Q-Learning

- Reinforcement Learning
- Basic Q-learning algorithm
- Common modifications

Reinforcement Learning

- Delayed reward
  - We don't immediately know whether we did the correct thing
- Encourages exploration
- We don't necessarily know the precise results of our actions before we do them
- We don't necessarily know all about the current state
- Life-long learning

Our Problem

- We don't immediately know how beneficial our last move was
- Rewards: 100 for win, -100 for loss
- We don't know what the new state will be from an action
- Current state is well defined
- Life-long learning?

Q-Learning Basics

- At each step s, choose the action a which maximizes the function Q(s, a)
  - Q is the estimated utility function – it tells us how good an action is given a certain state
- Q(s, a) = immediate reward for making an action + best utility (Q) for the resulting state
- Note: recursive definition
- More formally ...

Formal Definition

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

$r(s,a) =$ Immediate reward

$\gamma =$ relative value of delayed vs. immediate rewards (0 to 1)

$s' =$ the new state after action a

a, a': actions in states s and s', respectively

Selected action:

$$\pi(s) = argmax_a Q(s,a)$$

Q Learning Algorithm

For each state-action pair (s, a), initialize the table entry $\hat{Q}(s,a)$ to zero

Observe the current state s

Do forever:

---Select an action a and execute it

---Receive immediate reward r

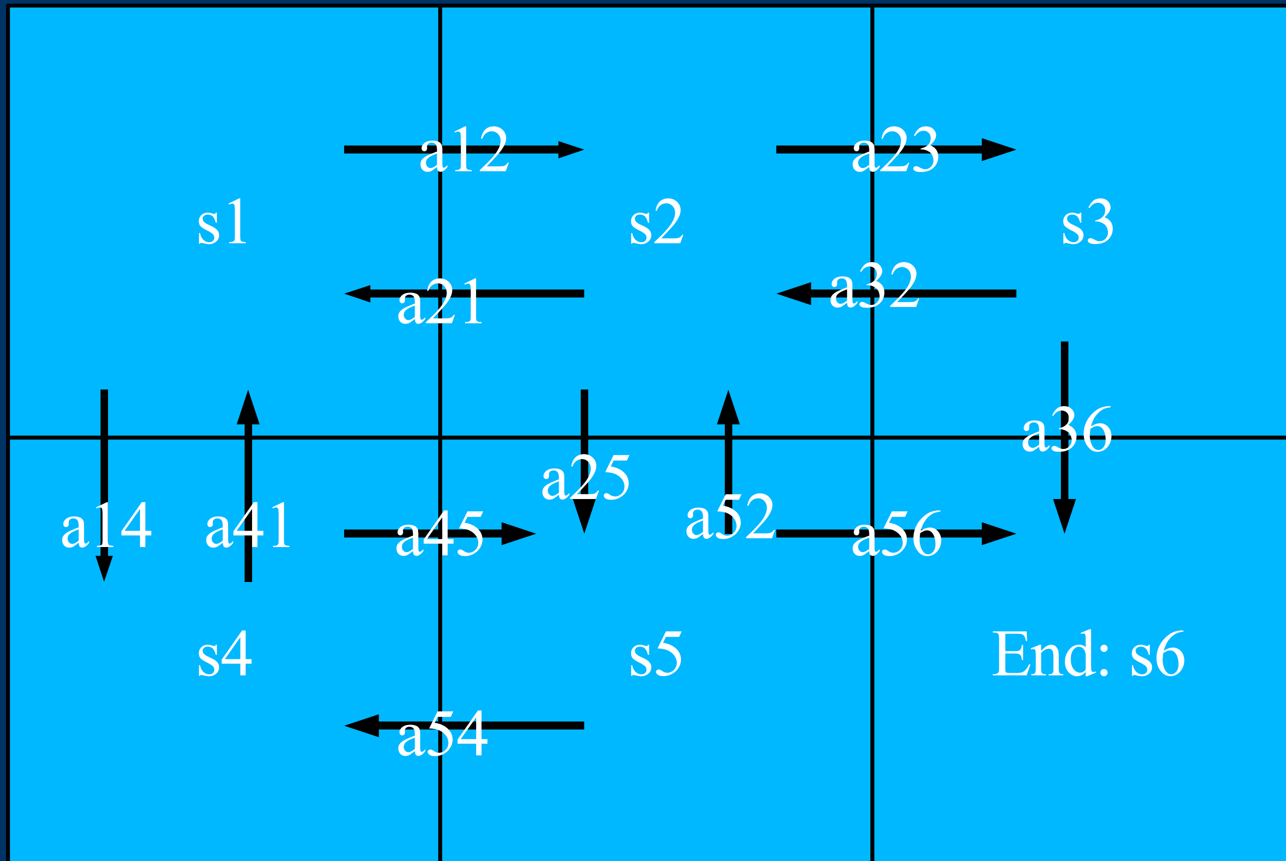
---Observe the new state s'

---Update the table entry for $\hat{Q}(s,a)$ as follows:

$$\hat{Q}(s,a) = r + \gamma \, max_{a}' \hat{Q}(s',a')$$

--- $s = s'$

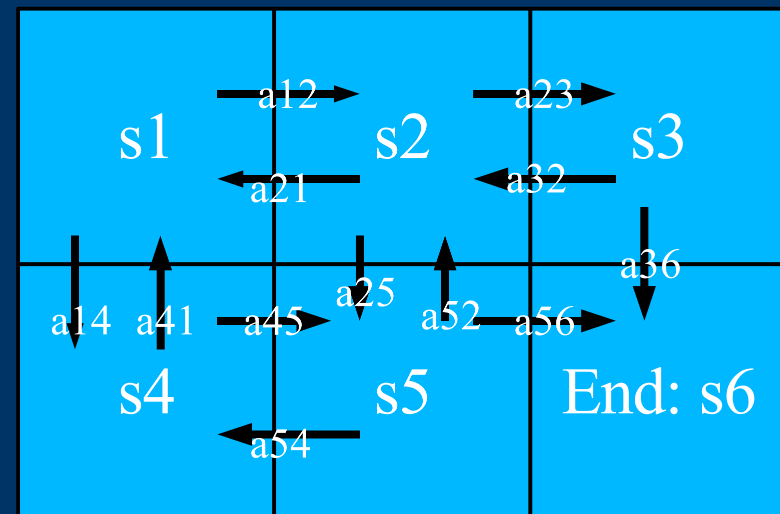
Example Problem



$\gamma = .5$, r = 100 if moving into state s6, 0 otherwise

Initial State

s1, a12	0
s1, a14	0
s2, a21	0
s2, a23	0
s2, a25	0
s3, a32	0
s3, a36	0
s4, a41	0
s4, a45	0
s5, a54	0
s5, a52	0
s5, a56	0

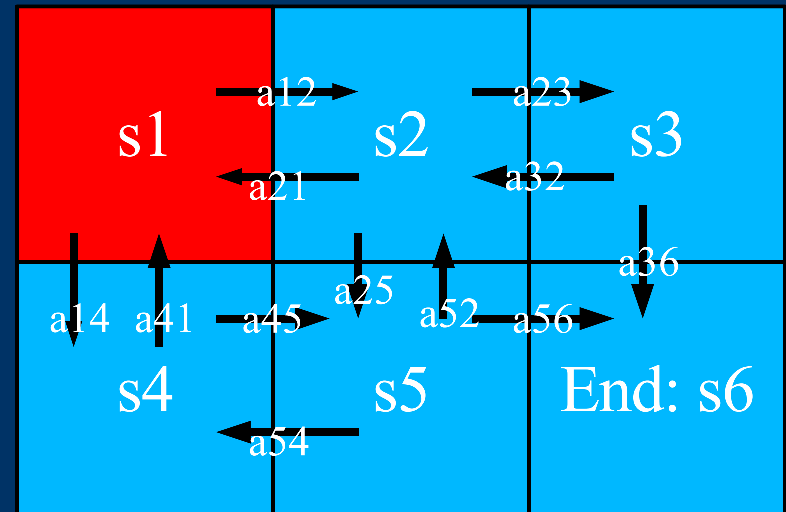


The Algorithm

s1, a12	0
s1, a14	0
s2, a21	0
s2, a23	0
s2, a25	0
s3, a32	0
s3, a36	0
s4, a41	0
s4, a45	0
s5, a54	0
s5, a52	0

Current Position: Red

Available actions: a12, a14
Chose a12



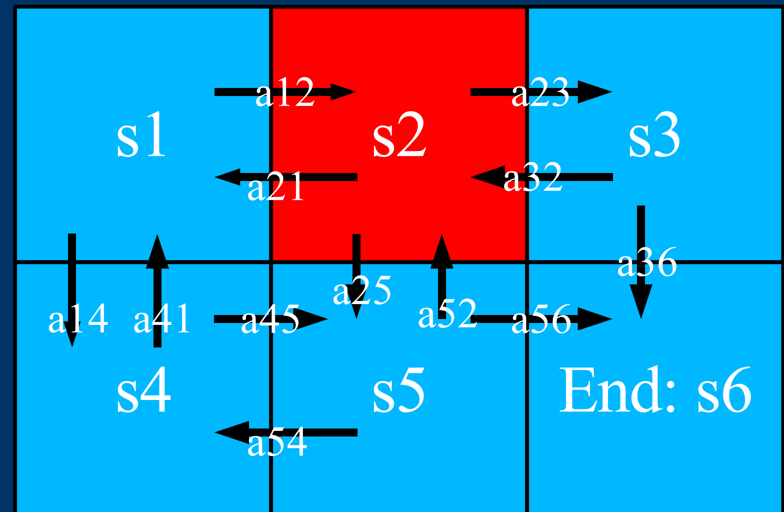
Update Q(s1, a12)

s1, a12	0
s1, a14	0
s2, a21	0
s2, a23	0
s2, a25	0
s3, a32	0
s3, a36	0
s4, a41	0
s4, a45	0
s5, a54	0
s5, a52	0

Current Position: Red

Available actions: a21, a25, a23

Update Q(s1, a12):
Q(s1, a12) = r + .5 * max(Q(s2,a21), Q(s2,a25), Q(s2,a23))
= 0

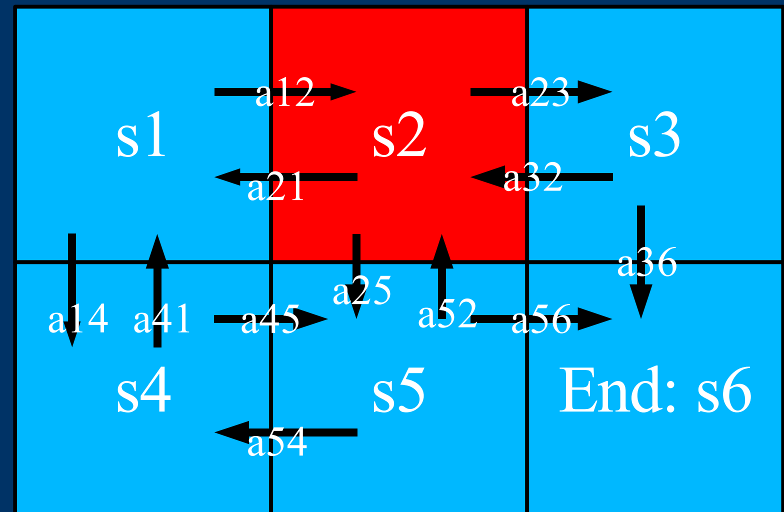


Next Move

s1, a12	0
s1, a14	0
s2, a21	0
s2, a23	0
s2, a25	0
s3, a32	0
s3, a36	0
s4, a41	0
s4, a45	0
s5, a54	0
s5, a52	0

Current Position: Red

Available actions: a21, a25, a23
Chose a23



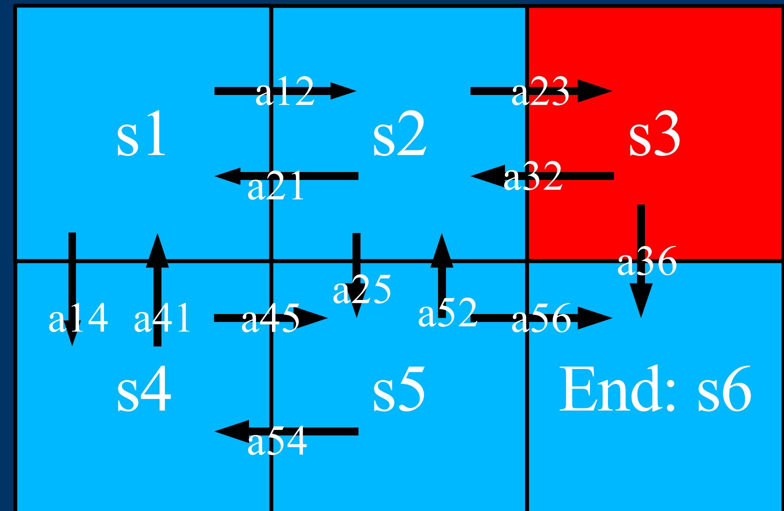
Update Q(s2, a23)

s1, a12	0
s1, a14	0
s2, a21	0
s2, a23	0
s2, a25	0
s3, a32	0
s3, a36	0
s4, a41	0
s4, a45	0
s5, a54	0
s5, a52	0

Current Position: Red

Available actions: a32, a36

Update Q(s1, a12):
Q(s2, a23) = r + .5 * max(Q(s3,a32), Q(s3,a36))
= 0

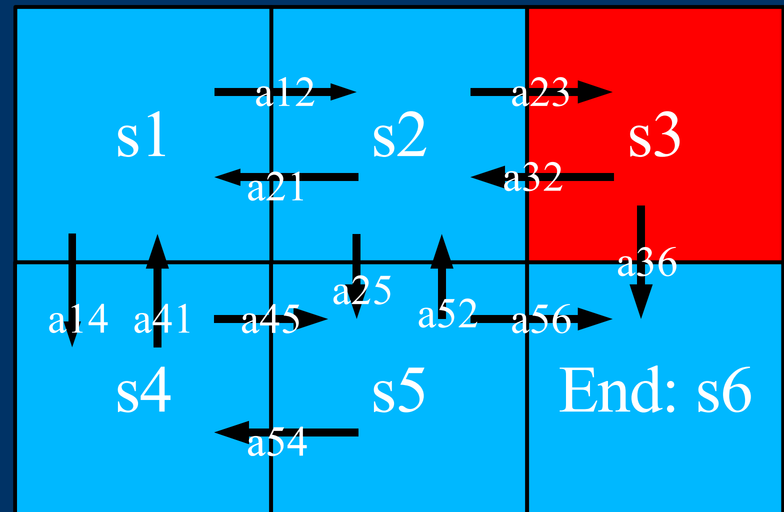


Next Move

s1, a12	0
s1, a14	0
s2, a21	0
s2, a23	0
s2, a25	0
s3, a32	0
s3, a36	0
s4, a41	0
s4, a45	0
s5, a54	0
s5, a52	0

Current Position: Red

Available actions: a32, a36
Chose a36



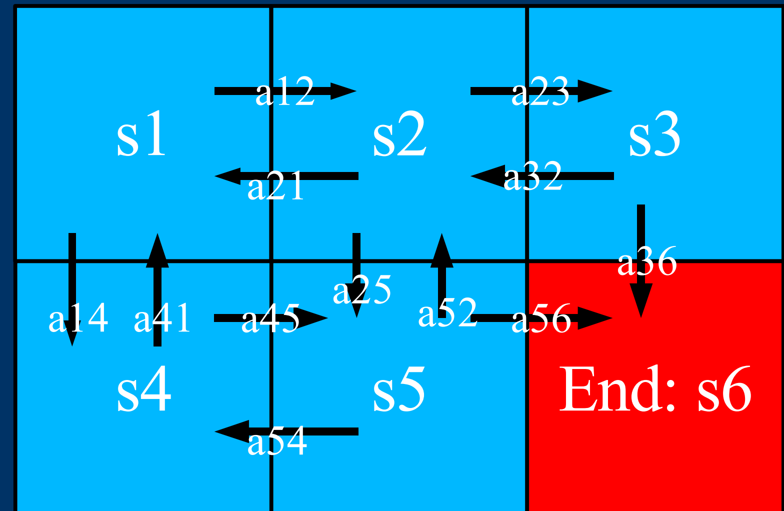
Update Q(s3,a36)

s1, a12	0
s1, a14	0
s2, a21	0
s2, a23	0
s2, a25	0
s3, a32	0
s3, a36	100
s4, a41	0
s4, a45	0
s5, a54	0
s5, a52	0

Current Position: Red

FINAL STATE!

Update Q(s1, a12):
Q(s2, a23) = r = 100

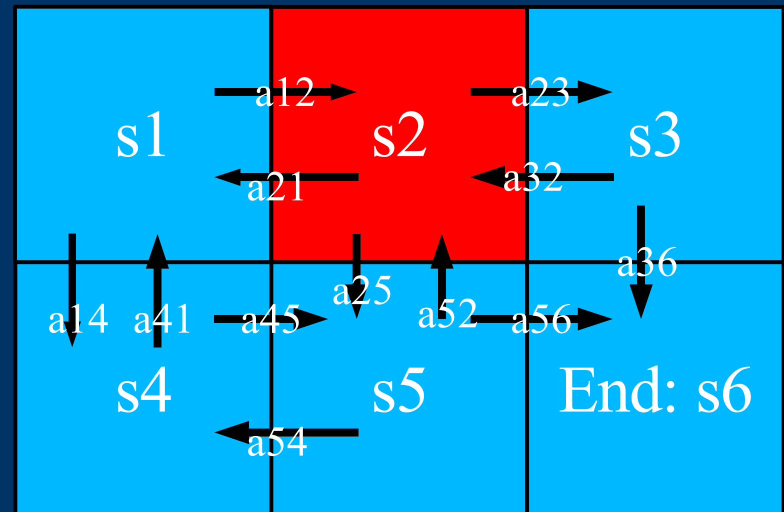


New Game

s1, a12	0
s1, a14	0
s2, a21	0
s2, a23	0
s2, a25	0
s3, a32	0
s3, a36	100
s4, a41	0
s4, a45	0
s5, a54	0
s5, a52	0

Current Position: Red

Available actions: a21, a25, a23
Chose a23



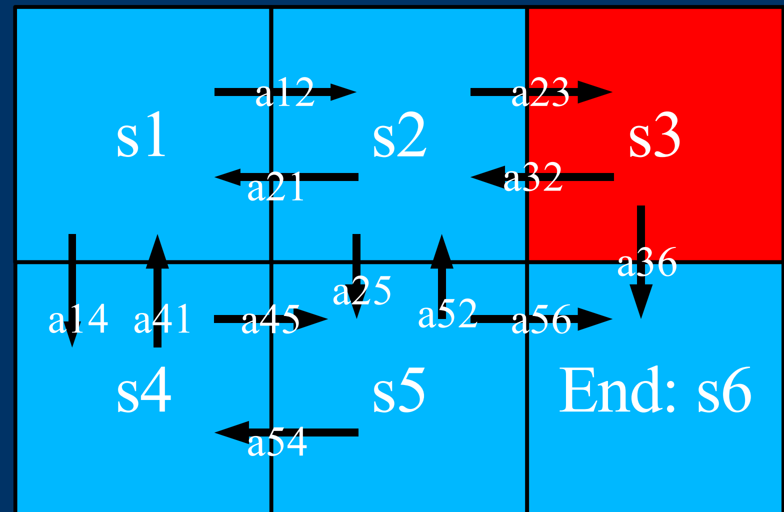
Update Q(s2, a23)

s1, a12	0
s1, a14	0
s2, a21	0
s2, a23	50
s2, a25	0
s3, a32	0
s3, a36	100
s4, a41	0
s4, a45	0
s5, a54	0
s5, a52	0

Current Position: Red

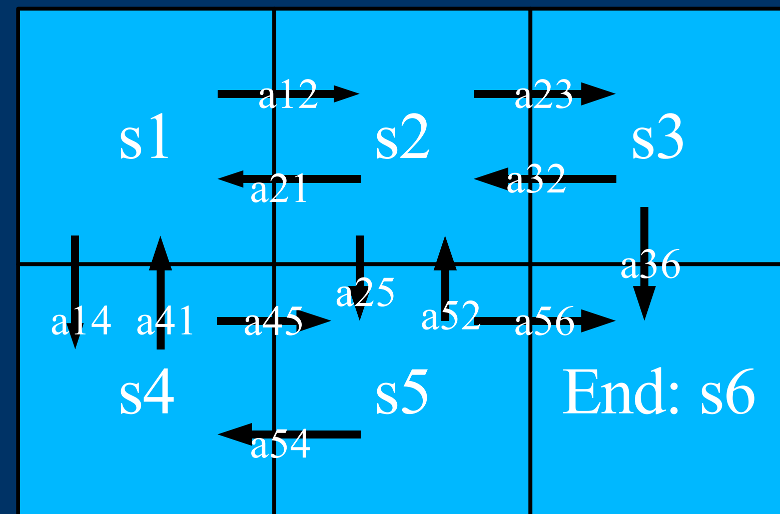
Available actions: a32, a36

Update Q(s1, a12):
Q(s2, a23) = r + .5 * max(Q(s3,a32), Q(s3,a36))
= 0 + .5 * 100 = 50



Final State (after many iterations)

s1, a12	25
s1, a14	25
s2, a21	12.5
s2, a23	50
s2, a25	25
s3, a32	25
s3, a36	100
s4, a41	12.5
s4, a45	50
s5, a54	25
s5, a52	25
s5, a56	100



Properties

- Convergence: Our approximation will converge to the true Q function
  - But we must visit every state-action pair infinitely many times!
- Table size can be very large for complex environments like a game
- We do not estimate unseen values
- How to we fix these problems?

Neural Network Approximation

- Instead of the table, use a neural network
  - Inputs are the state and action
  - Output is a number between 0 and 1 that represents the utility
- Encoding the states and actions *properly* will be challenging
  - Helpful idea: multiple neural networks, one for each action

Enhancements

- Exploration strategy
- Store past state-action transitions and retrain on them periodically
  - The values may change as time progresses
- Temporal Difference Learning

Exploration Strategy

- Want to focus exploration on the good states
- Want to explore all states
- Solution: Randomly choose the next action
  - Give a higher probability to the actions that currently have better utility

$$P(a_i|s)=\frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}}$$

Temporal Difference (TD) Learning

- Look farther into the future of a move
- Update the Q function after looking farther ahead
- Speeds up the learning process
- We will discuss this more when the time comes