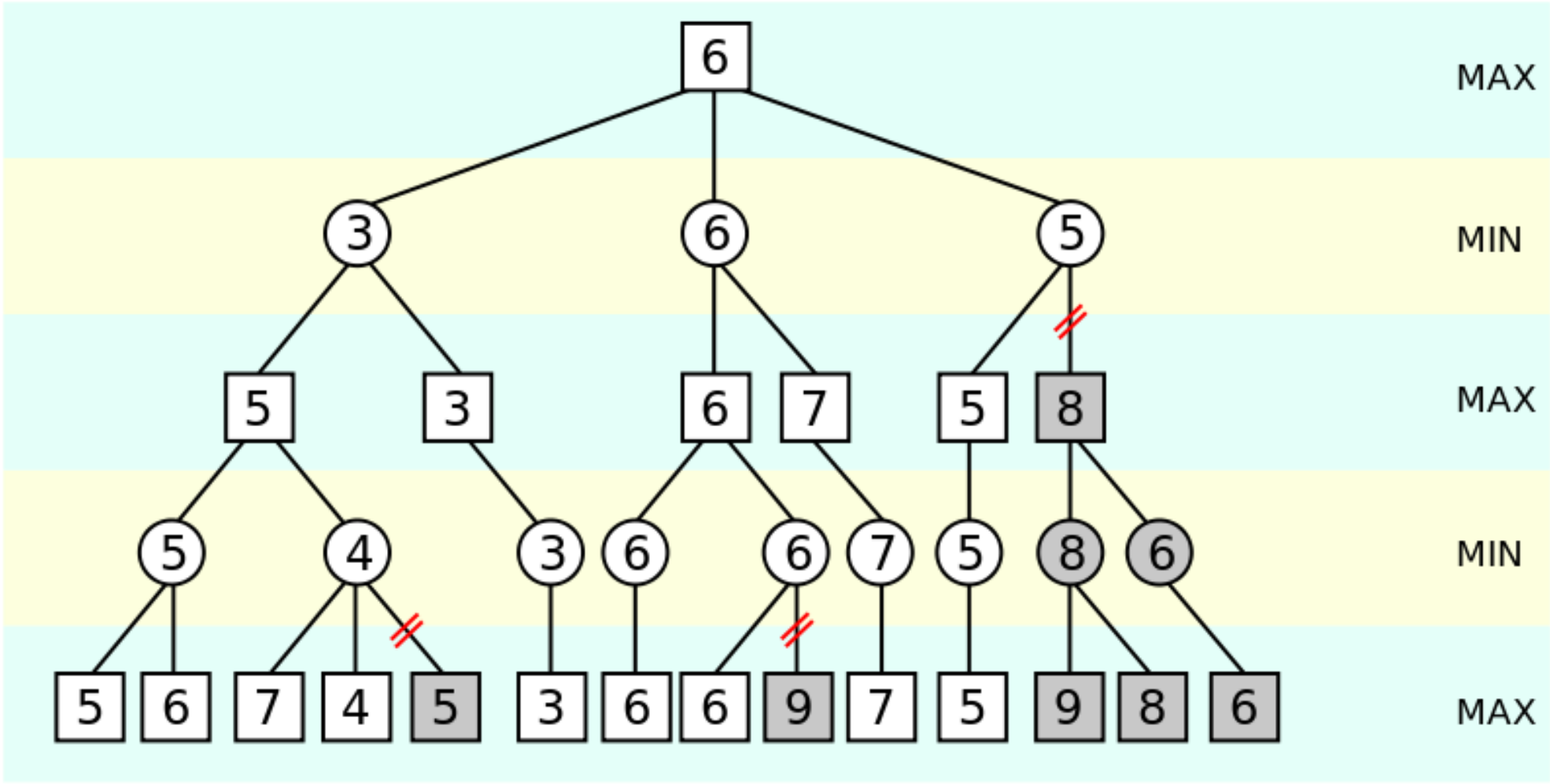
Suppose player is looking ahead k moves.

Exploring the tree rooted at first move showed a reward of 21 (e.g., captured the other player's queen)

Now it never pays to explore another subtree as soon as we determine that the total reward cannot be higher than 20.



Alpha-beta example.



Evaluated left to right.

Greyed -out nodes do not need to be explored

Iterative Deepening Search

- Go k steps; evaluate.
- Use alpha-beta to prune search tree.
- Go another few levels…

Many many extensions, as you can imagine.

Obvious generalizations of these ideas to games that depend upon chance (eg roll of dice, or randomly dealt cards)
(use expected reward instead of reward)

Next few lectures: Reinforcement Learning.
(decision-making by autonomous agents)