Reinforcement Learning

Reading:
M. E. Harmon and S. S. Harmon,
"Reinforcement Learning:
A Tutorial"

(linked from class webpage)

Reinforcement Learning Overview

- We've focused a lot on supervised learning:
  - Training examples: $(\mathbf{x}_1, y_1)$, $(\mathbf{x}_2, y_2)$,...

- But consider a different type of learning problem, in which a robot has to learn to do tasks in a particular environment.
  - E.g.,
    - Navigate without crashing into anything
    - Locate and retrieve an object
    - Perform some multi-step manipulation of objects resulting in a desired configuration (e.g., sorting objects)

- This type of problem doesn’t typically provide clear “training examples” with detailed feedback at each step.

- Rather, robot gets intermittent “rewards” for doing “the right thing”, or “punishment” for doing “the wrong thing”.

- Goal: To have robot (or other learning system) learn, based on such intermittent rewards, what action to take in a given situation.

- Ideas for this have been inspired by “reinforcement learning” experiments in psychology literature.

Exploitation vs. Exploration

- *On-line versus off-line learning*

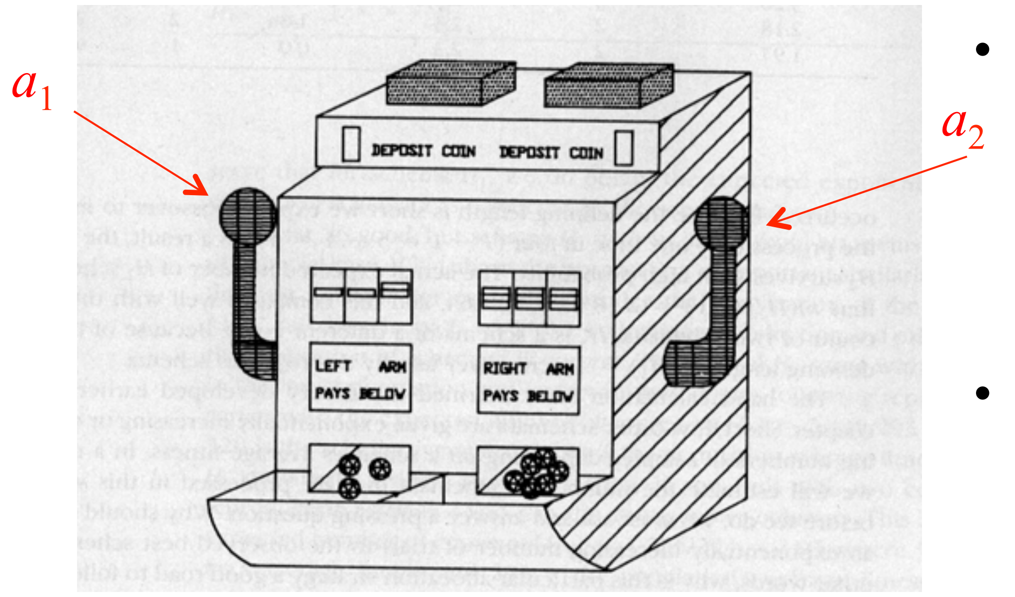
- On-line learning requires the correct balance between “exploitation” and “exploration”

- **Exploitation**
  - Exploit current good strategies to obtain known reward

- **Exploration**
  - Explore new strategies in hope of finding ways to increase reward

Exploitation vs. exploration in genetic algorithms?

Two-armed bandit model for exploitation and exploration with non-deterministic rewards



(From D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley 1989.)

- You are given n quarters to play with, and don't know the probabilities of payoffs of the respective arms, a_1 and a_2.

- What is the optimal way to allocate your quarters between the two arms so as to maximize your earnings (or minimize your losses) over the n arm-pulls ?

- Each arm roughly corresponds with a possible “strategy” to test.

- The question is, how should you allocate the next sample between the two different arms, given what rewards you have obtained so far?

- Let $Q(a)$ = *expected reward of arm a.*

- Here is one algorithm for estimating $Q(a)$:

$Q_0(a_i) = 0$ for a_1, a_2

Repeat for $t = 1$ to n:

    Choose arm a' if $Q(a') = \max_a Q(a)$ (choose random if tie)

$$Q_t(a') \leftarrow Q_{t-1}(a') + \eta\left[r_t(a') - Q_{t-1}(a')\right]$$

where r_t is the reward obtained by pulling arm a^* at time t, and η is a learning rate parameter.

- Can generalize to “*k*-armed bandit”, where each arm is a possible “strategy”.

- This model was used both for GAs and for reinforcement learning.

Applications of reinforcement learning: A few examples

- Learning to play backgammon (and more recently, Go)
- Robot arm control (juggling)
- Robo-soccer
- Robot navigation
- Elevator dispatching
- Power systems stability control
- Job-shop scheduling
- Air traffic control

Robby the Robot can learn via reinforcement learning

Sensors:

- N,S,E,W,C(urrent)

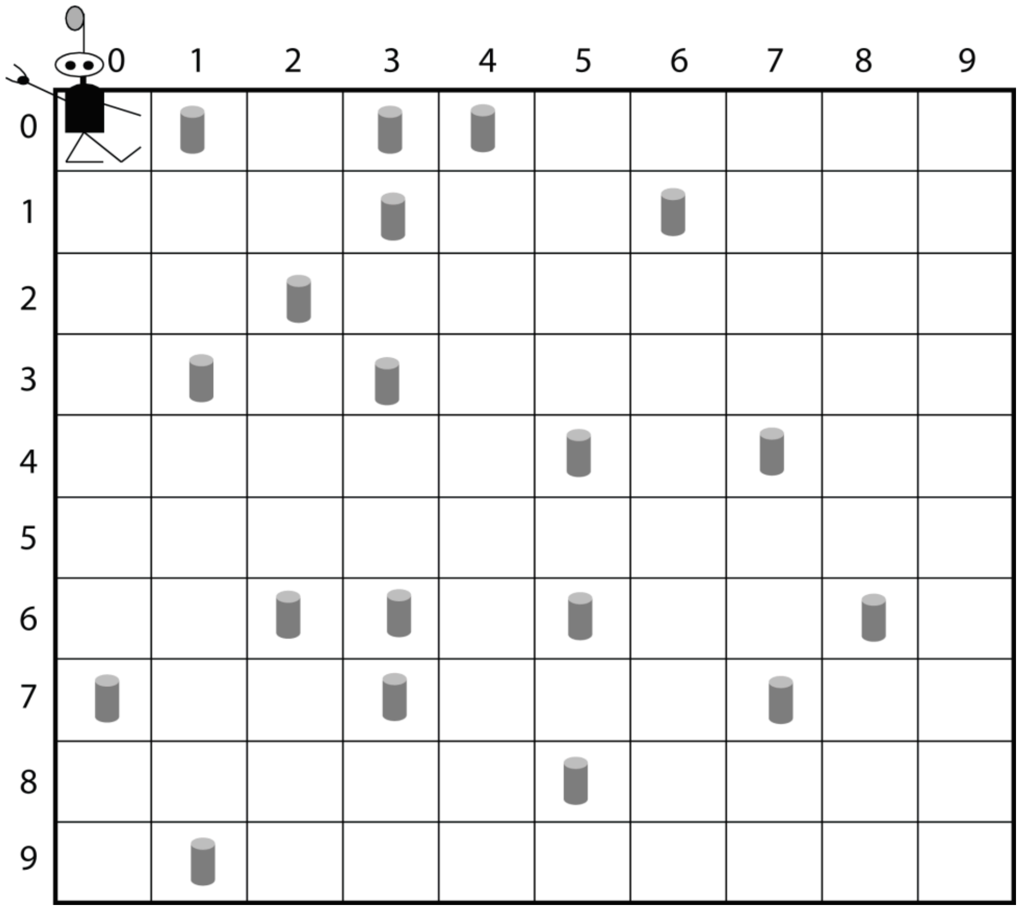
Actions:

- Move N
- Move S
- Move E
- Move W
- Move random
- Stay put
- Try to pick up can

Rewards/Penalties (points):

- Picks up can: 10
- Tries to pick up can on empty site: -1
- Crashes into wall: -5

"policy" = "strategy"



Formalization

Reinforcement learning is typically formalized as a **Markov decision process** (MDP):

Agent *L only knows current state and actions available from that* state.

Agent *L* can:

- perceive a set *S* of distinct states of an environment
- perform a set *A* of actions.

Components of Markov Decision Process (MDP):

- Possible set *S* of states (state space)
- Possible set *A* of actions (action space)
- State transition function (unknown to learner L)

$$\delta : S \times A \longrightarrow S \qquad \delta(s_t, a_t) = s_{t+1}$$

- Reward function (unknown to learner L)

$$r : S \times A \longrightarrow \Re \qquad r(s_t, a_t) = r_t$$

Formalization

Reinforcement learning is typically formalized as a **Markov decision process** (MDP):

Agent *L only knows current state and actions available from that state.*

Agent *L* can:

- perceive a set *S* of distinct states of an environment
- perform a set *A* of actions.

Components of Markov Decision Process (MDP):

- Possible set *S* of states (state space)
- Possible set *A* of actions (action space)
- State transition function (unknown to learner L)

$$\delta : S \times A \longrightarrow S \qquad \delta(s_t, a_t) = s_{t+1}$$

- Reward function (unknown to learner L)

$$r : S \times A \longrightarrow \Re \qquad r(s_t, a_t) = r_t$$

Note: Both δ and r can be deterministic or probabilistic.

In the Robby the Robot example, they are both deterministic

Goal is for agent L to learn policy π, $\pi : S \longrightarrow A$, such that π maximizes cumulative reward

Example: Cumulative value of a policy

Sensors:

- N,S,E,W,C(urrent)

Actions:

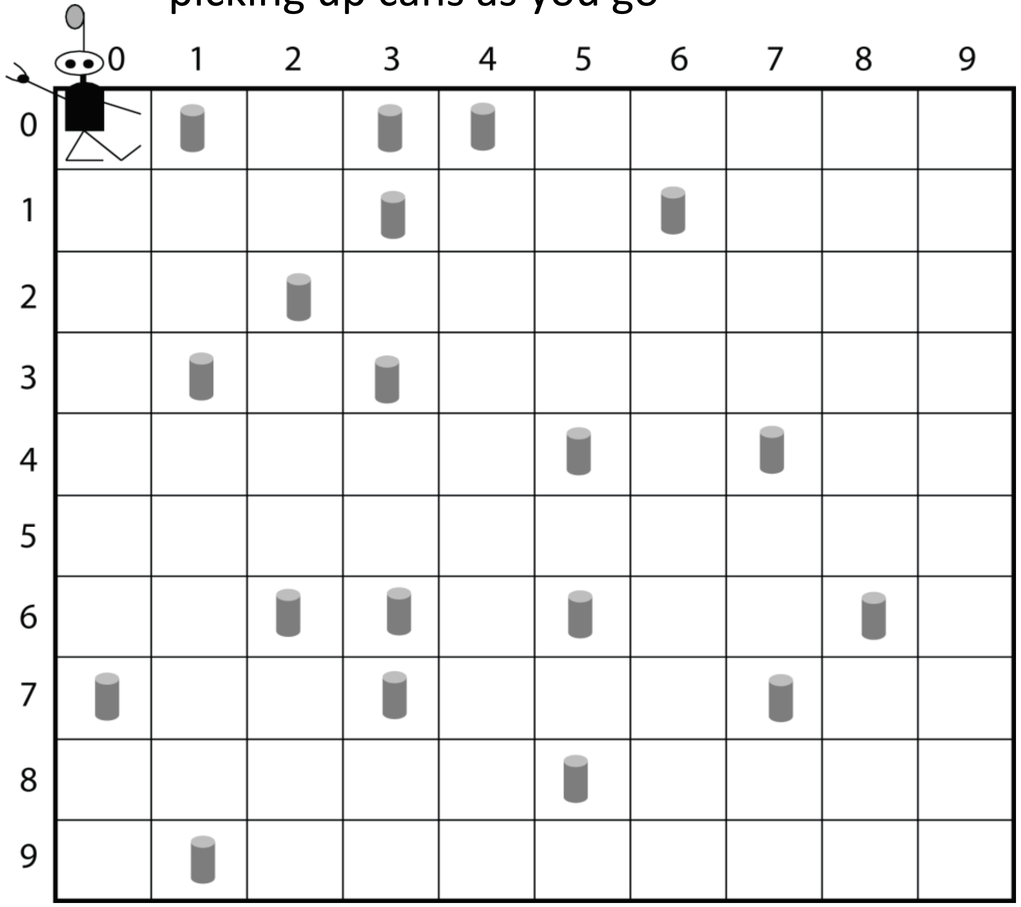
- Move N
- Move S
- Move E
- Move W
- Move random
- Stay put
- Try to pick up can

Rewards/Penalties (points):

- Picks up can: 10
- Tries to pick up can on empty site: -1
- Crashes into wall: -5

Policy A: Always move east, picking up cans as you go

Policy B: Move up and down the columns of the grid, picking up cans as you go



Value function:

Formal definition of "cumulative value" with discounted rewards

Let $V^{\pi}(s_t)$ denote the "cumulative value" of π starting from initial state s_t:

$$
\begin{aligned}
V^{\pi}(s_t) &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \\
&= \sum_{i=0}^{\infty} \gamma^i r_{t+i}
\end{aligned}
$$

- where $0 \leq \gamma \leq 1$ is a "discounting" constant that determines the relative value of delayed versus immediate rewards.
- $V^{\pi}(s_t)$ is called the "value function" for policy π . It gives the expected value of starting in state st and following policy π "forever".

Value function:

Formal definition of "cumulative value" with discounted rewards

Let $V^{\pi}(s_t)$ denote the "cumulative value" of π starting from initial state s_t:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$
$$= \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

Why use a discounting constant?

- where $0 \leq \gamma \leq 1$ is a "discounting" constant that determines the relative value of delayed versus immediate rewards.

- $V^{\pi}(s_t)$ is called the "value function" for policy π . It gives the expected value of starting in state st and following policy π "forever".

- Note that rewards received *i* times steps into the future are discounted exponentially (by a factor of γ^i).

- If $\gamma = 0$, we only care about immediate reward.

- The closer γ is to 1, the more we care about future rewards, relative to immediate reward.

- **Precise specification of learning task:**

  We require that the agent learn policy π that maximizes $V^{\pi}(s)$ for all states s.

  We call such a policy an *optimal* policy, and denote it by π^*:

  To simplify notation, let

$$V^*(s) = V^{\pi^*}(s)$$

What should L learn?

- Hard to learn π^* directly, since we don't have training data of the form (s, a)

- Only training data availale to learner is sequence of rewards:

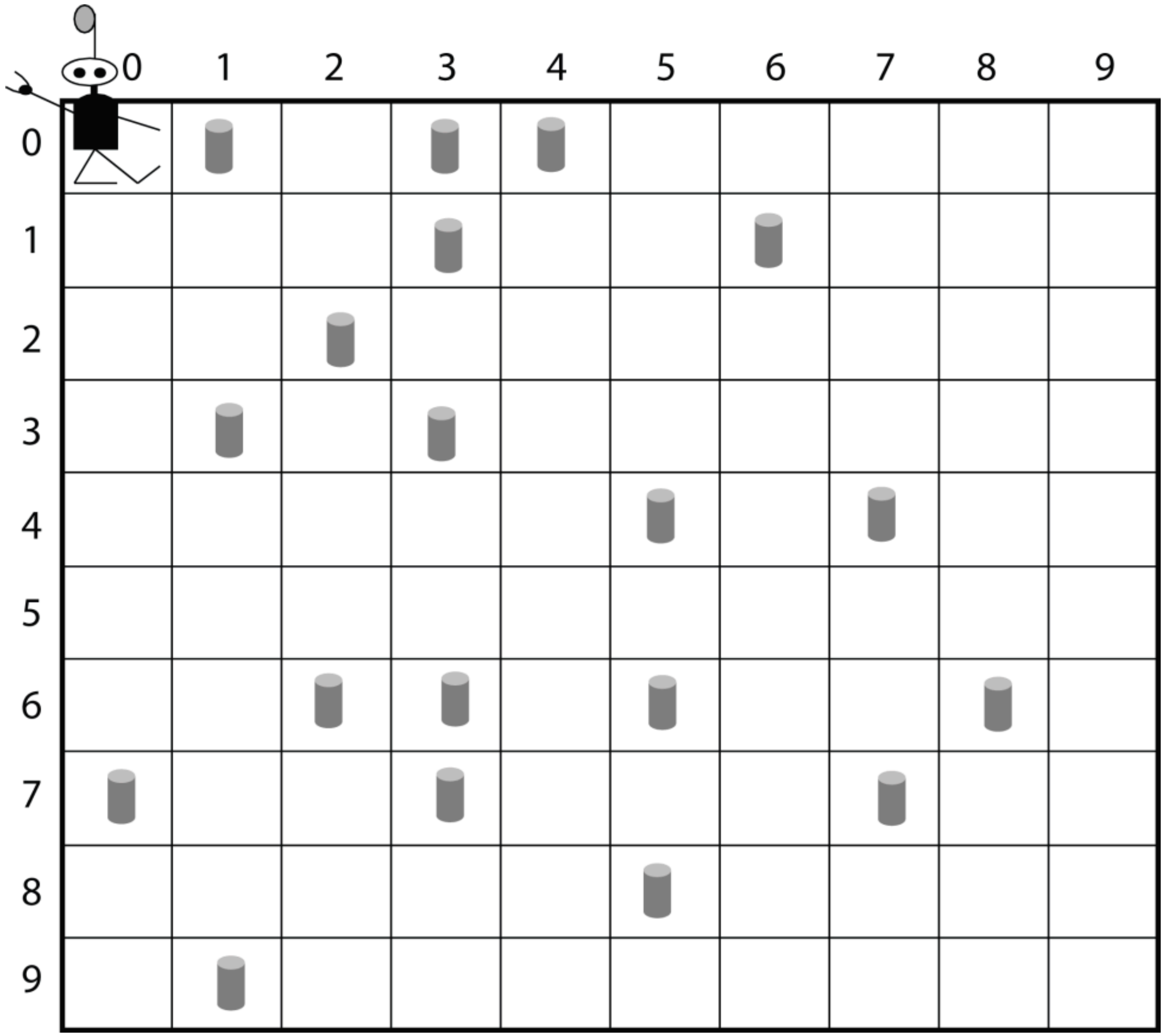
$$r(s_0, a_0), r(s_1, a_1), \ldots$$

- So, what should the learner L learn?

Learn evaluation function V?

- One possibility: learn evaluation function $V^*(s)$.
- Then L would know what action to take next:
  - L should prefer state s_1 over s_2 whenever $V^*(s_1) > V^*(s_2)$
  - Optimal action a in state s is the one that maximizes sum of $r(s,a)$ (immediate reward) and V^* of the immediate successor state, discounted by γ:

$$\pi^*(s) = \underset{a}{\arg\max}\left[r(s,a) + \gamma V^*(\delta(s,a))\right]$$



$$\pi^*(s) = \underset{a}{\arg\max}\left[r(s,a) + \gamma V^*(\delta(s,a))\right]$$

Problem with learning V^*

Problem: Using V^* to obtain optimal policy π^* *requires* perfect knowledge of δ and r, which we earlier said are unknown to L.

Alternative: *Q* Learning

- Alternative: we will estimate the following evaluation function $Q: S \times A \longrightarrow \Re$, as follows:

$$\pi^*(s) = \underset{a}{\arg\max}\left[r(s,a) + \gamma V^*(\delta(s,a))\right]$$
$$= \underset{a}{\arg\max}\left[Q(s,a)\right],$$

$$\text{where } Q(s,a) = r(s,a) + \gamma V^*(\delta(s,a))$$

- Suppose learner L has a good estimate for $Q(s,a)$. Then, at each time step, L simply chooses action that maximizes $Q(s,a)$.

How to learn Q ? (In theory)

We have:

$$\pi^*(s) = \arg\max_{a'}\left[r(s,a') + \gamma V^*(\delta(s,a'))\right]$$

$$V^*(s) = \max_{a}\left[r(s,a') + \gamma V^*(\delta(s,a'))\right]$$

$$= \max_{a'}\left[Q(s,a')\right]$$

So,

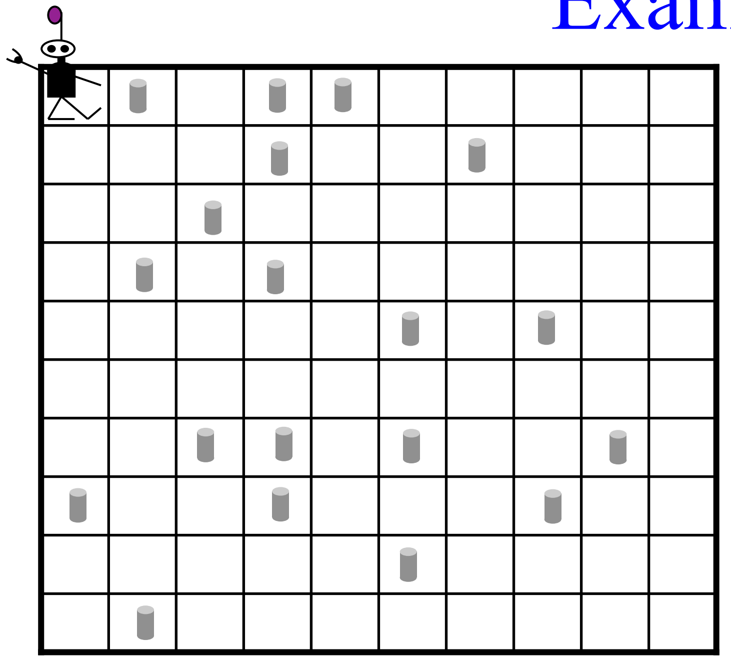
$$Q(s,a) = r(s,a) + \gamma \max_{a'}\left[Q(\delta(s,a),a')\right]$$

We can learn Q via iterative approximation.

How to learn Q ? (In practice)

- Initialize $Q(s,a)$ to small random values for all s and a
- Initialize s
- Repeat forever (or as long as you have time for):
  - Select action a
  - Take action a and receive reward r
  - Observe new state s'
  - Update $Q(s,a) \Leftarrow Q(s,a) + \eta\ (r + \gamma \max_{a'} Q(s',a') - Q(s, a))$
  - Update $s \Leftarrow s'$

Example of Q learning



Let $\gamma = .2$
Let $\eta = 1$

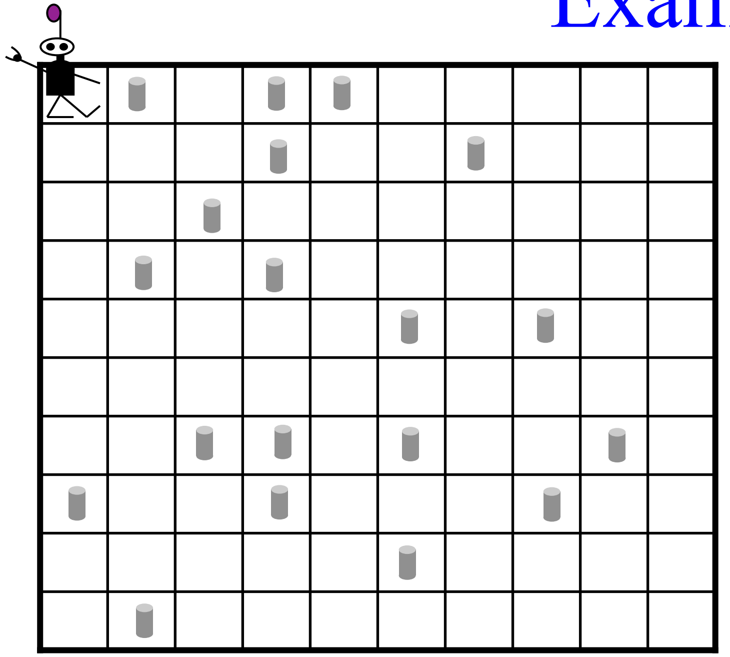
State: North, South, East, West, Current

W = Wall
E = Empty
C = Can

Reward:

$Q(s,a)$	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…							
…							

Example of Q learning



Initialize *Q*(*s*,*a*) to small random values (here, 0) for all *s* and *a*

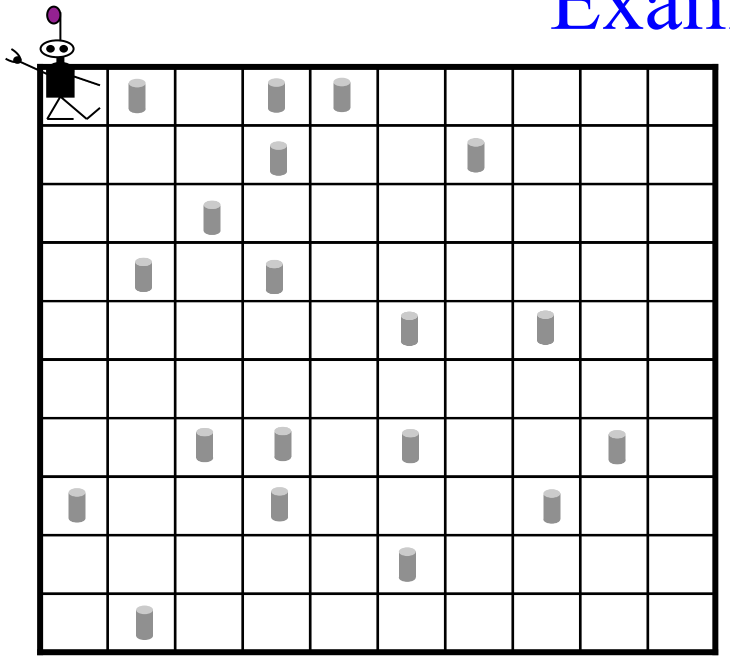
State: North, South, East, West, Current

W = Wall
E = Empty
C = Can

Reward:

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Initialize *s*

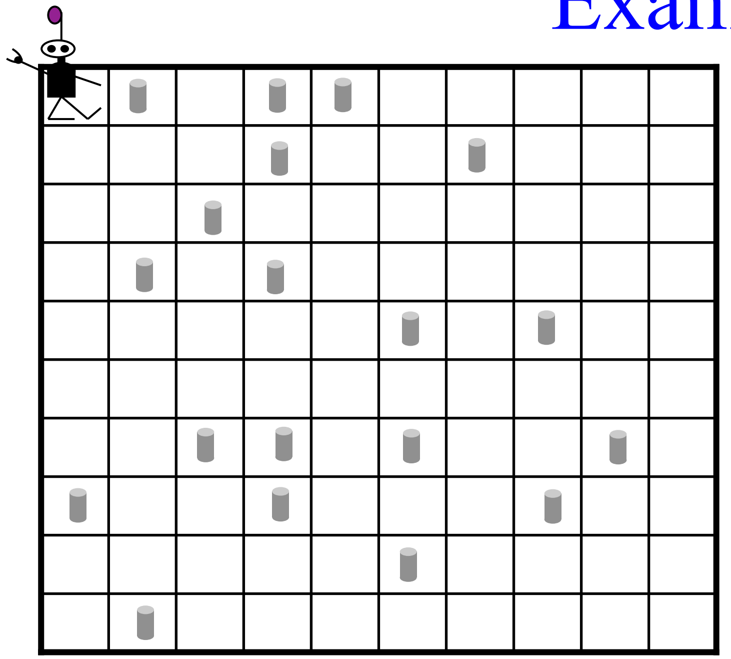
State: North, South, East, West, Current

W = Wall
E = Empty
C = Can

Reward:

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Select action *a*

State: North, South, East, West, Current

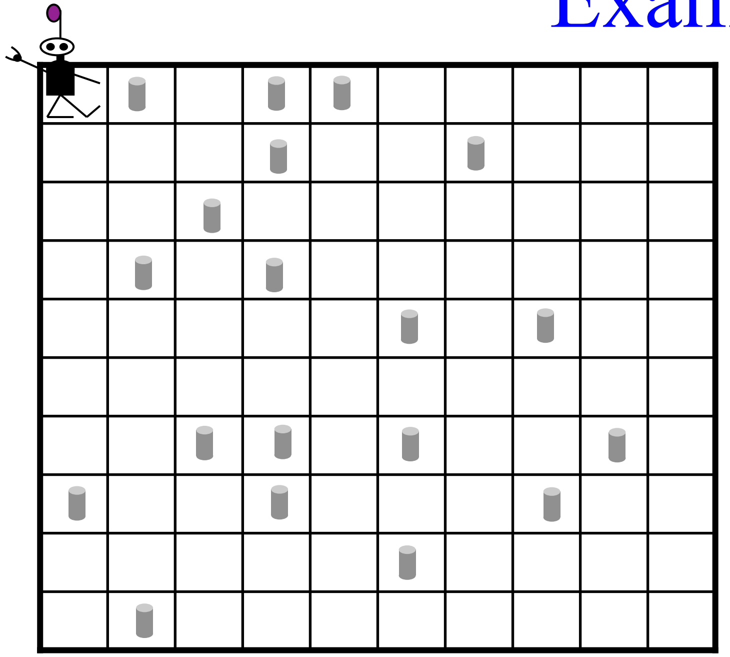
W = Wall
E = Empty
C = Can

Trial: 1

Reward:

Q(*s*,*a*)	MoveN	MoveS	MoveE	**MoveW**	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Perform action *a* and receive reward *r*

State: North, South, East, West, Current

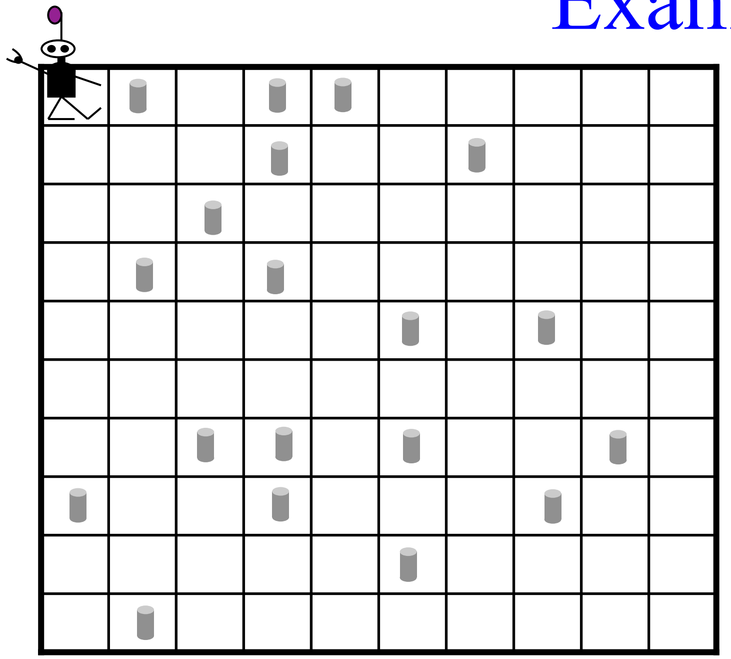
W = Wall
E = Empty
C = Can

Reward: −1

Trial: 1

Q(*s*,*a*)	MoveN	MoveS	MoveE	**MoveW**	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Observe new state *s*´

State: North, South, East, West, Current

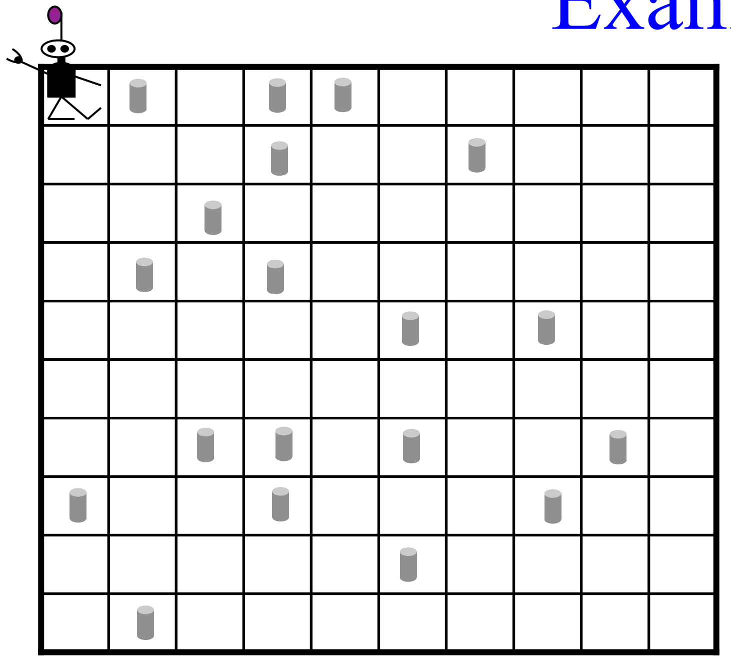
W = Wall
E = Empty
C = Can

Trial: 1

Reward: −1

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Update

$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$

State: North, South, East, West, Current

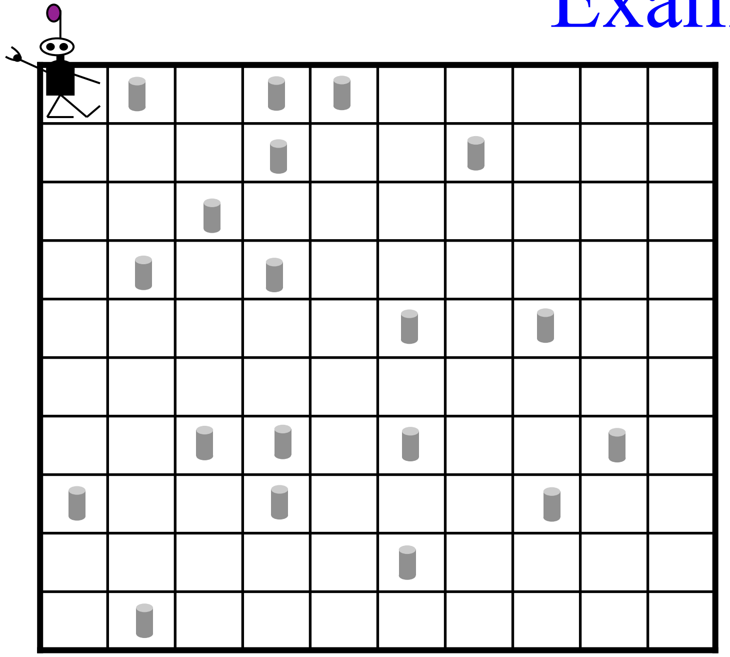
W = Wall
E = Empty
C = Can

Trial: 1

Reward: −1

$Q(s,a)$	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	**0**	0	0	0
	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Update

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

State: North, South, East, West, Current

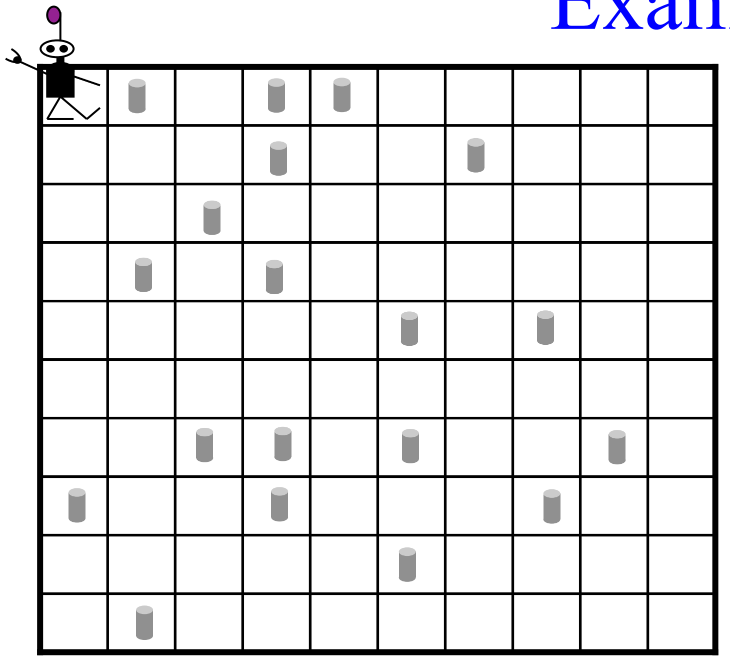
W = Wall
E = Empty
C = Can

Trial: 1

Reward: −1

$Q(s,a)$	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	**-1**	0	0	0
	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Update s ⇐ *s´*

State: North, South, East, West, Current

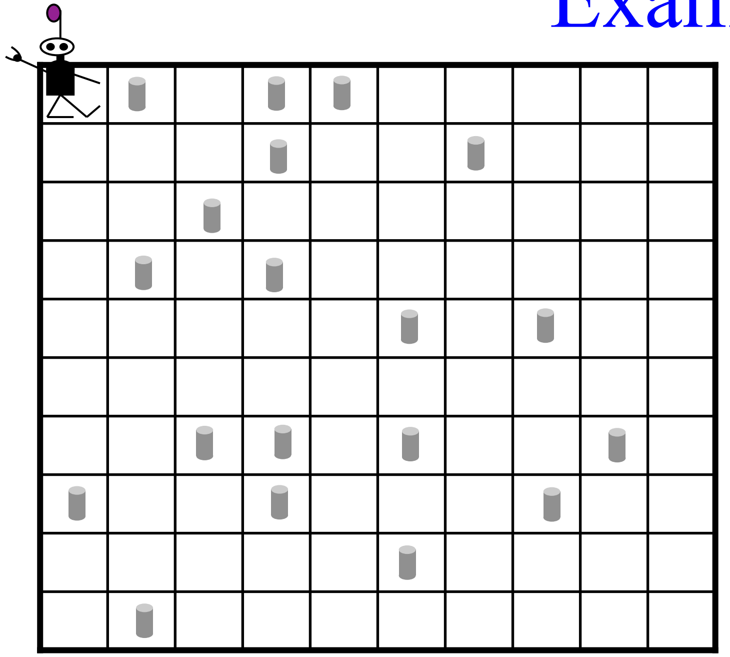
W = Wall
E = Empty
C = Can

Trial: 1

Reward:

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Select action *a*

State: North, South, East, West, Current

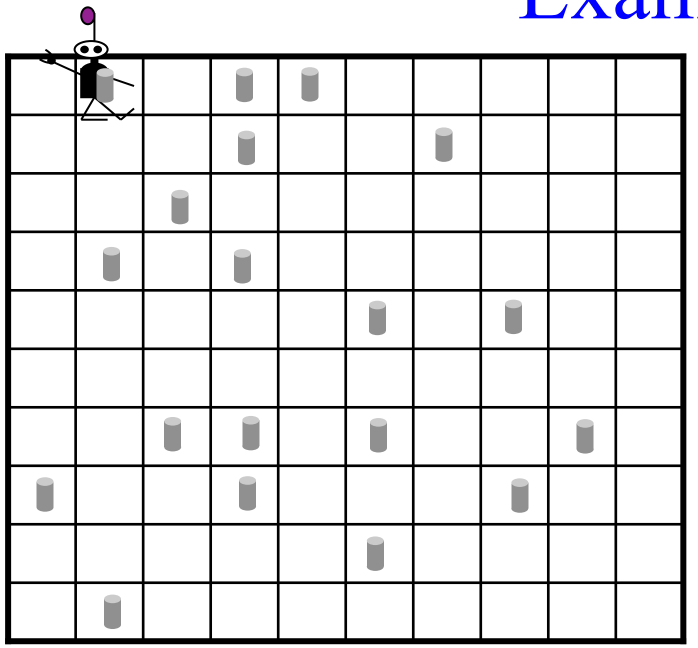
W = Wall
E = Empty
C = Can

Reward:

Trial: 2

Q(*s*,*a*)	MoveN	MoveS	**MoveE**	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Perform action *a* and receive reward *r*

State: North, South, East, West, Current

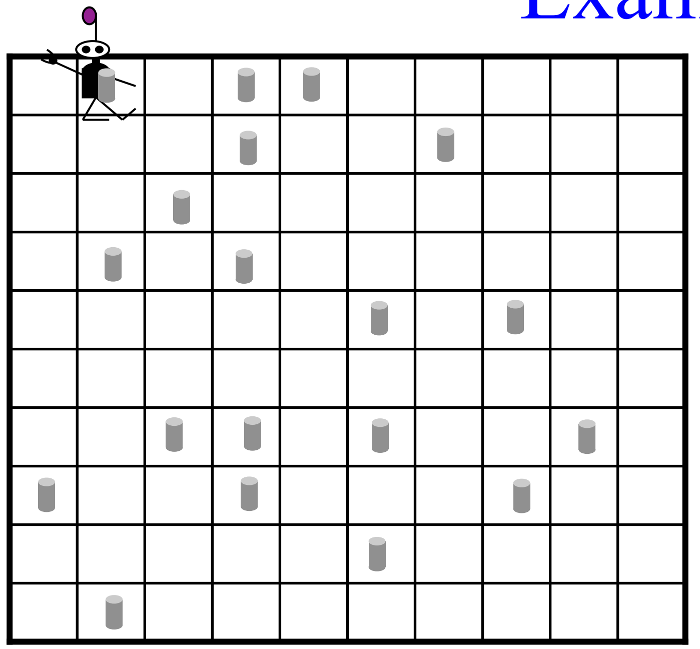
W = Wall
E = Empty
C = Can

Trial: 2

Reward: 0

Q(*s*,*a*)	MoveN	MoveS	**MoveE**	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Observe new state *s*´

State: North, South, East, West, Current

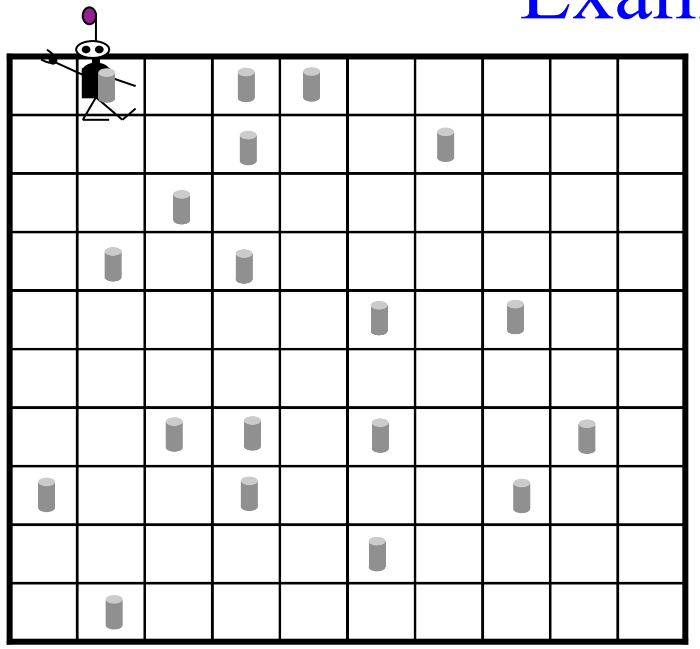
W = Wall
E = Empty
C = Can

Trial: 2

Reward: 0

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Update

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

State: North, South, East, West, Current

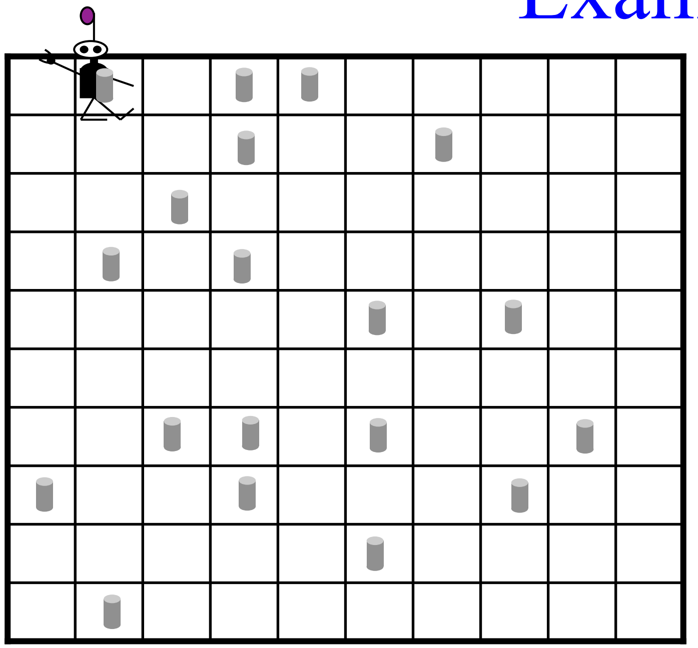
W = Wall
E = Empty
C = Can

Trial: 2

Reward: 0

$Q(s,a)$	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	**0**	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Update

$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$

State: North, South, East, West, Current

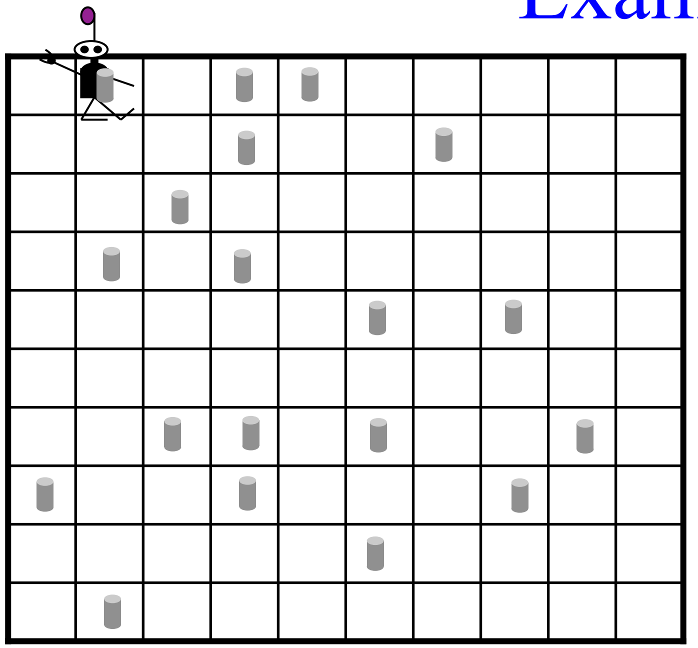
W = Wall
E = Empty
C = Can

Trial: 2

Reward: 0

$Q(s,a)$	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	**0**	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Update s ⇐ *s*´

State: North, South, East, West, Current

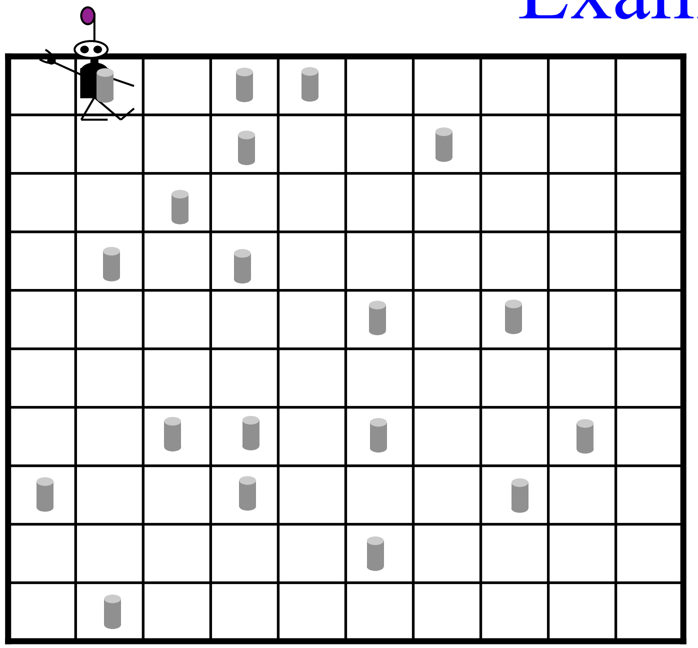
W = Wall
E = Empty
C = Can

Trial: 2

Reward: 0

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Select action *a*

State: North, South, East, West, Current

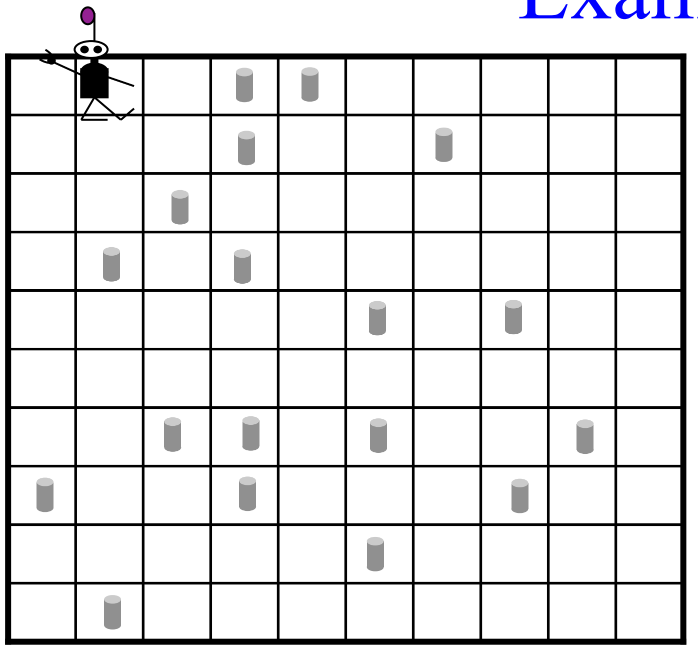
W = Wall
E = Empty
C = Can

Trial: 3

Reward:

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	**Pick Up Can**
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Perform action *a* and receive reward *r*

State: North, South, East, West, Current

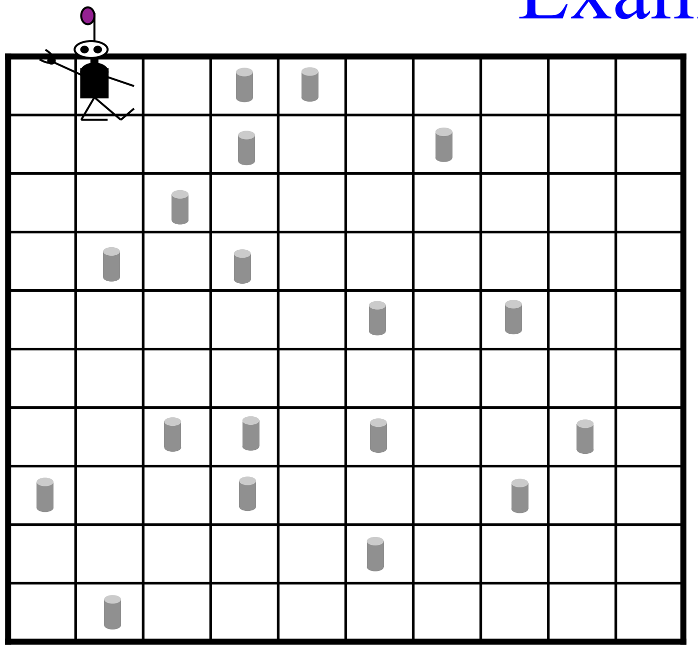
W = Wall
E = Empty
C = Can

Trial: 3

Reward: 10

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	**Pick Up Can**
…	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Observe new state *s´*

State: North, South, East, West, Current

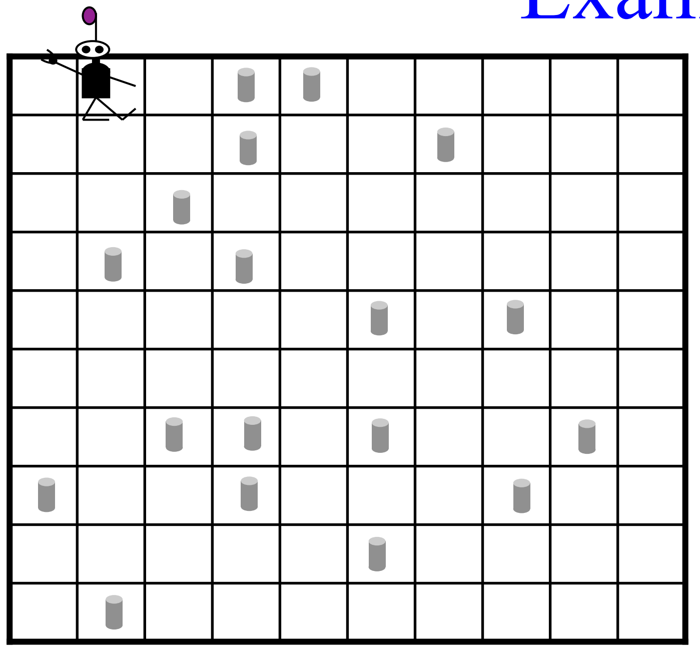
W = Wall
E = Empty
C = Can

Trial: 3

Reward: 10

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	0
…	0	0	0	0	0	0	0

Example of Q learning



Update

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

State: North, South, East, West, Current

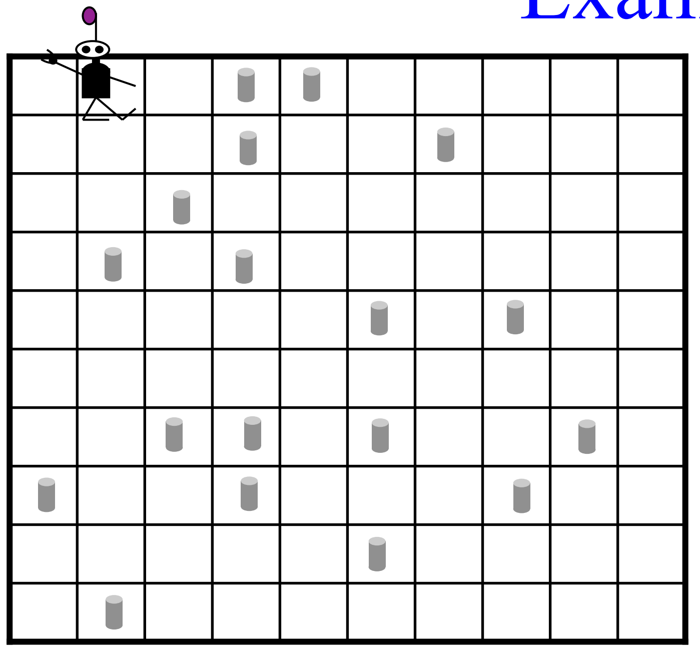
W = Wall
E = Empty
C = Can

Trial: 3

Reward: 10

$Q(s,a)$	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	**0**
…	0	0	0	0	0	0	0

Example of Q learning



Update

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

State: North, South, East, West, Current

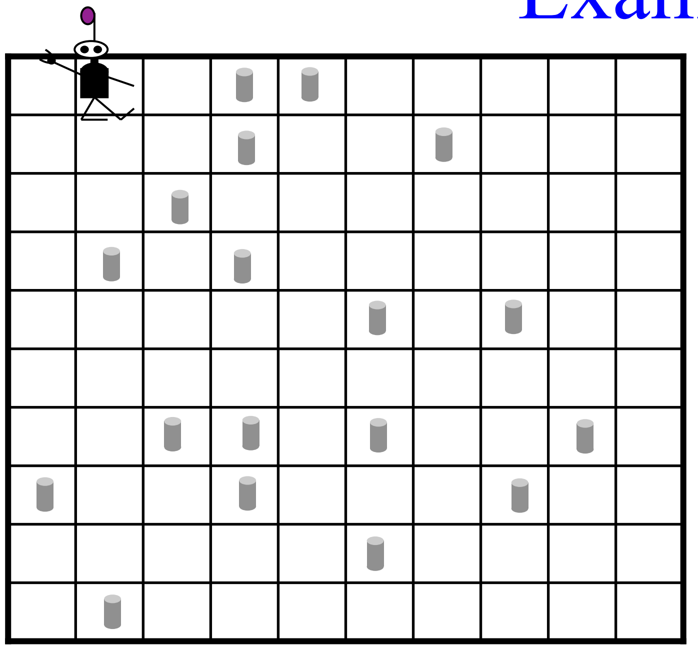
W = Wall
E = Empty
C = Can

Trial: 3

Reward: 10

$Q(s,a)$	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	**10**
…	0	0	0	0	0	0	0

Example of Q learning



Update s ⇐ *s*´

State: North, South, East, West, Current

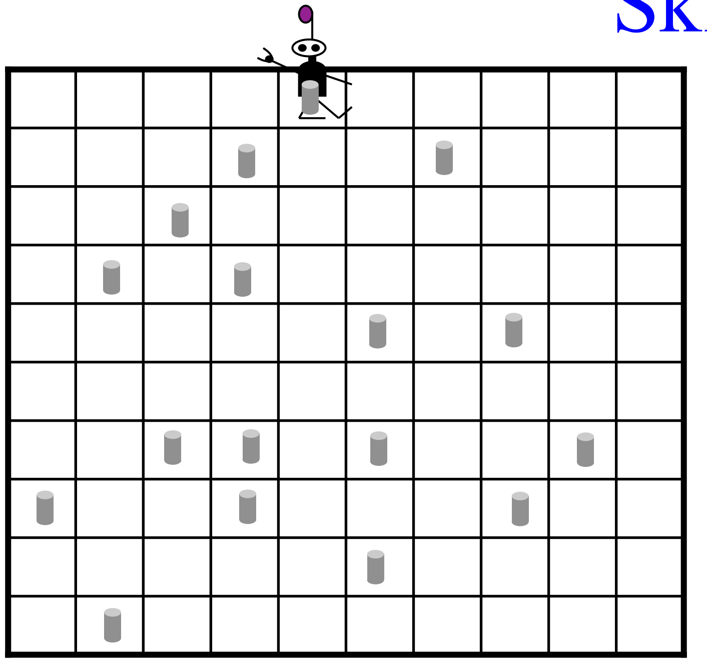
W = Wall
E = Empty
C = Can

Trial: 3

Reward: 10

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	10
…	0	0	0	0	0	0	0

Skipping ahead...



State: North, South, East, West, Current

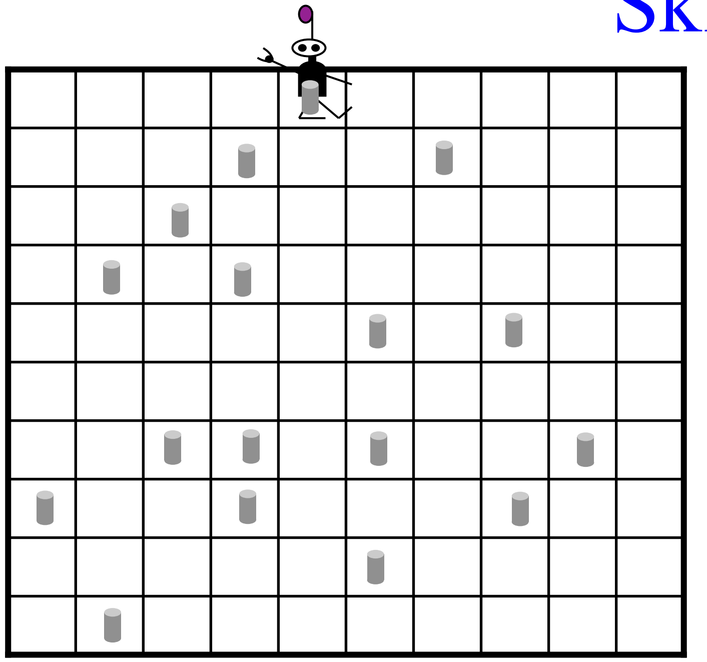
W = Wall
E = Empty
C = Can

Trial: *m*

Reward:

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	10
…	0	0	0	0	0	0	0

Skipping ahead...



Select action *a*

State: North, South, East, West, Current

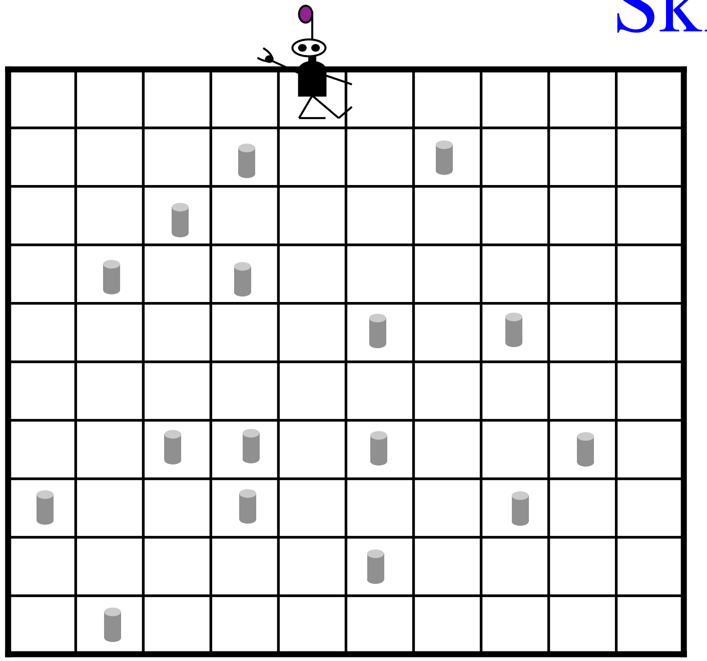
W = Wall
E = Empty
C = Can

Trial: ***m***

Reward:

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	**Pick Up Can**
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	10
…	0	0	0	0	0	0	0

Skipping ahead...



Perform action *a* and receive reward *r*

State: North, South, East, West, Current

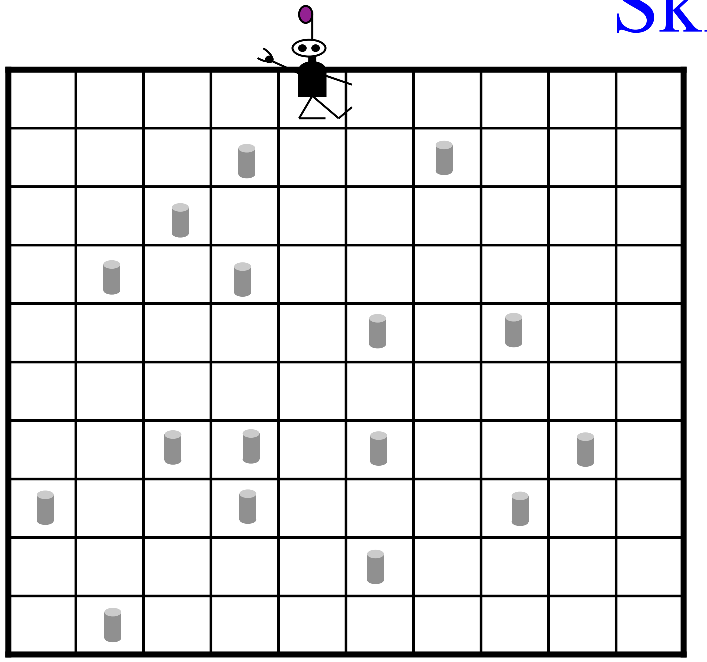
W = Wall
E = Empty
C = Can

Trial: *m*

Reward: 10

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	**Pick Up Can**
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	10
…	0	0	0	0	0	0	0

Skipping ahead...



Observe new state *s*´

State: North, South, East, West, Current

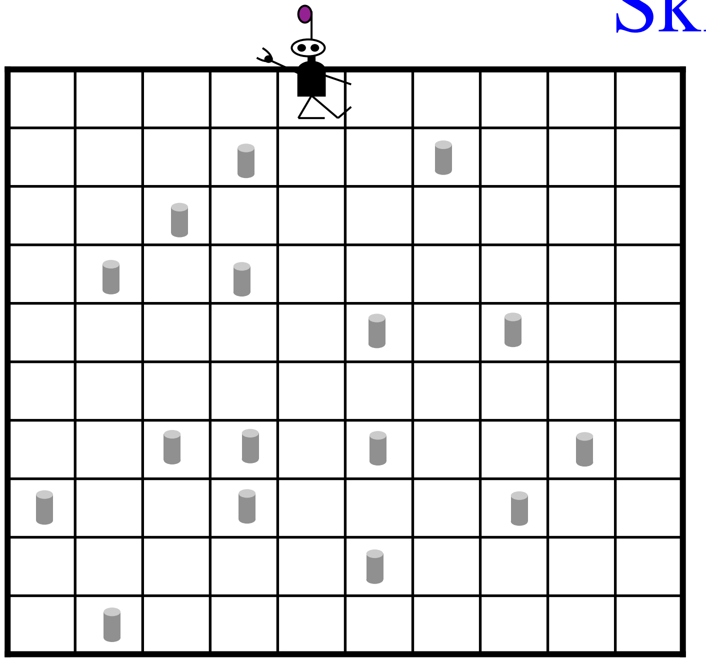
W = Wall
E = Empty
C = Can

Trial: *m*

Reward: 10

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	**Pick Up Can**
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	10
…	0	0	0	0	0	0	0

Skipping ahead...



Update

$Q(s,a) \Leftarrow Q(s,a) + \eta\ (r + \gamma \max_{a'} Q(s',a') - Q(s, a))$

State: North, South, East, West, Current

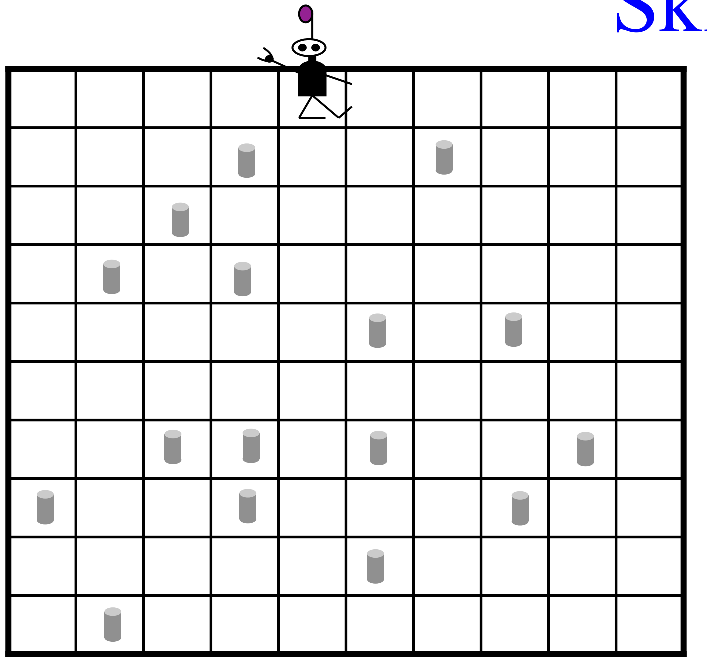
W = Wall
E = Empty
C = Can

Trial: *m*

Reward: 10

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	**Pick Up Can**
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	**10**
…	0	0	0	0	0	0	0

Skipping ahead...



Update

$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$

State: North, South, East, West, Current

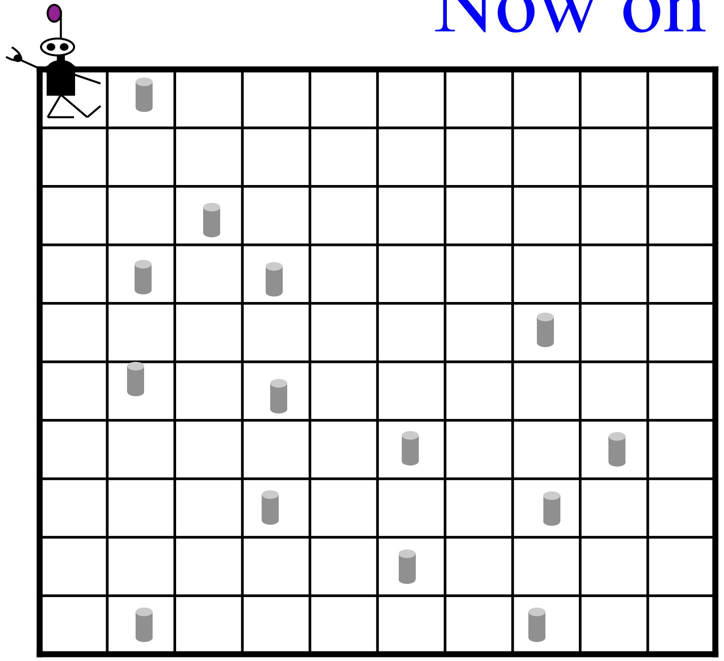
W = Wall
E = Empty
C = Can

Trial: *m*

Reward: 10

$Q(s,a)$	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	**Pick Up Can**
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	**10**
…	0	0	0	0	0	0	0

Now on a new environment...



Select action *a*

State: North, South, East, West, Current

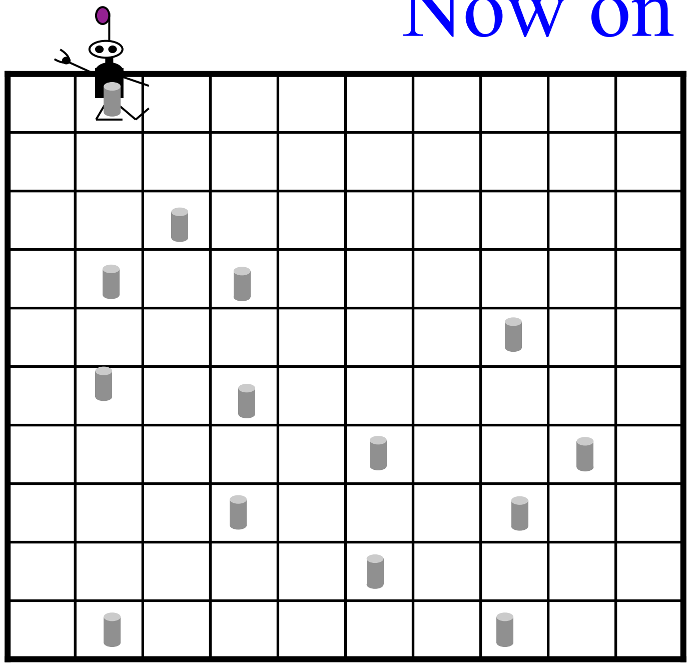
W = Wall
E = Empty
C = Can

Trial: *n*

Reward:

Q(*s*,*a*)	MoveN	MoveS	**MoveE**	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	10
…	0	0	0	0	0	0	0

Now on a new environment...



Perform action *a* and receive reward *r*

State: North, South, East, West, Current

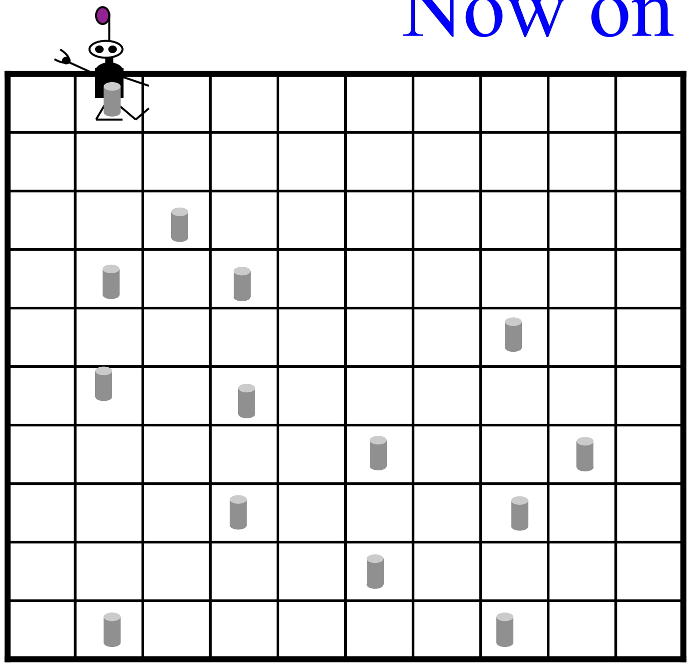
W = Wall
E = Empty
C = Can

Trial: *n*

Reward: 0

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	10
…	0	0	0	0	0	0	0

Now on a new environment...



Observe new state *s´*

State: North, South, East, West, Current

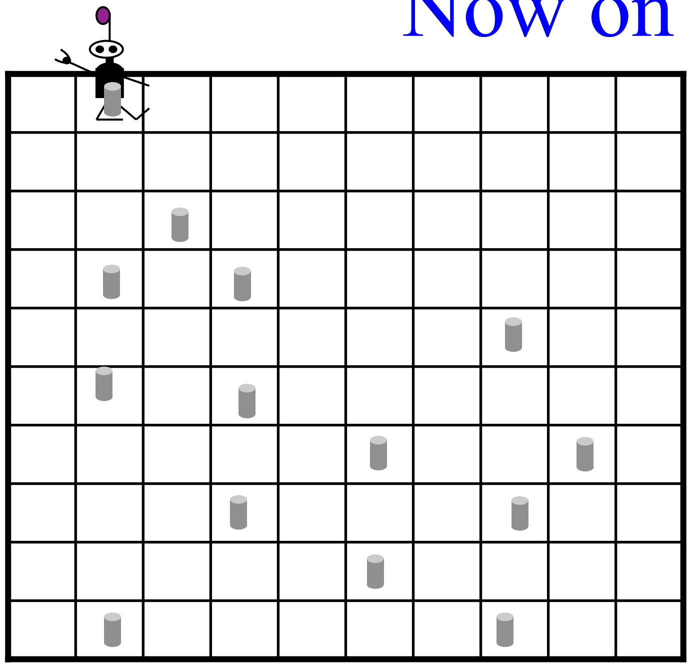
W = Wall
E = Empty
C = Can

Trial: *n*

Reward: 0

Q(*s*,*a*)	MoveN	MoveS	**MoveE**	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	0	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	10
…	0	0	0	0	0	0	0

Now on a new environment...



Update

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

State: North, South, East, West, Current

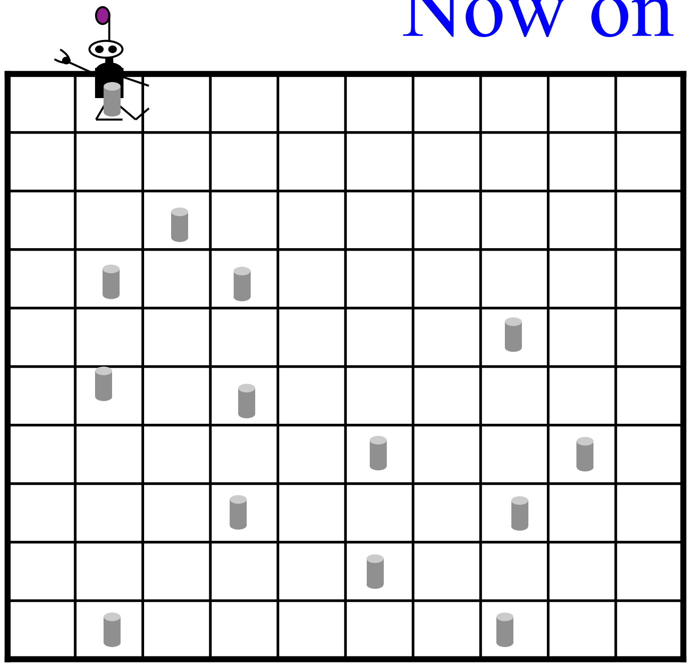
W = Wall
E = Empty
C = Can

Trial: *n*

Reward: 0

$Q(s,a)$	MoveN	MoveS	**MoveE**	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	**0**	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	10
…	0	0	0	0	0	0	0

Now on a new environment...



Update

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

State: North, South, East, West, Current

W = Wall
E = Empty
C = Can

Trial: *n*

Reward: 0

Q(*s*,*a*)	MoveN	MoveS	MoveE	MoveW	Move Random	Stay Put	Pick Up Can
…	0	0	0	0	0	0	0
W, E, E, E, E	0	0	0	0	0	0	0
W, E, C, W, E	0	0	2	-1	0	0	0
W, E, E, E, C	0	0	0	0	0	0	10
…	0	0	0	0	0	0	0

How to choose actions?

- Naïve strategy: at each time step, choose action that maximizes $Q(s,a)$

- This exploits current Q but doesn't further explore the state-action space (in case Q is way off)

- Common in Q learning to use probabilistic approach, e.g.,

$$P(a_i \mid s) = \frac{e^{Q(s,a_i)/T}}{\sum_j e^{Q(s,a_i)/T}}, \quad T > 0$$

- This balances exploitation and exploration in a tunable way:
  - high *T:* more exploration (more random)
  - low *T:* more exploitation (more deterministic)

- Can start with high *T*, and decrease it as improves

Representation of $Q(s, a)$

- Note that in all of the above discussion, $Q(s, a)$ was assumed to be a look-up table, with a distinct table entry for each distinct (s,a) pair.

- More commonly, $Q(s, a)$ is represented as a function (e.g., as a neural network), and the function is estimated (e.g., through back-propagation).

Example: Learning to play backgammon

Rules of backgammon

Complexity of Backgammon

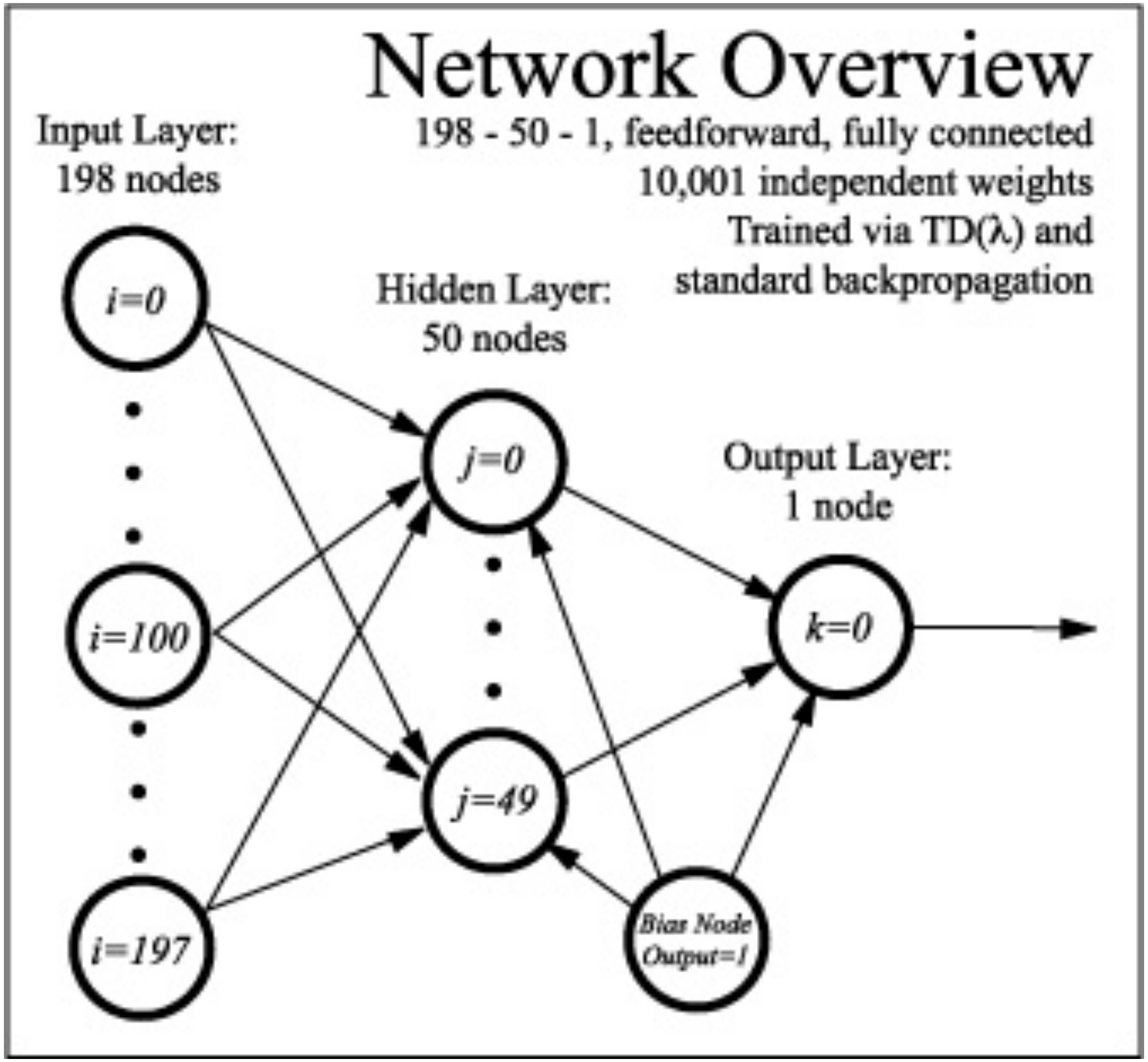
- Over 10^{20} possible states.
- At each ply, 21 dice combinations, with average of about 20 legal moves per dice combination. Result is branching ratio of several hundred per ply.
- Chess has branching ratio of about 30-40 per ply.
- Brute-force look-ahead search is not practical!

Neurogammon (Tesauro, 1989)

- Used supervised learning approach: multilayer NN trained by back-propagation on data base of recorded expert games.
- Input: raw board information (number of pieces at each location), and a few hand-crafted features that encoded important expert concepts.
- Neurogammon achieved strong intermediate level of play.
- Won backgammon championship at 1989 International Computer Olympiad. But not close to beating best human players.

TD-Gammon (G. Tesauro, 1994)

- Program had two main parts:
  - **Move Generator**: Program that generates all legal moves from current board configuration.
  - **Predictor network**: multi-layer NN that predicts $Q(s,a)$: probability of winning the game from the current board configuration.
- Predictor network scores all legal moves. Highest scoring move is chosen.
- **Rewards:** Zero for all time steps except those on which game is won or lost.



Network Overview
198 - 50 - 1, feedforward, fully connected
10,001 independent weights
Trained via TD(λ) and
standard backpropagation
Input Layer:
198 nodes
Hidden Layer:
50 nodes
Output Layer:
1 node
i=0
i=100
i=197
j=0
j=49
k=0
Bias Node
Output=1

- Input: 198 units
  - 24 positions, 8 input units for each position (192 input units)
    - First 4 input units of each group of 8 represent # white pieces at that position,
    - Second 4 represent # black units at that position
  - Two inputs represent who is moving (white or black)
  - Two inputs represent pieces on the bar
  - Two inputs represent number of pieces borne off by each player.

- 50 hidden units
- 1 output unit (activation represents probability that white will win from given board configuration)

Program plays against itself.

On each turn:

- Use network to evaluate all possible moves from current board configuration. Choose the move with the highest (lowest as black) evaluation. This produces a new board configuration.

- If this is end of game, run back-propagation, with target output activation of 1 or 0 depending on whether white won or lost.

- Else evaluate new board configuration with neural network. Calculate difference between current evaluation and previous evaluation.

- Run back-propagation, using the current evaluation as desired output, and the board position previous to the current move as the input.

Program	Training Games	Opponents	Results
TDG 1.0	300,000	Robertie, Davis, Magriel	-13 pts/51 games (-0.25 ppg)
TDG 2.0	800,000	Goulding, Woolsey, Snellings, Russell, Sylvester	-7 pts/38 games (-0.18 ppg)
TDG 2.1	1,500,000	Robertie	-1 pt/40 games (-0.02 ppg)

Table 1. Results of testing TD-Gammon in play against world-class human opponents. Version 1.0 used 1-play search for move selection; versions 2.0 and 2.1 used 2-ply search. Version 2.0 had 40 hidden units; versions 1.0 and 2.1 had 80 hidden units.

- From Sutton & Barto, *Reinforcement Learning: An Introduction*:

  "After playing about 300,000 games against itself, TD-Gammon 0.0 as described above learned to play approximately as well as the best previous backgammon computer programs."

  "TD-Gammon 3.0 appears to be at, or very near, the playing strength of the best human players in the world. It may already be the world champion. These programs have already changed the way the best human players play the game. For example, TD-Gammon learned to play certain opening positions differently than was the convention among the best human players. Based on TD-Gammon's success and further analysis, the best human players now play these positions as TD-Gammon does (Tesauro, 1995)."

Applying RL to Robotics

Robosoccer with RL-Decision-tree learning:

http://www.youtube.com/watch?v=mRpX9DFCdwI

Summary:
Q learning algorithm

– For each (s, a), initialize $Q(s,a)$ to be zero (or small value).

– Observe the current state s.

– Do forever:

  - Select an action a and execute it.
  - Receive immediate reward r
  - Learn:
    – Observe the new state s′
    – Update the table entry for Q(s,a) as follows:

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

  - $s \leftarrow s'$