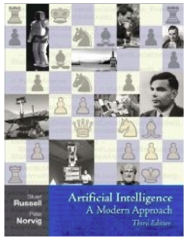
Adversarial Search and Reinforcement Learning

Chapters 5 & 22



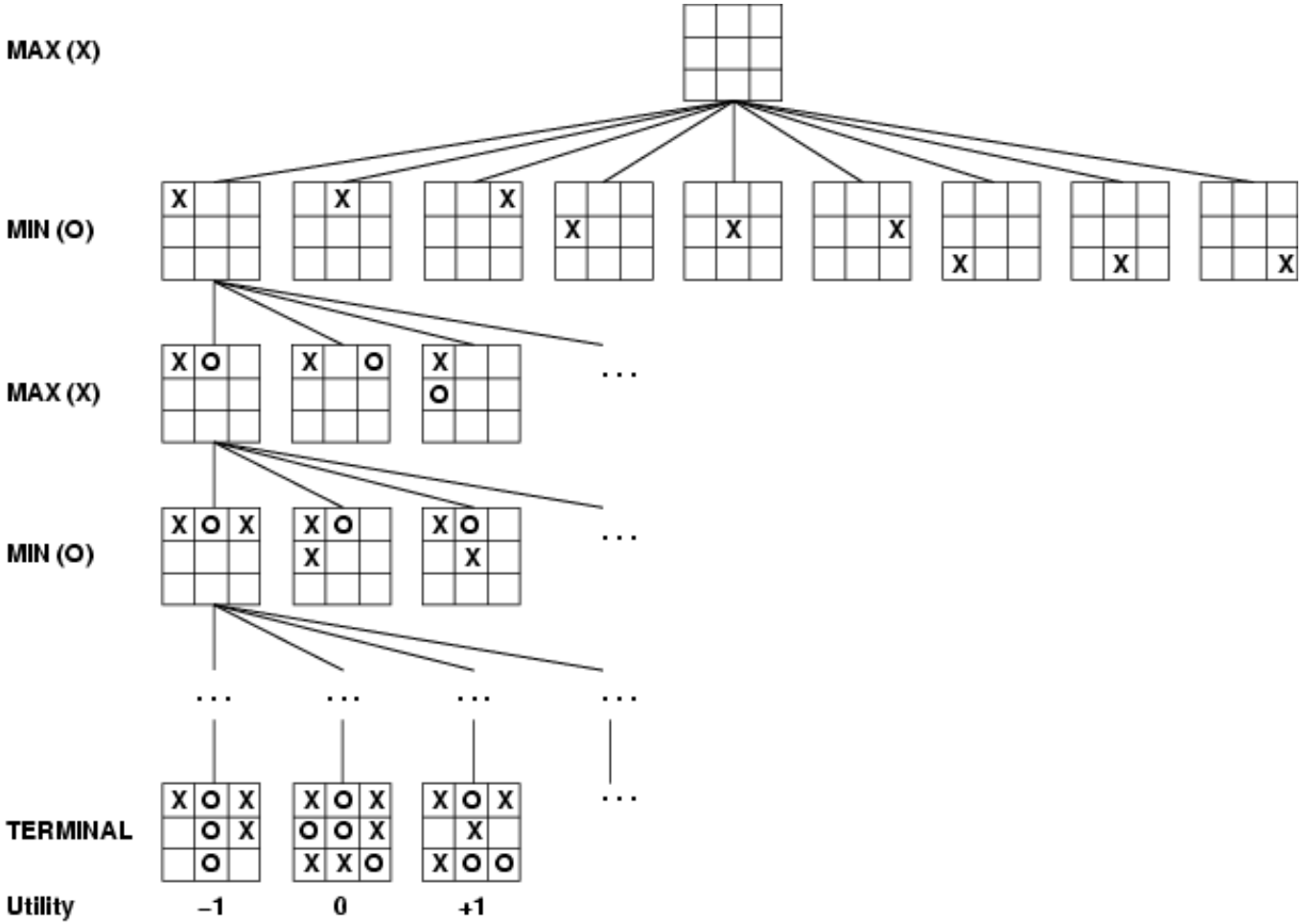
Many slides based on Russell & Norvig's slides Artificial Intelligence: A Modern Approach

Slides also due to Sriraam Natarajan, Matthew Taylor and Eric Sandholm

Games vs. search problems

- "Unpredictable" opponent → specifying a move for every possible opponent reply
  - Impossible
  - Unrealistic
- Time limits → unlikely to find goal, must approximate
- Most games are
  - Deterministic
  - Turn-taking
  - Two player
  - Zero-sum games
- Most real games are stochastic, parallel, multi-agent and utility based

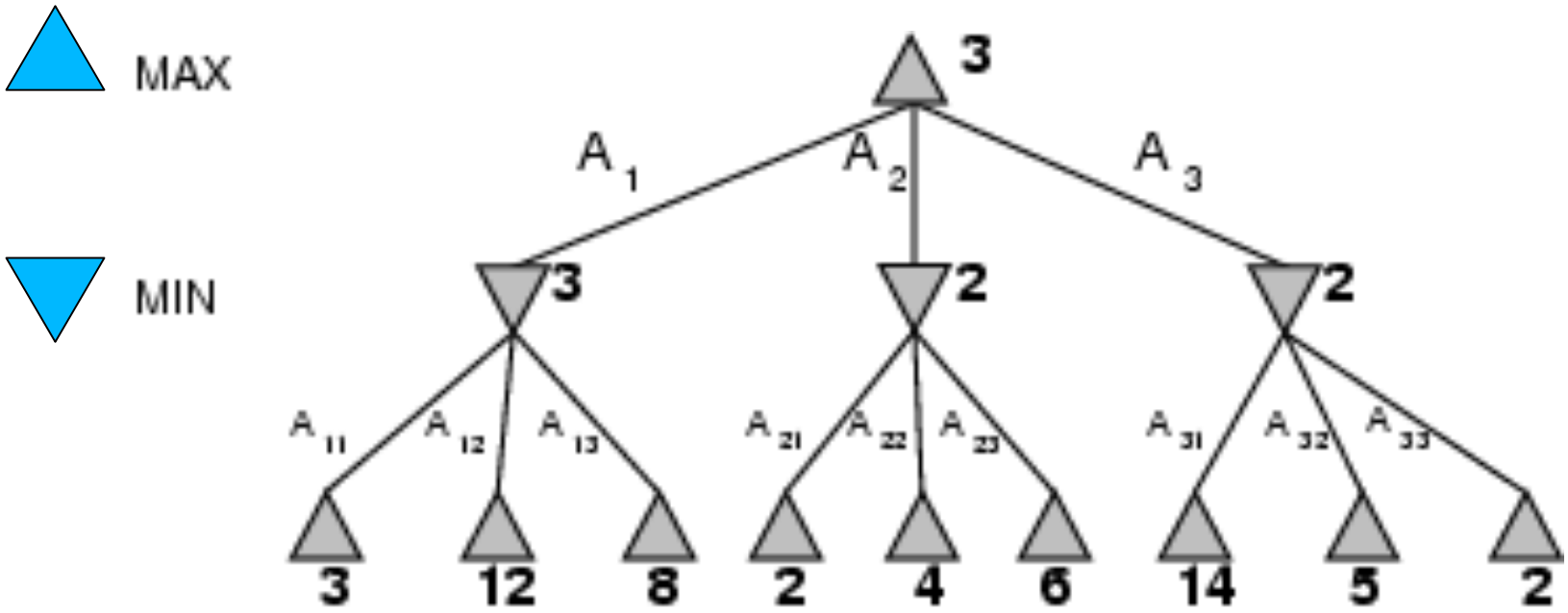
Game tree (2-player, deterministic, turns)



This is a tic-tac-toe game. The tree is supposedly small. Fewer than 9! = 362,880 terminal nodes. Chess, this number is 10^{40}.

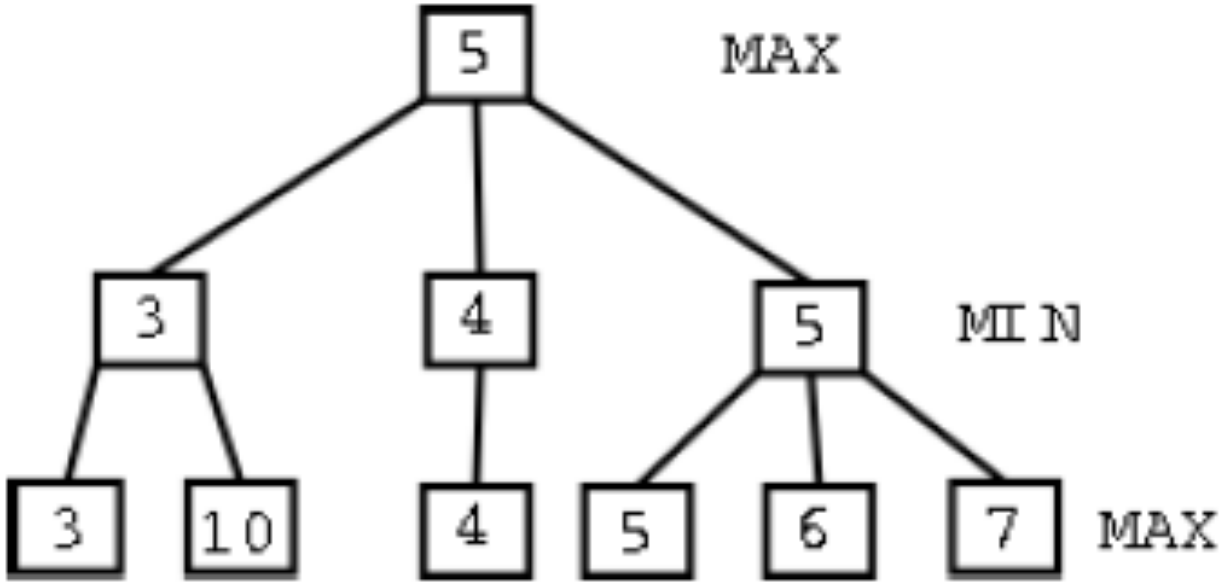
Minimax

- Perfect play for deterministic games
- Idea: choose move to position with highest minimax value
  = best achievable payoff against best play
- E.g., 2-ply game:



Example

- You are the “max” player
- Opponent is the min player
- (S)He wants to minimize your score
- You want to maximize
- Key assumption: (S)He is optimal
- How can this be extended to multiple players?
- Replace each utility with a vector of utilities corresponding to each player



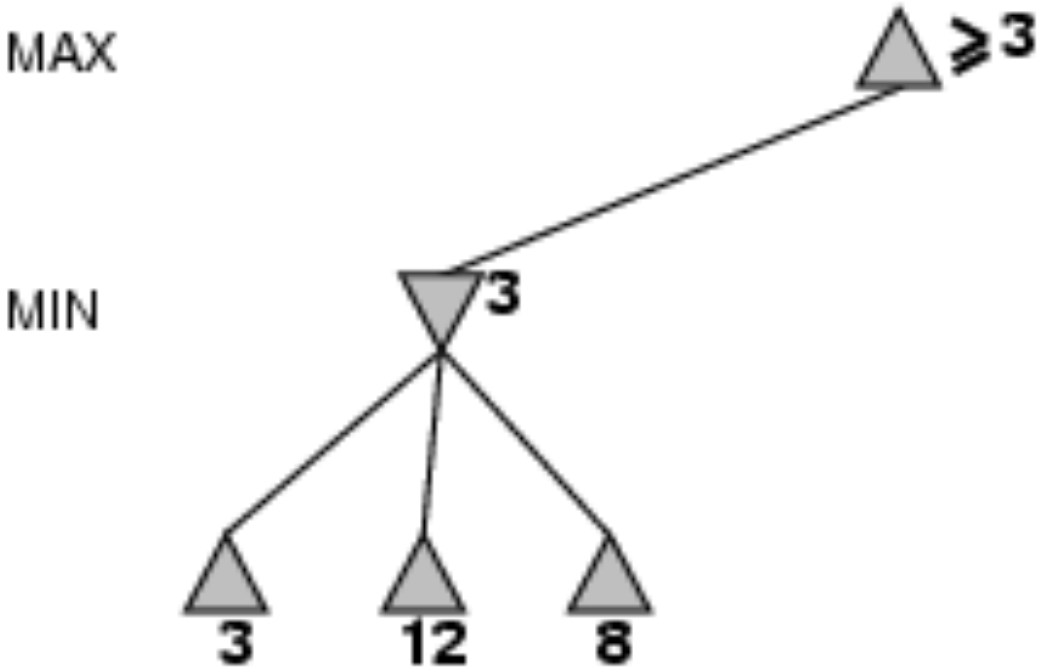
Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

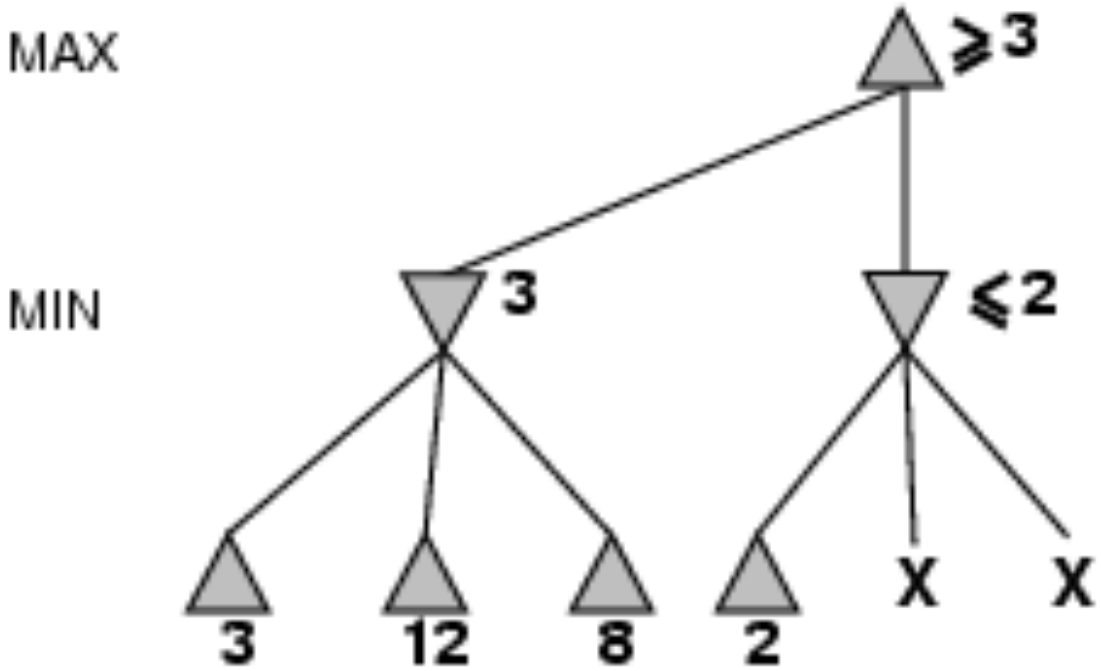
Properties of minimax

- Complete? Yes (if tree is finite)
- Optimal? Yes (against an optimal opponent)
- Time complexity? $O(b^m)$
- Space complexity? O(bm) (depth-first exploration)
- For chess, b ≈ 35, m ≈100 for "reasonable" games
  → exact solution completely infeasible

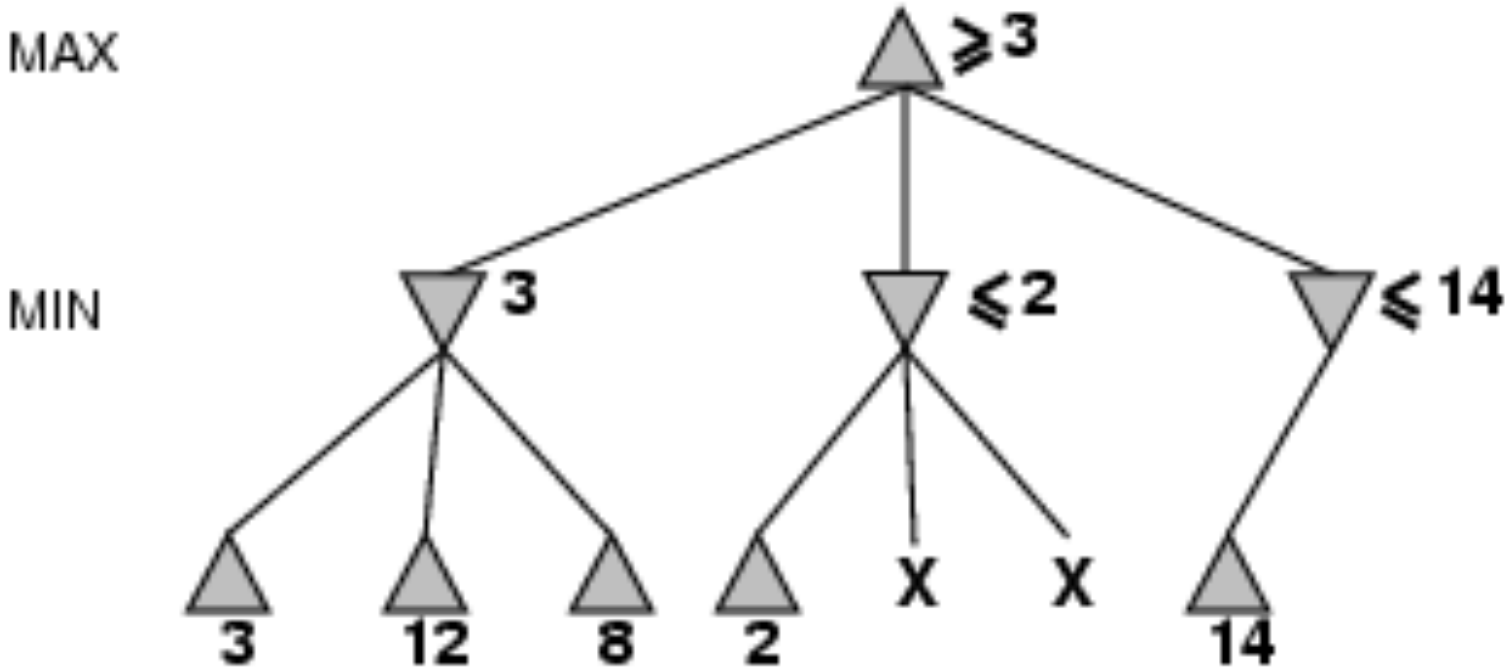
α-β pruning example



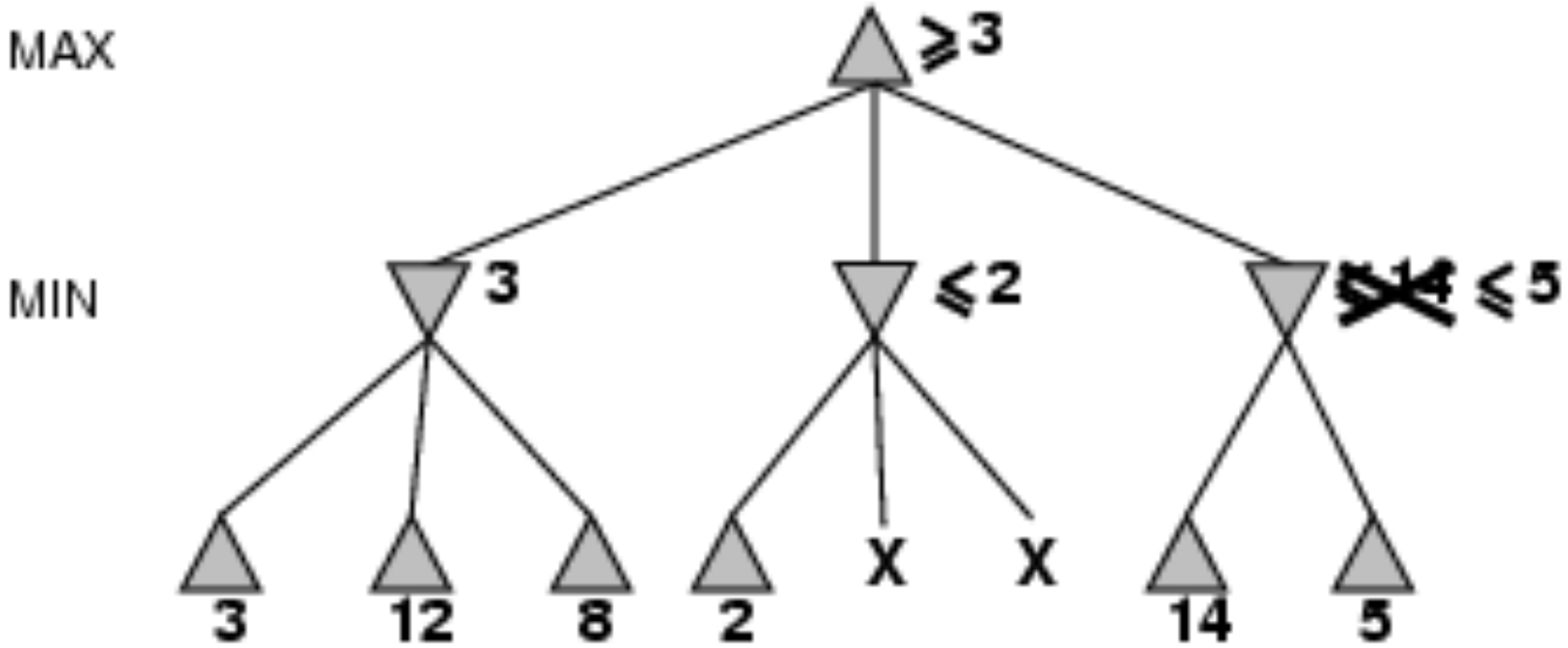
α-β pruning example



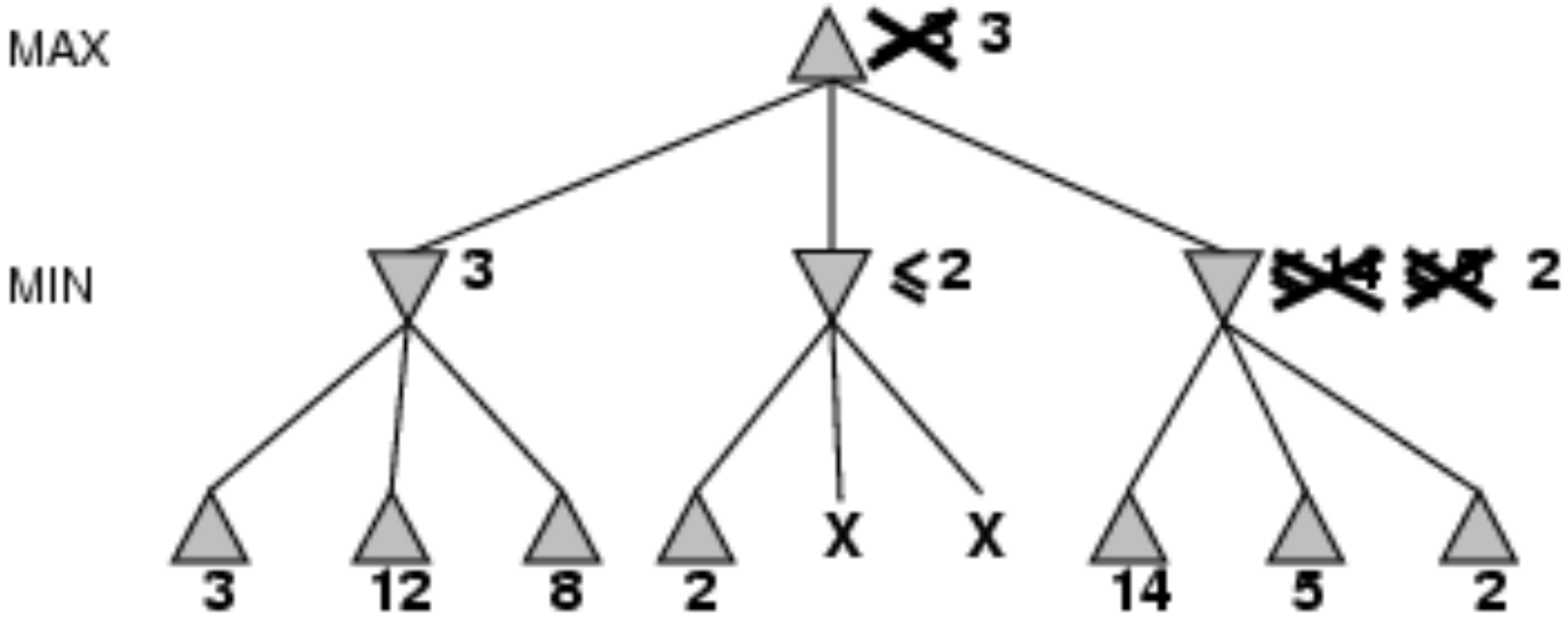
α-β pruning example



α-β pruning example

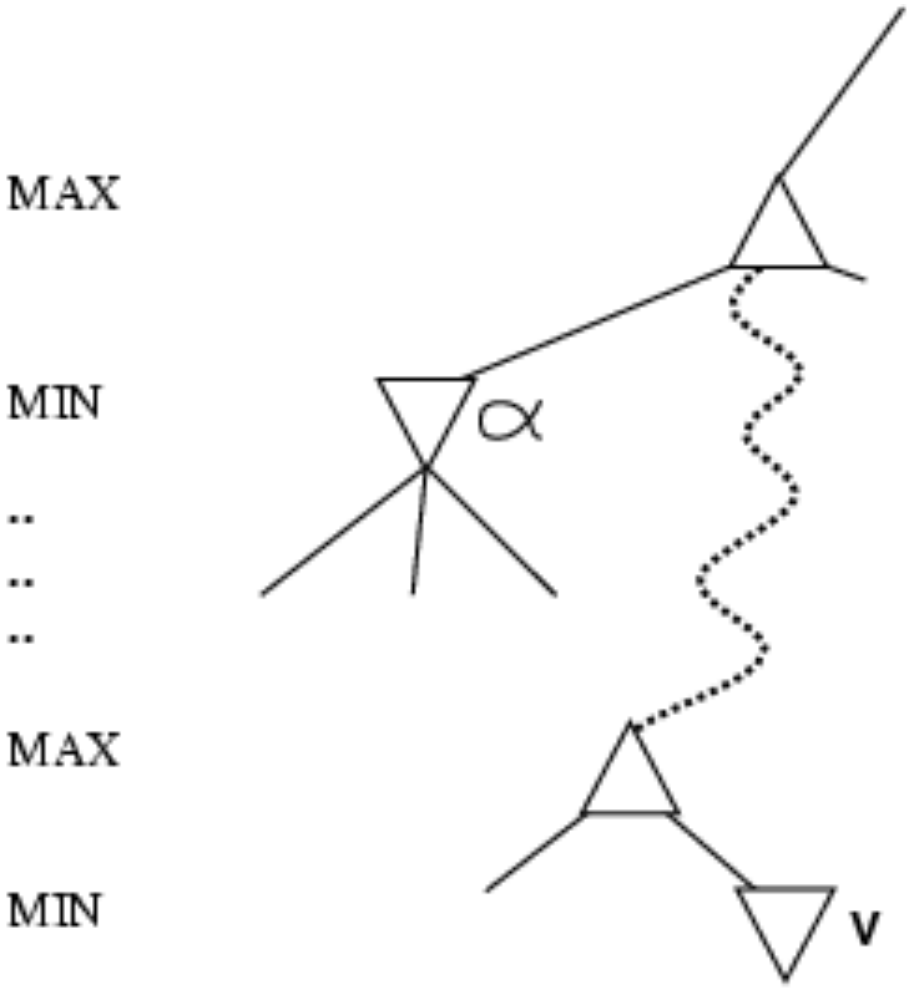


α-β pruning example



Why is it called α-β?

- α is the value of the best (i.e., highest-value) choice found so far at any choice point along the path for max
- If v is worse than α, max will avoid it → prune that branch
- Define β similarly for min



The α-β algorithm

function ALPHA-BETA-SEARCH(*state*) **returns** *an action*
  inputs: *state*, current state in game

  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in SUCCESSORS(*state*) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  inputs: *state*, current state in game
    α, the value of the best alternative for MAX along the path to *state*
    β, the value of the best alternative for MIN along the path to *state*

  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for a, s in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(s, α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

The α-β algorithm

function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  inputs: *state*, current state in game
    α, the value of the best alternative for MAX along the path to *state*
    β, the value of the best alternative for MIN along the path to *state*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for a, s in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(s, α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Evaluation functions

Nevertheless, we need a way to evaluate situations/states

- For chess, typically linear weighted sum of features

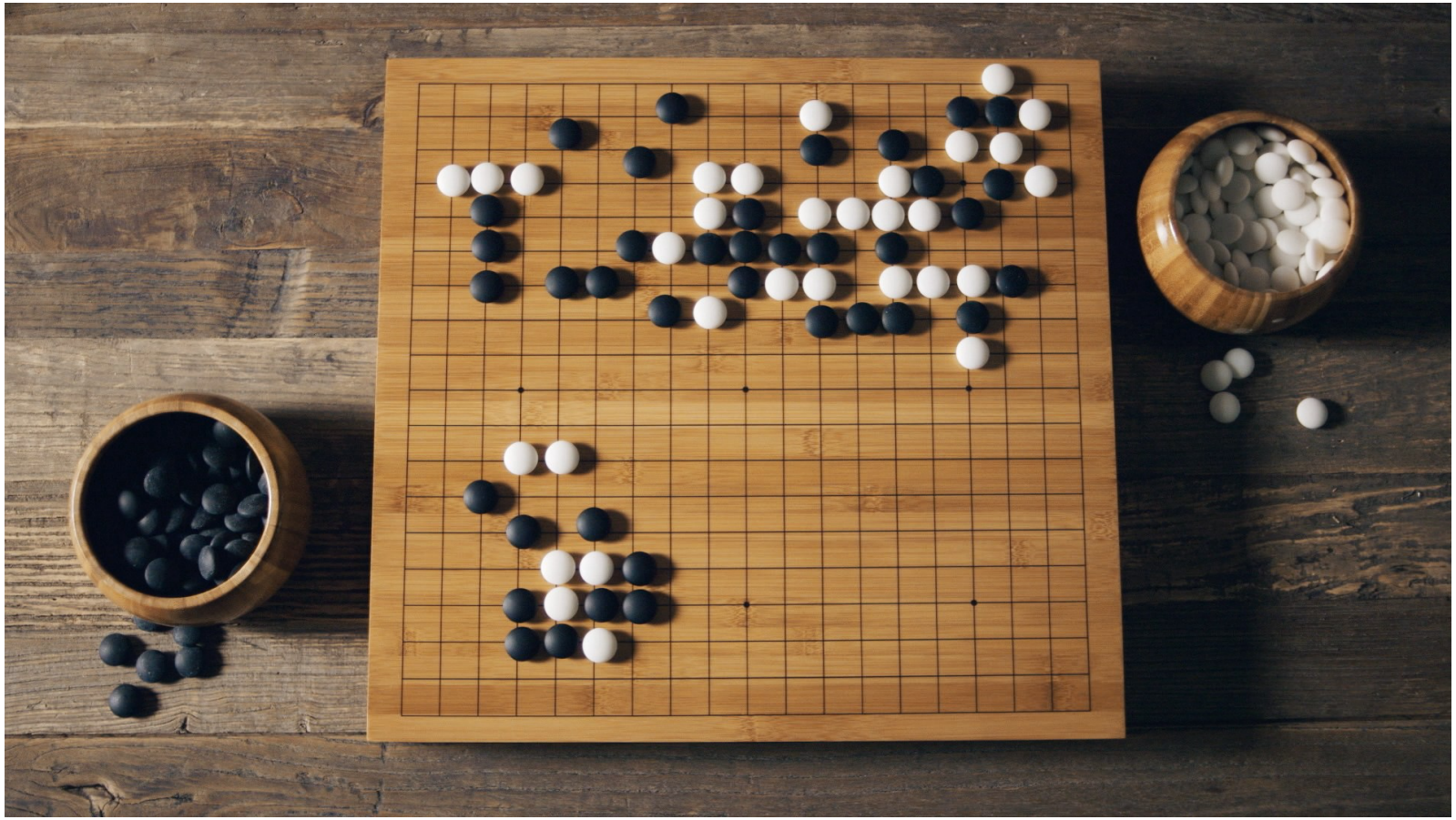
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- e.g., $w_1 = 9$ with

  $f_1(s)$ = (number of white queens) – (number of black queens), etc.

Where can we find deterministic games in practice?

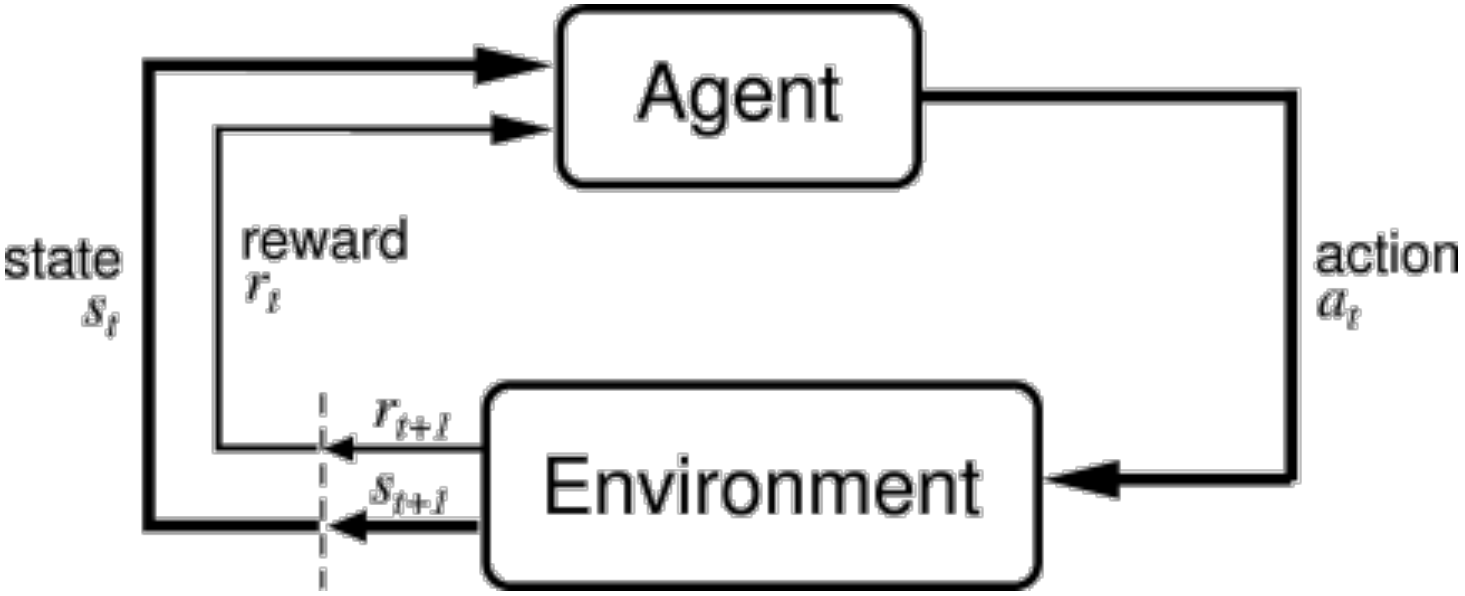
- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a precomputed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.
- Chess: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.
- Othello: human champions refuse to compete against computers, who are too good.
- Go: human champions refused to compete against computers, who were supposedly too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.



Speaking of Go, we saw already that Deep Networks (within Reinforcement Learning) may help. But what is Reinforcement Learning actually?

Reinforcement Learning

- Basic idea:
  - Receive feedback in the form of rewards
  - Agent's utility is defined by the reward function
  - Must (learn to) act so as to maximize expected rewards



Reinforcement Learning (RL)

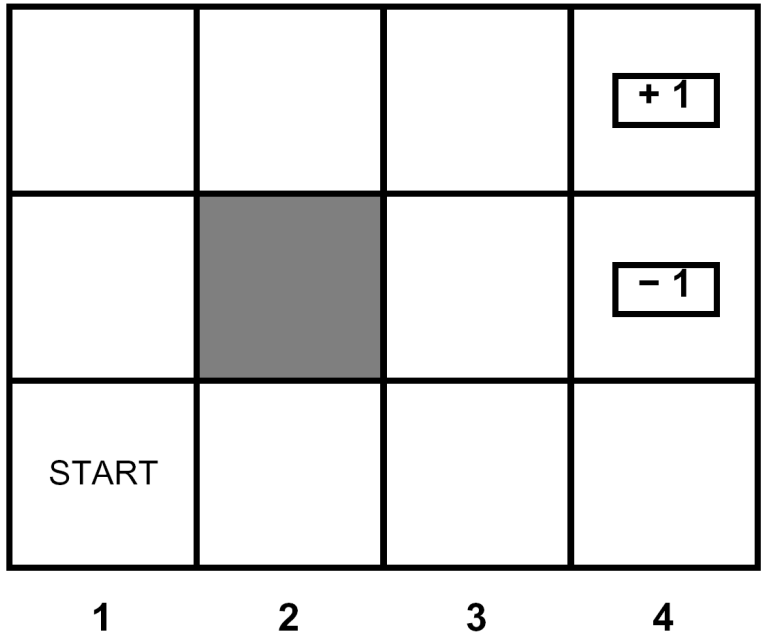
- RL algorithms attempt to find a policy (what action to do in which state) for maximizing cumulative reward for the agent over the course of the problem.
- Typically represented by a **Markov Decision Process**
- RL differs from supervised learning in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected.

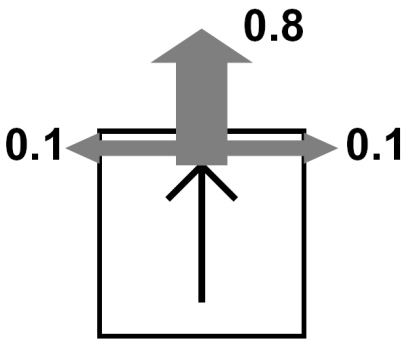
Credit Assignment Problem

- Given a sequence of states and actions, and the final sum of time-discounted future rewards, how do we infer which actions were effective at producing lots of reward and which actions were not effective?
- How do we assign credit for the observed rewards given a sequence of actions over time?
- Every reinforcement learning algorithm must address this problem

The Fruit Fly of RL: Grid World

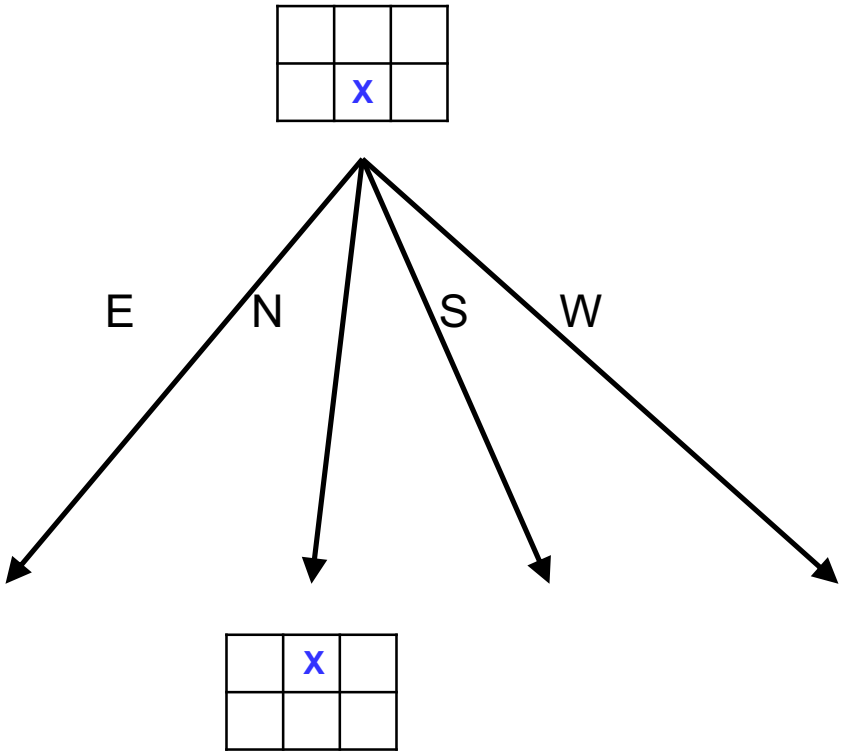
- The agent lives in a grid
- Walls block the agent's path
- The agent's actions do not always go as planned:
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put
- Small "living" reward each step
- Big rewards come at the end
- Goal: maximize sum of rewards*



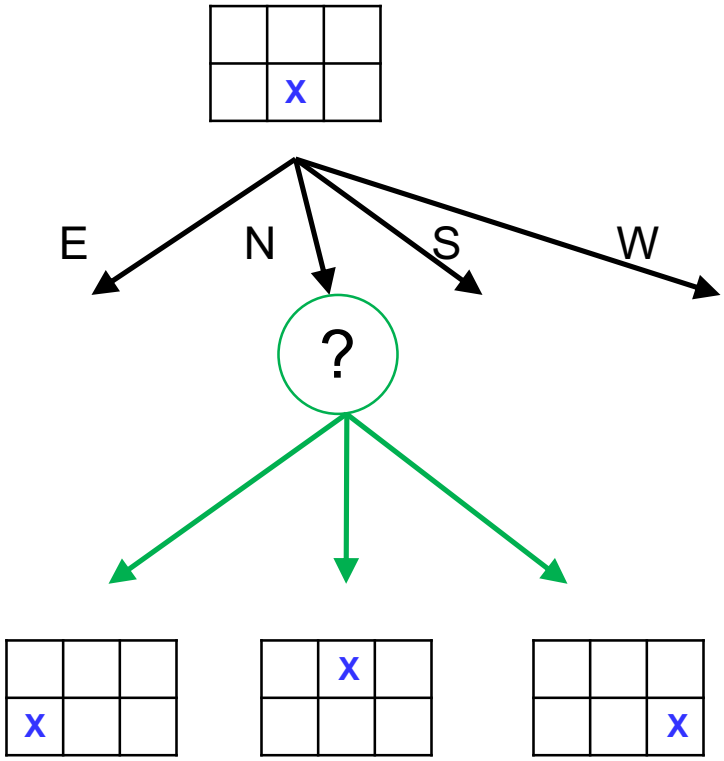


One can imaging two “Grid Futures”

Deterministic Grid World

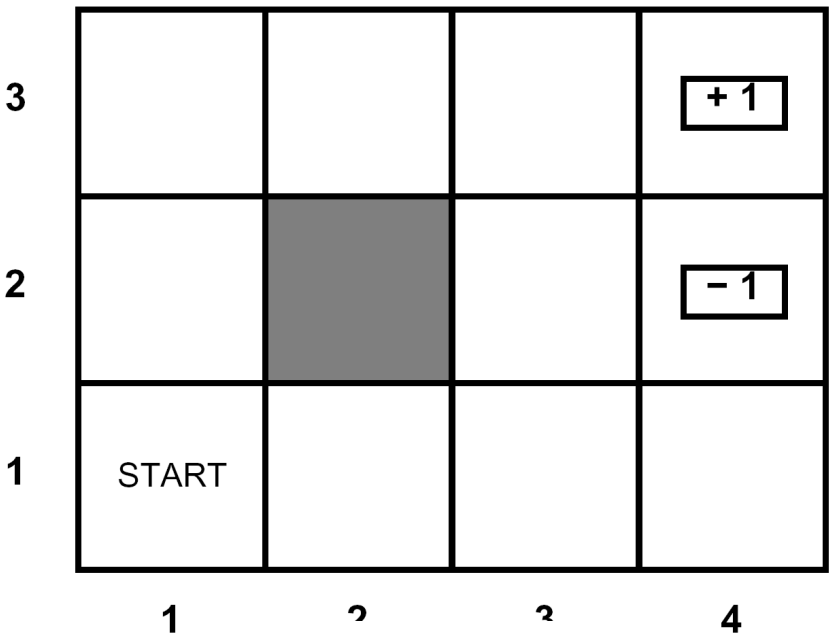


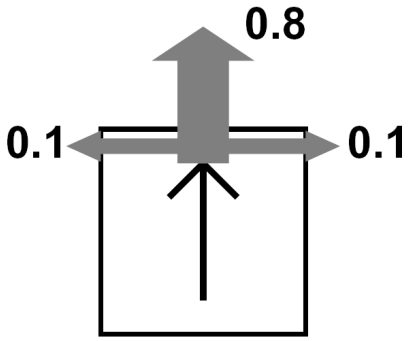
Stochastic Grid World



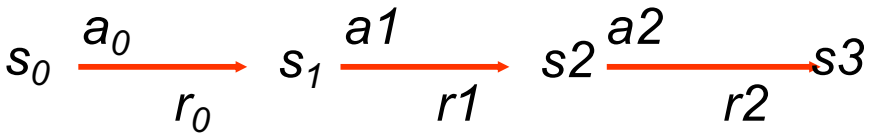
Now, what are Markov Decision Processes (MDP)?

- An MDP is defined by:
  - A set of states $s \in S$
  - A set of actions $a \in A$
  - A transition function T(s,a,s')
    - Prob that a from s leads to s'
    - i.e., P(s' | s,a)
    - Also called the model
  - A reward function R(s, a, s')
    - Sometimes just R(s) or R(s')
  - A start state (or distribution)
  - Maybe a terminal state
  - A discount factor: γ





- MDPs are a family of non-deterministic search problems
  - Reinforcement learning: MDPs where we don't know the transition or reward functions



What is Markov about MDPs?

- Andrey Markov (1856-1922)
- "Markov" generally means that given the present state, the future and the past are independent
- For Markov decision processes, "Markov" means:

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots S_0 = s_0)$$

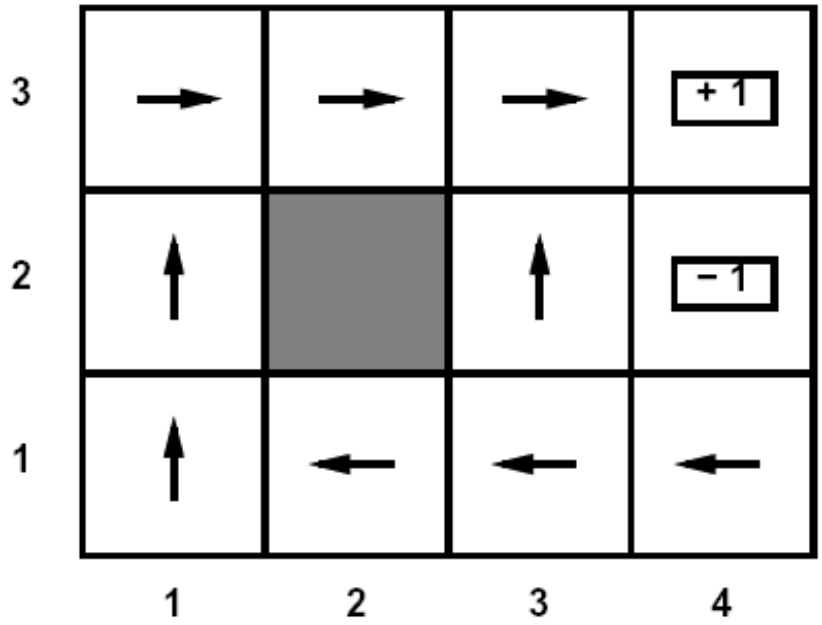
$$=$$

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

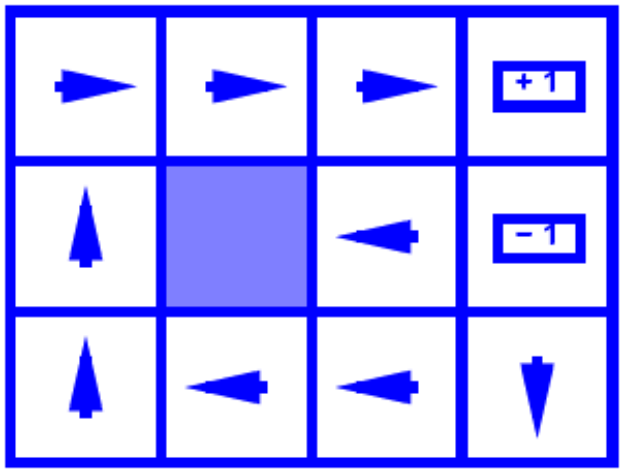
What does it mean to solve MDPs?

- In deterministic single-agent search problems, want an optimal plan, or sequence of actions, from start to a goal
- In an MDP, we want to compute an optimal policy $\pi^*: S \rightarrow A$
  - A policy π gives an action for each state
  - An optimal policy maximizes expected utility if followed
  - Defines a reflex agent

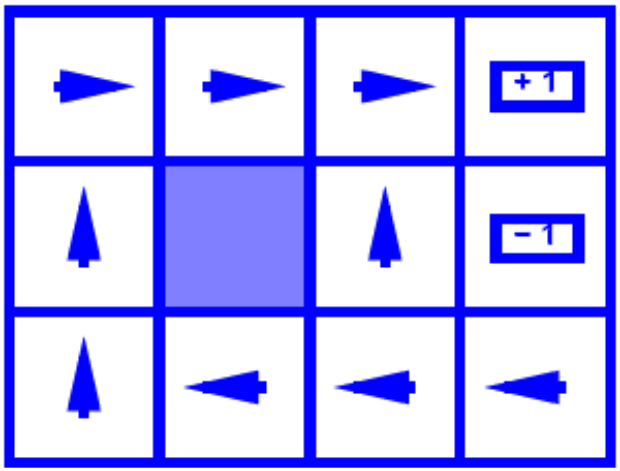
Optimal policy when R(s, a, s') = -0.03 for all non-terminals s



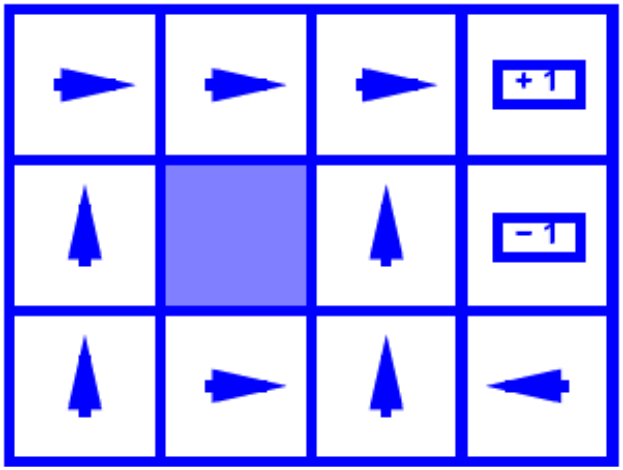
Depending on the rewards, optimal policies may look differently



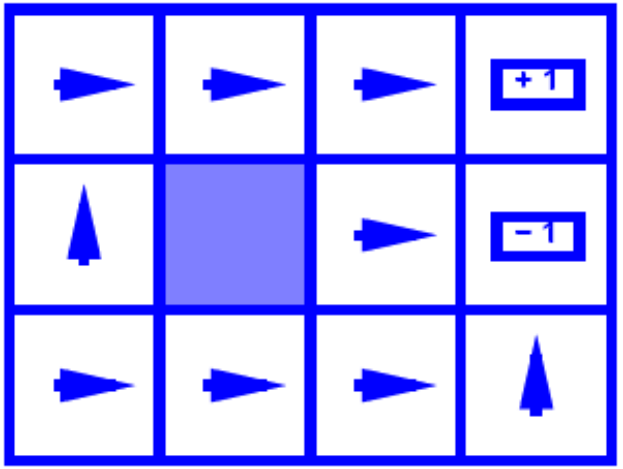
R(s) = -0.01



R(s) = -0.03



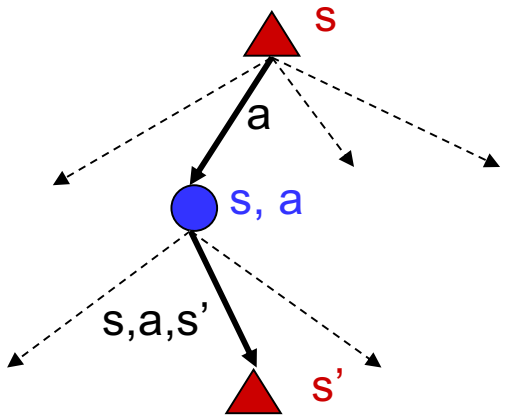
R(s) = -0.4



R(s) = -2.0

Recap: Defining MDPs

- Markov decision processes:
  - States S
  - Start state s_0
  - Actions A
  - Transitions P(s'|s,a) (or T(s,a,s'))
  - Rewards R(s,a,s') (and discount γ)



- MDP quantities so far:
  - Policy = Choice of action for each state
  - Utility (or return) = sum of discounted rewards

Intuition of Action Selection

- At time t, the agent tries to select an action to maximise the sum G_t of discounted rewards received in the future

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$$

Why Discounted Rewards

- **Rewards in the future are worth less than an immediate reward. This helps to ensure convergence of the epectations.**

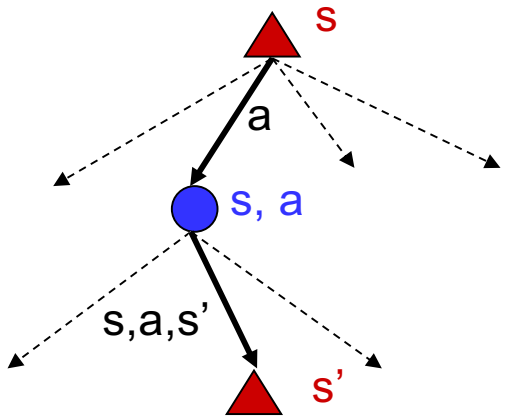
- Discount factor $\gamma \leq 1$ (often $\gamma = 0.9$)
- Assume reward n years in the future is only worth $(\gamma)^n$ of the value of immediate reward

  - $(0.9\text{^}6) * 10{,}000 = 0.531 * 10{,}000 = 5310$

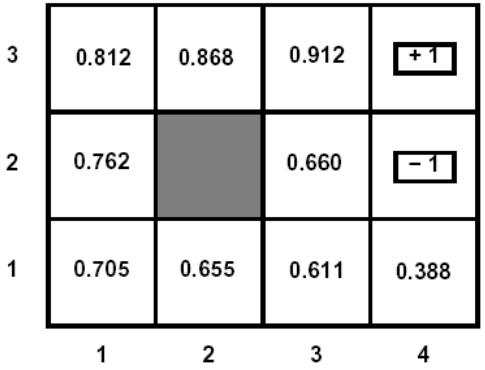
- For each state, calculate a *utility* value equal to the *Sum of Future Discounted Rewards*

Optimal Utilities

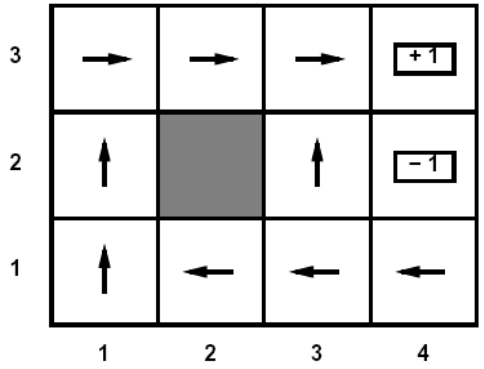
- Fundamental operation: compute the values (optimal expectimax utilities) of states s (based on the rewards)
- Why? Optimal values define optimal policies!



- Value Function defines the value of a state s: $V^*(s)$ = expected utility starting in s and thereafter acting optimally



- Action-Value Function defines the value of a q-state (s,a): $Q^*(s,a)$ = expected utility starting in s, taking action a and thereafter acting optimally



- Define the optimal policy:
  $\pi^*(s)$ = optimal action from state s

Optimality

- A policy is defined to be better than or equal to another policy if its expected return is greater than or equal to that of the other policy for all states
- There is always at least one optimal policy π_* that is better than or equal to all other policies
- All optimal policies share the same optimal state-value function v*, which gives the maximum expected return for any state s over all possible policies
- All optimal policies share the same optimal action-value function q*, which gives the maximum expected return for any state-action pair (s, a) over all possible policies

How can we compute, eg, the value function? Value Iteration

Idea:

- Start with $V_0^*(s) = 0$
- Given V_i^*, calculate the values for all states for depth i+1:

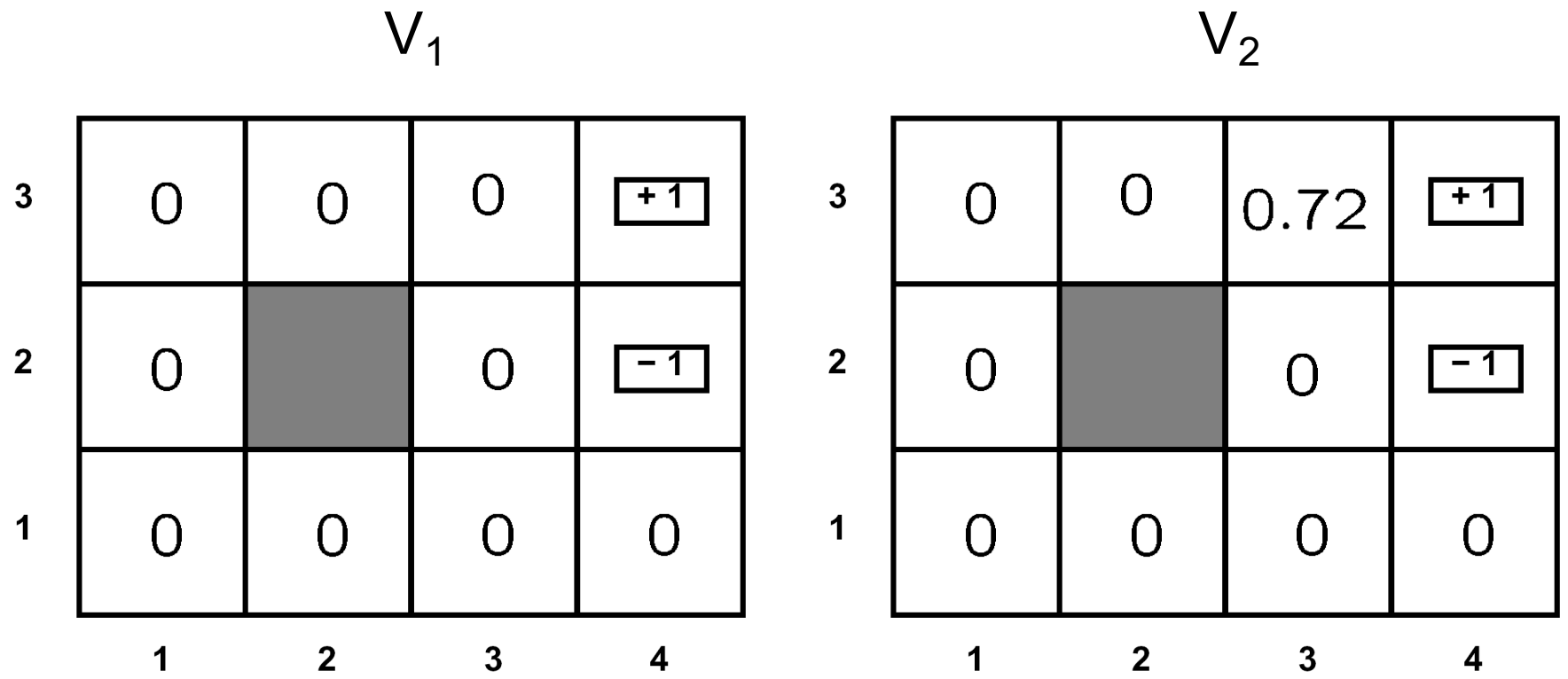
$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_i(s') \right]$$

- This is called a value update or Bellman update
- Repeat until convergence

Theorem: will converge to unique optimal values

- Basic idea: approximations get refined towards optimal values
- Policy may converge long before values do

Example: Value Iteration



- Information propagates outward from terminal states and eventually all states have correct value estimates

Example: Value Iteration

V_2

3	0	0	0.72	+1
2	0		0	−1
1	0	0	0	0
	1	2	3	4

V_3

3	0	0.52	0.78	+1
2	0		0.43	−1
1	0	0	0	0
	1	2	3	4

- Information propagates outward from terminal states and eventually all states have correct value estimates

OK, but what is now Reinforcement Learning?

Reinforcement learning still assume an MDP:

- A set of states $s \in S$
- A set of actions (per state) A
- A model T(s,a,s')
- A reward function R(s,a,s')
- A discount factor γ (could be 1)

- and we still seek for a policy $\pi(s)$

New twist: we don't know T or R

- i.e. don't know which states are good or what the actions do
- Must actually try actions and states out to learn

Model based v.s. Model free

- Model based approach RL:
  learn the model, and use it to derive the optimal policy.

- Model free approach RL:
  derive the optimal policy without learning the model.

On-policy versus Off-policy

- An **on-policy** agent learns only about the policy that it is executing.
- An **off-policy** agent learns about a policy or policies different from the one that it is executing

Passive learning v.s. Active learning

In passive learning

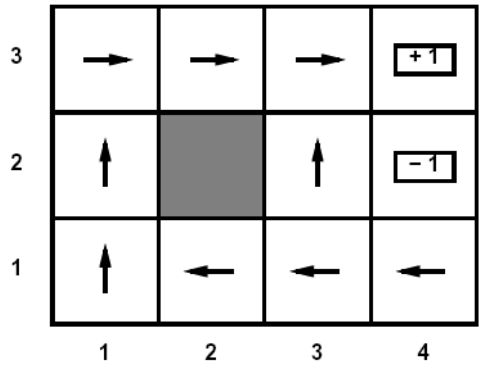
- The agent simply watches the world going by and tries to learn the utilities of being in various states. The goal is to execute a fixed policy (sequence of actions) and evaluate it

Active learning

- The agent not simply watches, but also acts. The goal is to act and learn an optimal policy.

Passive Learning

- Simplified task
  - You don't know the transitions T(s,a,s')
  - You don't know the rewards R(s,a,s')
  - You are given a policy $\pi(s)$
  - Goal: learn the state values of the policy



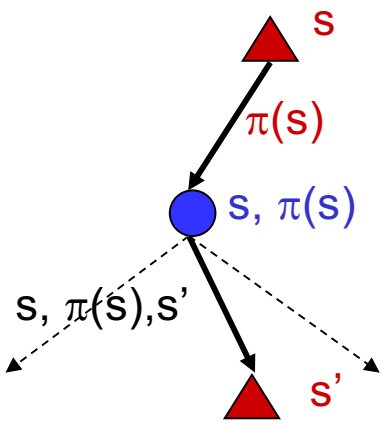
- In this case:
  - Learner "along for the ride"
  - No choice about what actions to take
  - Just execute the policy and learn from experience
  - We'll get to the active case soon
  - This is NOT offline planning! You actually take actions in the world and see what happens…

Model-Based Learning

- Idea:
  - Learn the model empirically through experience
  - Solve for values as if the learned model were correct

- Simple empirical model learning
  - Count outcomes for each s,a
  - Normalize to give estimate of **T(s,a,s')**
  - Discover **R(s,a,s')** when we experience (s,a,s')

- Solving the MDP with the learned model
  - Iterative policy evaluation, for example

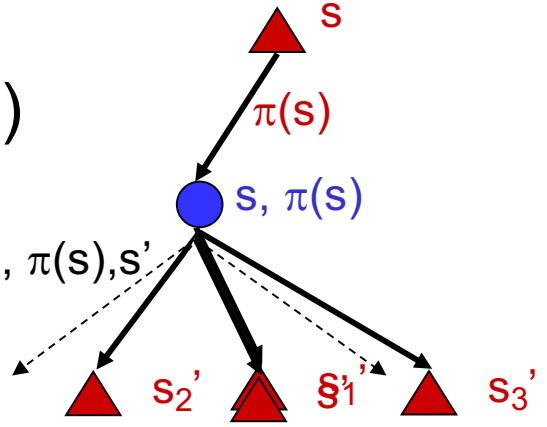


$$V^{\pi}_{i+1}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}_{i}(s')]$$

Sample-Based Policy Evaluation?

$$V^{\pi}_{i+1}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}_{i}(s')]$$

- Who needs T and R? Approximate the expectation with samples (drawn from T!)



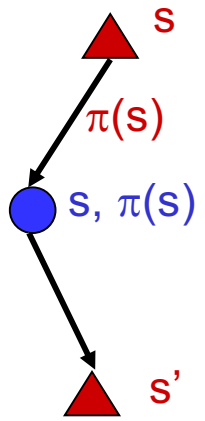
$$sample_1 = R(s, \pi(s), s'_1) + \gamma V^{\pi}_{i}(s'_1)$$
$$sample_2 = R(s, \pi(s), s'_2) + \gamma V^{\pi}_{i}(s'_2)$$
$$\ldots$$
$$sample_k = R(s, \pi(s), s'_k) + \gamma V^{\pi}_{i}(s'_k)$$

$$V^{\pi}_{i+1}(s) \leftarrow \frac{1}{k} \sum_{i} sample_i$$

Almost! But we only actually make progress when we move to i+1.

Temporal-Difference Learning

- Big idea: learn from every experience!
  - Update V(s) each time we experience (s,a,s',r)
  - Likely s' will contribute updates more often
- Temporal difference learning
  - Policy still fixed!
  - Move values toward value of whatever successor occurs: running average!



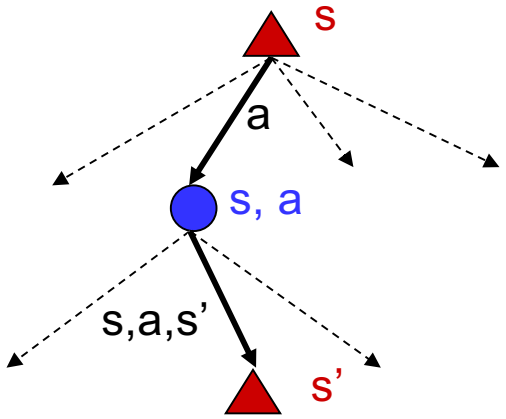
Sample of V(s): $sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$

Update to V(s): $V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha)sample$

Same update: $V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$

Problems with TD Value Learning

- TD value leaning is a model-free way to do policy evaluation
- However, if we want to turn values into a (new) policy, we're sunk:



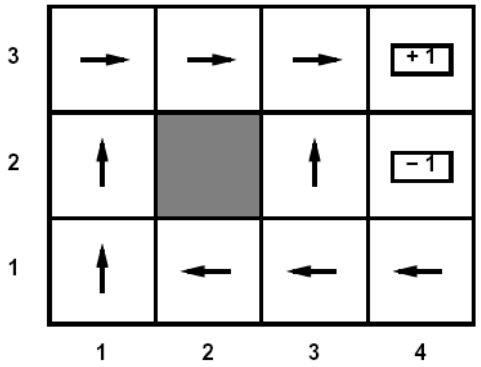
$$\pi(s) = \arg\max_a Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

- Idea: learn Q-values directly
- Makes action selection model-free too!

Active Learning

- Full reinforcement learning
  - You don’t know the transitions T(s,a,s’)
  - You don’t know the rewards R(s,a,s’)
  - You can choose any actions you like
  - Goal: learn the optimal policy



- In this case:
  - Learner makes choices!
  - Fundamental tradeoff: exploration vs. exploitation
  - This is NOT offline planning! You actually take actions in the world and find out what happens…

Q-Learning

- Q-Learning: sample-based Q-value iteration
- Learn Q*(s,a) values
  - Receive a sample (s,a,s',r)
  - Consider your old estimate: $Q(s,a)$
  - Consider your new sample estimate:

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right]$$

$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

  - Incorporate the new estimate into a running average:

$$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + (\alpha) \left[sample \right]$$

Q-Learning Properties

- Amazing result: Q-learning converges to optimal policy
  - If you explore enough
  - If you make the learning rate small enough
  - … but not decrease it too quickly!
  - Basically doesn't matter how you select actions (!)

Exploration / Exploitation

- Several schemes for forcing exploration
  - Simplest: random actions (**ε greedy**)
    - Every time step, flip a coin
    - With probability ε, act randomly
    - With probability 1-ε, act according to current policy
  - Problems with random actions?
    - You do explore the space, but keep thrashing around once learning is done
    - One solution: lower ε over time
    - Another solution: exploration functions

The Story So Far: MDPs and RL

Things we know how to do:

- If we know the MDP
  - Compute V*, Q*, π* exactly
  - Evaluate a fixed policy π
- If we don't know the MDP
  - We can estimate the MDP then solve
  - We can estimate V for a fixed policy π
  - We can estimate Q*(s,a) for the optimal policy while executing an exploration policy

Techniques:

- Model-based DPs
  - Value and policy Iteration
  - Policy evaluation
- Model-based RL
- Model-free RL:
  - Value learning
  - Q-learning

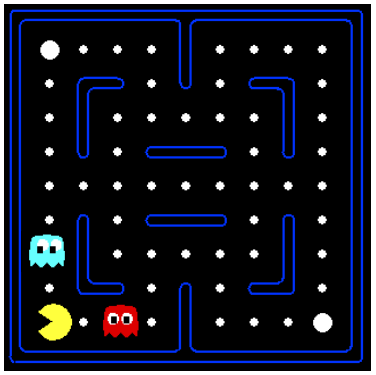
Q-Learning in realistic domains

- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - Too many states to hold the q-tables in memory

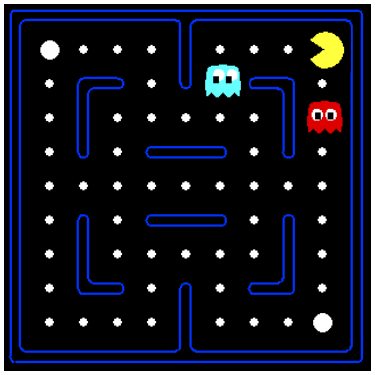
- Instead, we want to generalize:
  - Learn about some small number of training states from experience
  - Generalize that experience to new, similar states
  - This is a fundamental idea in machine learning, and we'll see it over and over again

Example: Pacman

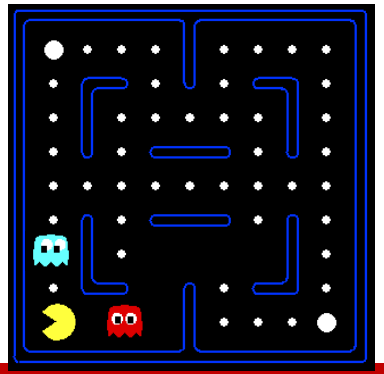
- Let's say we discover through experience that this state is bad:



- In naïve q learning, we know nothing about this state or its q states:

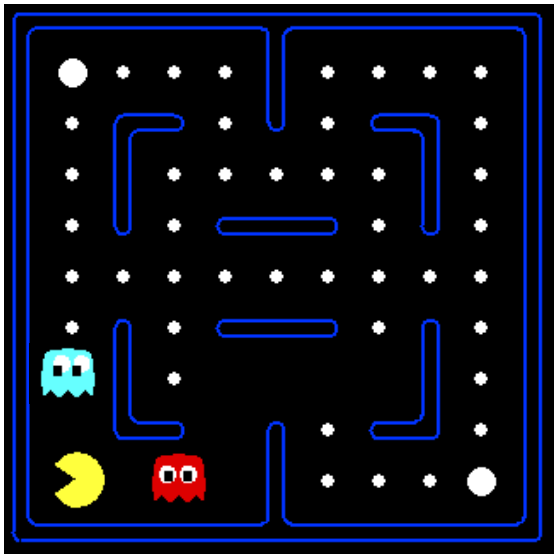


- Or even this one!



Feature-Based Representations

- Solution: describe a state using a vector of features
  - Features are functions from states to real numbers (often 0/1) that capture important properties of the state
  - Example features:
    - Distance to closest ghost
    - Distance to closest dot
    - Number of ghosts
    - 1 / (dist to dot)^2
    - Is Pacman in a tunnel? (0/1)
    - …… etc.
  - Can also describe a q-state (s, a) with features (e.g. action moves closer to food)



Linear Feature Functions

- Using a feature representation, we can write a q function (or value function) for any state using a few weights:

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

- Advantage: our experience is summed up in a few powerful numbers
- Disadvantage: states may share features but be very different in value!

Function Approximation

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

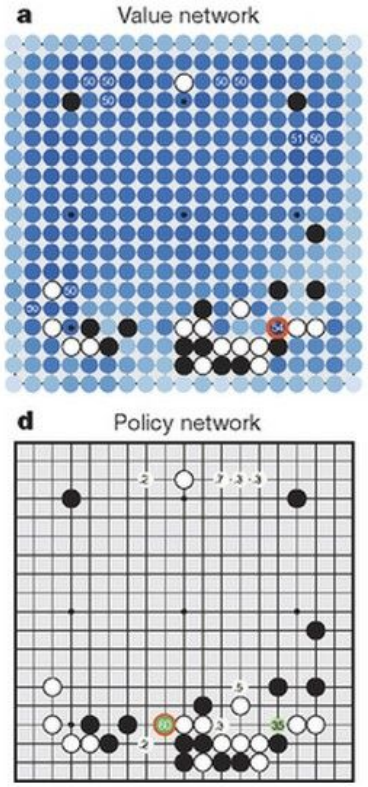
- Q-learning with linear q-functions:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\,[error]$$

$$w_i \leftarrow w_i + \alpha\,[error]\,f_i(s,a)$$

- Intuitive interpretation:
  - Adjust weights of active features
  - E.g. if something unexpectedly bad happens, disprefer all states with that state's features
- Formal justification: online least squares

Or one may go for deep neural networks



Deep policy network is trained to produce probability map of promising moves. The deep value network is used to prune the (mcmc) search tree

Deep Q-Networks (DQN)

- Deep Q-Networks is Q-learning with a deep neural network function approximator called the Q-network
- Discrete and finite set of actions A
- Example: Breakout has 3 actions – move left, move right, no movement
- Uses epsilon-greedy policy to select actions

Policy Search

http://heli.stanford.edu/

Policy Search

- Problem: often the feature-based policies that work well aren't the ones that approximate V / Q best
- Solution: learn the policy that maximizes rewards rather than the value that predicts rewards
- This is the idea behind policy search, such as what controlled the upside-down helicopter.
- We may parameterize the policy and then run some gradient optimization, e.g., REINFORCE

$$\pi(a|s,\boldsymbol{\theta}) \doteq \frac{\exp(h(s,a,\boldsymbol{\theta}))}{\sum_b \exp(h(s,b,\boldsymbol{\theta}))} \qquad \boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\gamma^t G_t \frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}.$$

Policy Search

- Simplest policy search:
  - Start with an initial linear value function or q-function
  - Nudge each feature weight up and down and see if your policy is better than before
- Problems:
  - How do we tell the policy got better?
  - Need to run many sample episodes!
  - If there are a lot of features, this can be impractical