Reinforcement Learning (RL)

- agents learn by direct interaction with their environment
- assumption: the goals of the agents can be defined in terms of a *reward* function
- this is the ideal setting for learning with reactive agents!
- the framework on which this type of learning works is known as a *Markov Decision Process*

Markov Decision Process (MDP)

Let S be a set of states and A a set of actions.

At time step t, the agent is in state s_t and performs action a_t. The environment responds with a reward $r_t = r(s_t, a_t)$ and by producing a succeeding state $s_{t+1} = \delta(s_t, a_t)$

Goal: learn a policy $\pi : S \rightarrow A$

Policies in MDPs

What is a good policy in an MDP?

Typically, it is one that maximises *discounted cumulative reward* i.e. $V^{\pi}(s_t) = \sum_{i \geq 0} \gamma^i r_{t+i}$

where $0 \leq \gamma \leq 1$ is a discount factor and r_{t+i} is the reward received i timesteps into the future while performing actions according to policy π.

NB: for $\gamma = 0$ the agent is interested only in immediate reward.

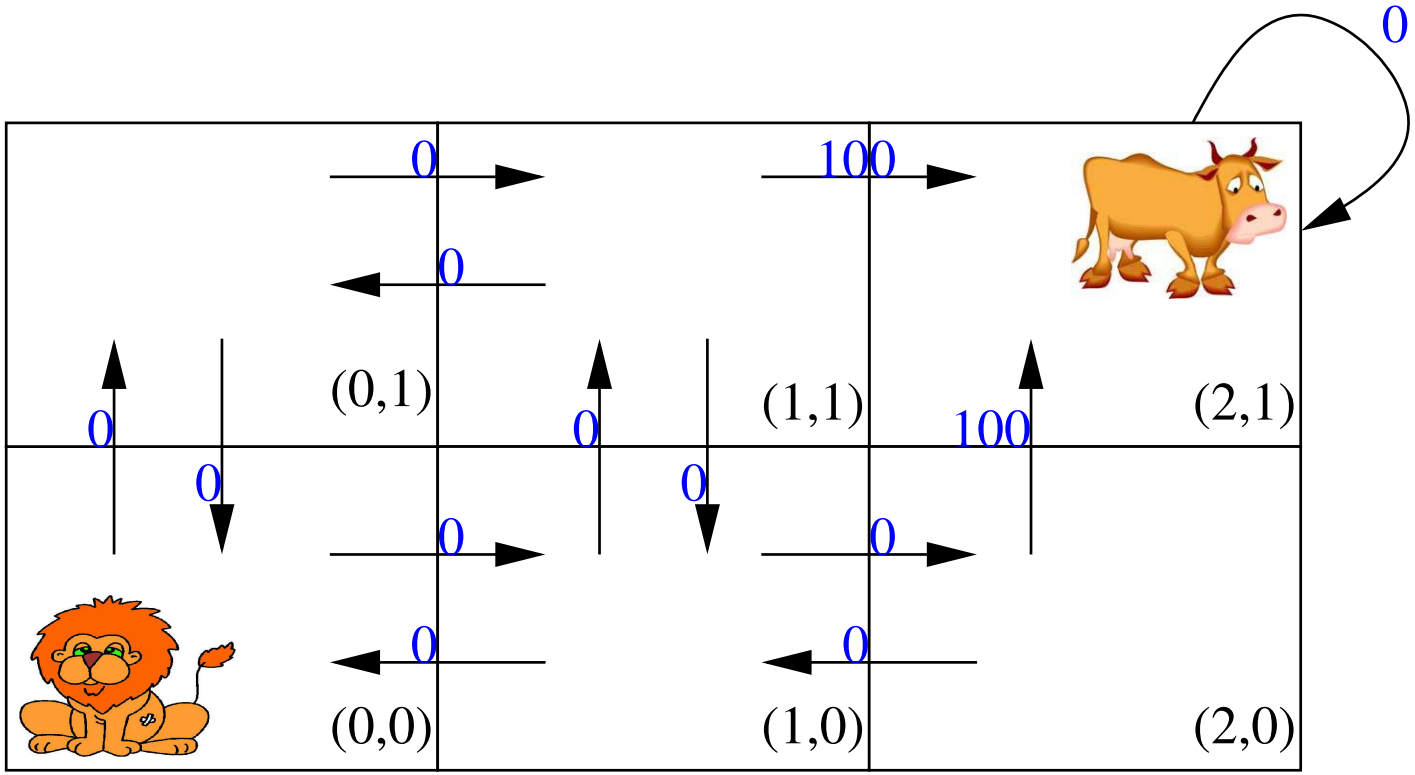
The optimal MDP policy

Goal: agent learns a policy π that maximises $V^{\pi}(s)$ for all states s.

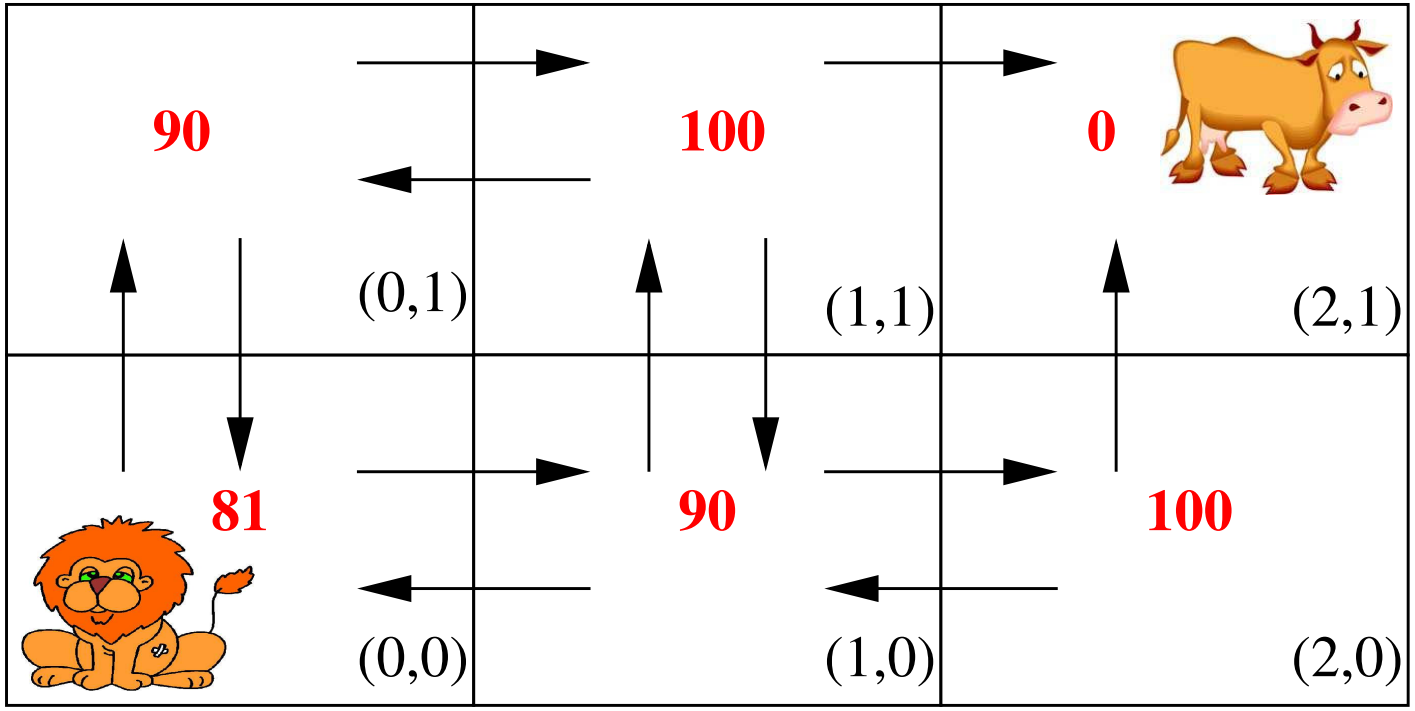
Optimal policy $\pi^* = \arg\max_{\pi} V^{\pi}(s)$

NB: for simplicity, we often write $V^*(s)$ to mean $V^{\pi^*}(s)$, the value of the optimal policy.

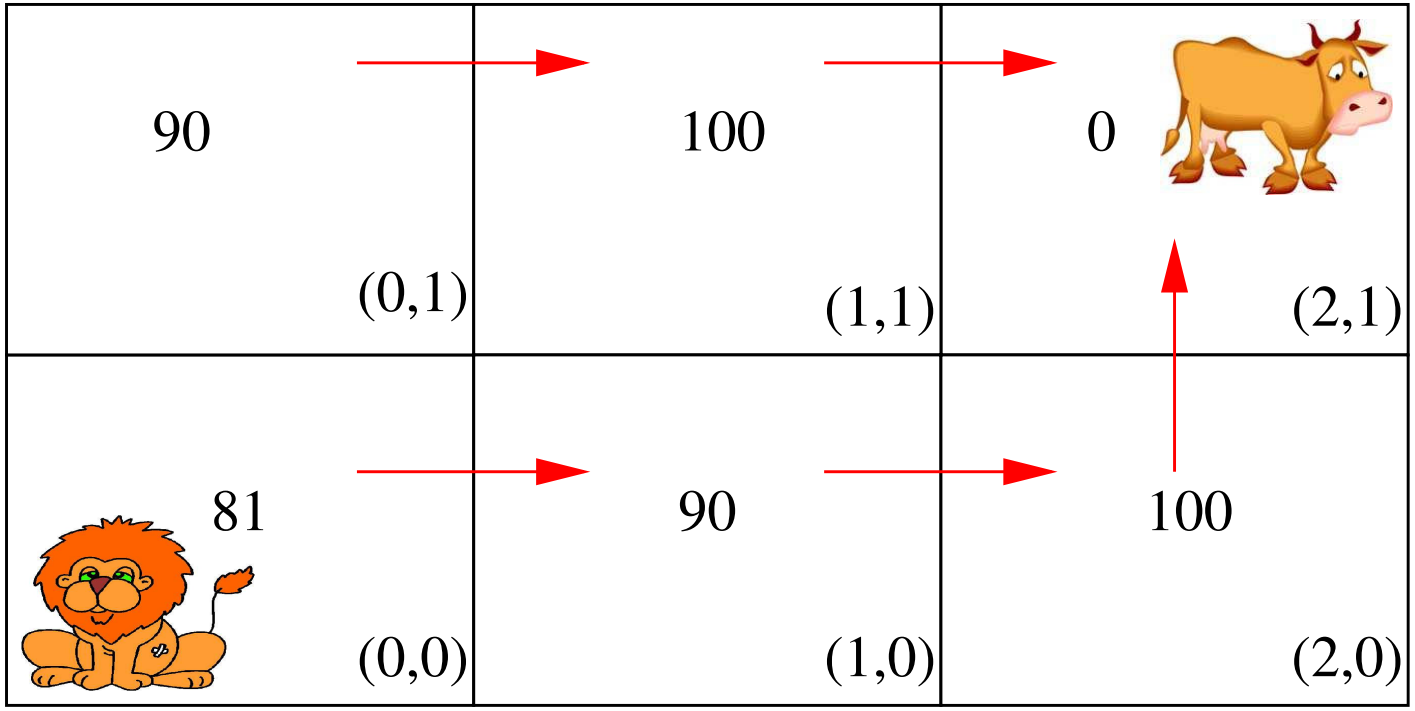
MDP: reward function $r(s,a)$



MDP: $V^*(s)$ for $\gamma = 0.9$



MDP: optimal policy



Learning the optimal value function V^*

The optimal action in state s is the action a that maximises the sum of immediate reward $r(s,a)$ plus the value $V*$ of the immediate successor state, discounted by γ

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

If functions δ and r are known, then the agent can acquire π^* by learning V^*, which is an evaluation function on states.

Q Learning

What if δ and r are unknown?

Learn the Q function:
$Q(s,a) = r(s,a) + \gamma V^*(\delta(s,a))$

It is possible to learn pi^* if both gamma and r are unknown by learning Q!

Q learning: algorithm (1/2)

- $V^*(s) = \max_{a'} Q(s, a')$
- $Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a')$

Question: How do we learn this new Q function?

Answer: Observe reward and update the estimate $\hat{Q}$ of the actual function accordingly.

Q Learning: algorithm (2/2)

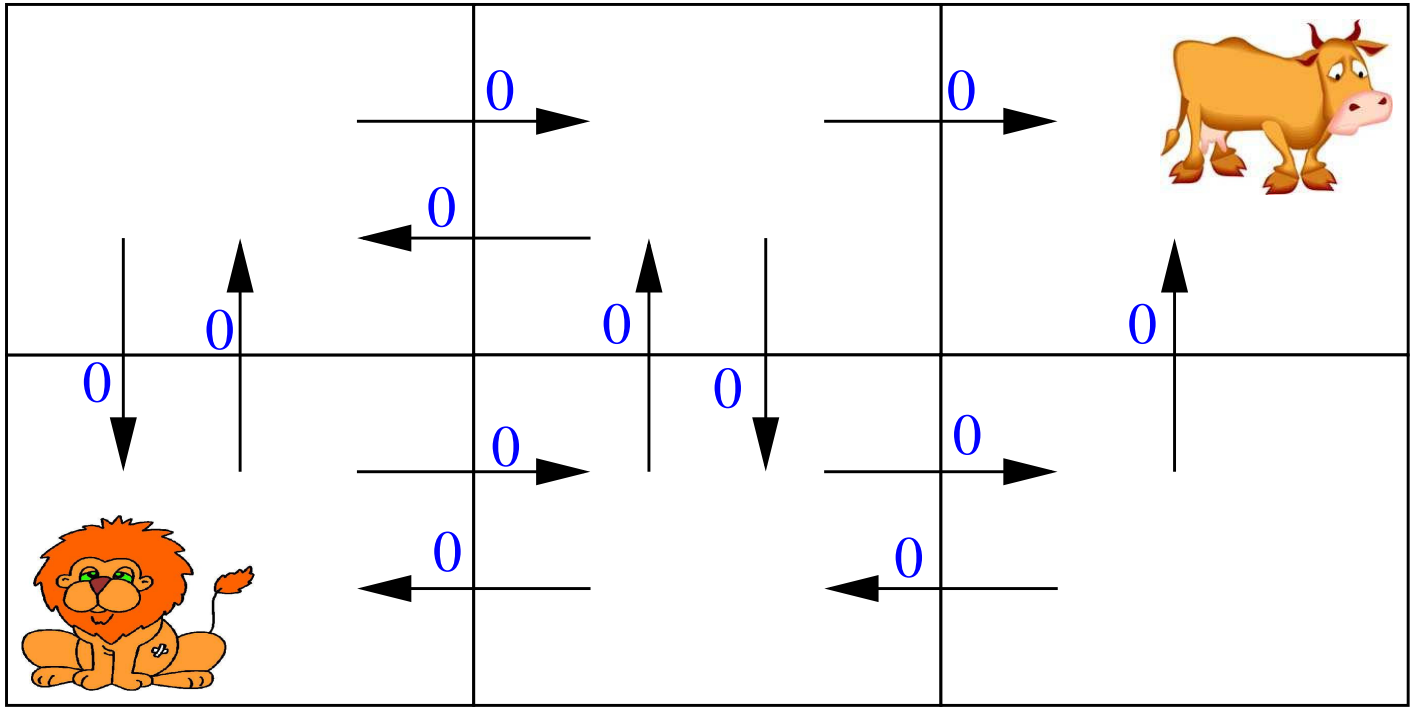
Compute estimate $\hat{Q}$ of Q
foreach *state s and action a* **do**
  $\hat{Q}(s,a) = 0;$
end
while *true* **do**
  select action a and execute it;
  observe new state s' and reward r;
  $\hat{Q}(s,a) = r + \gamma \max_{a'} \hat{Q}(s',a');$
  $s = s';$
end

Q Learning: setting

The task works in episodes because the goal is an *absorbing state*

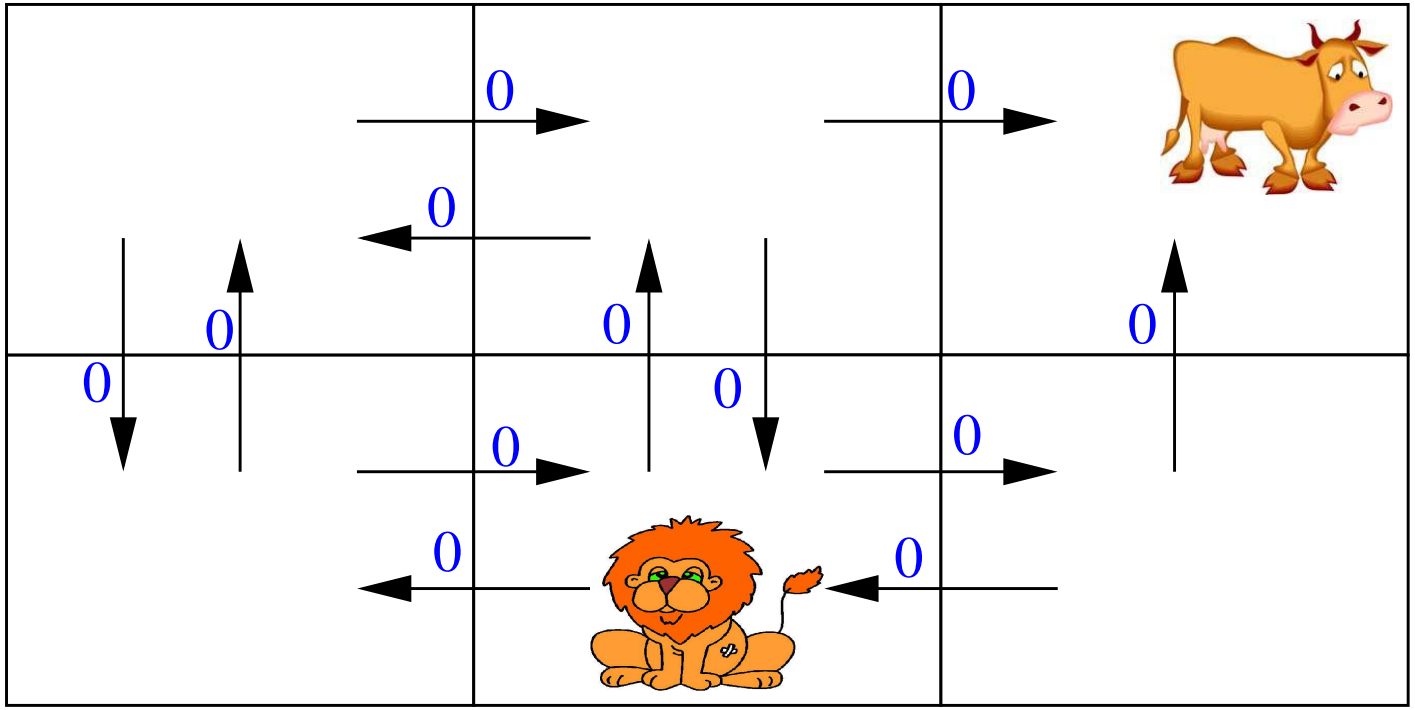
1. the lion starts in some state of the world randomly
2. the lion chooses a move to execute
3. the lion updates the $\hat{Q}$ estimate of the state-action pair
4. this continues until the lion gets to the goal

Q Learning: trace (1/)



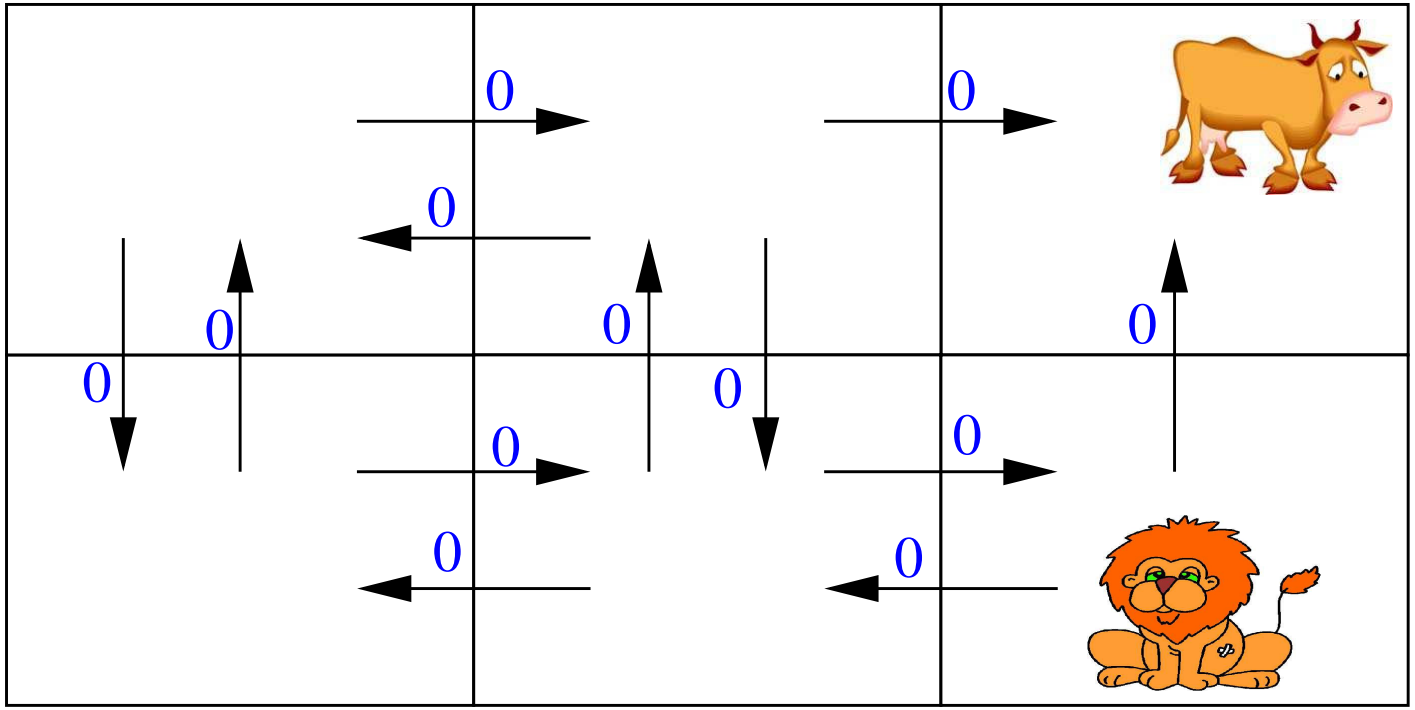
The lion starts in the bottom left corner.

Q Learning: trace (2/)



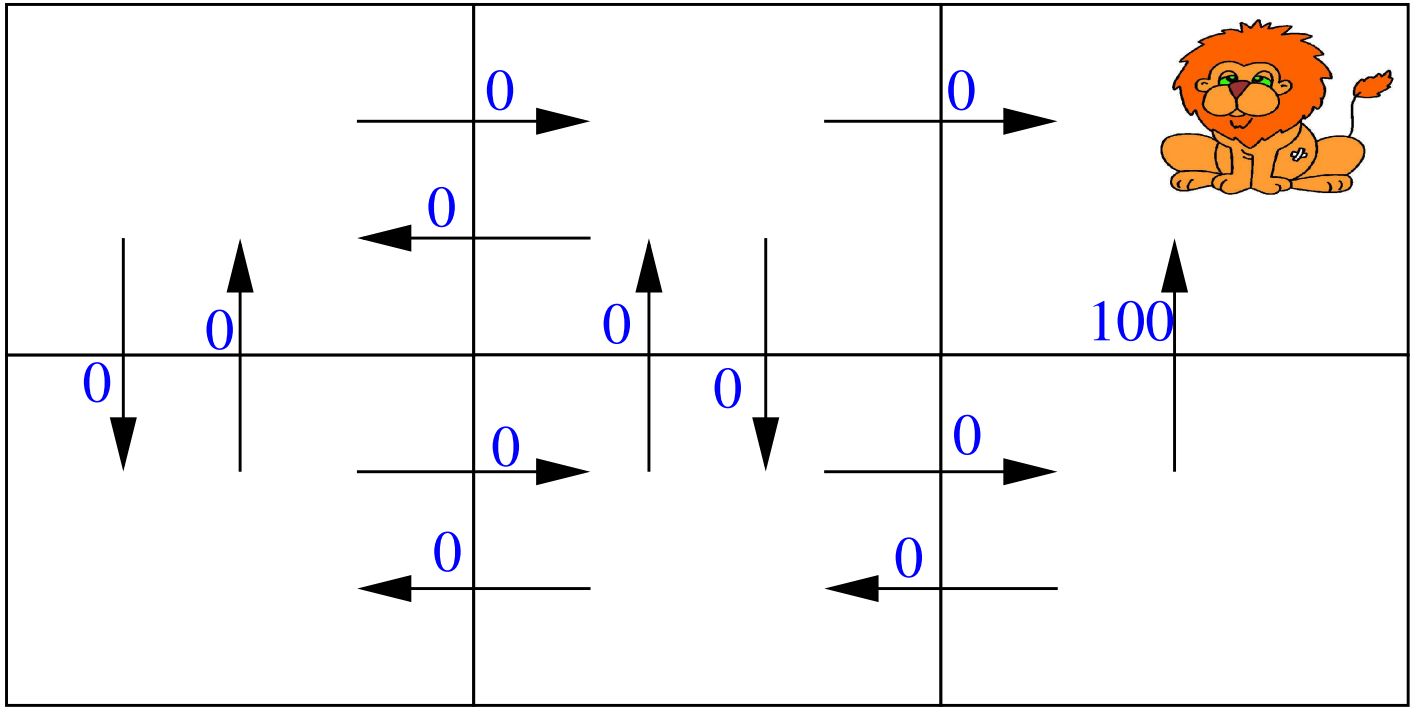
The lion makes a move to the right.

Q Learning: trace (3/)



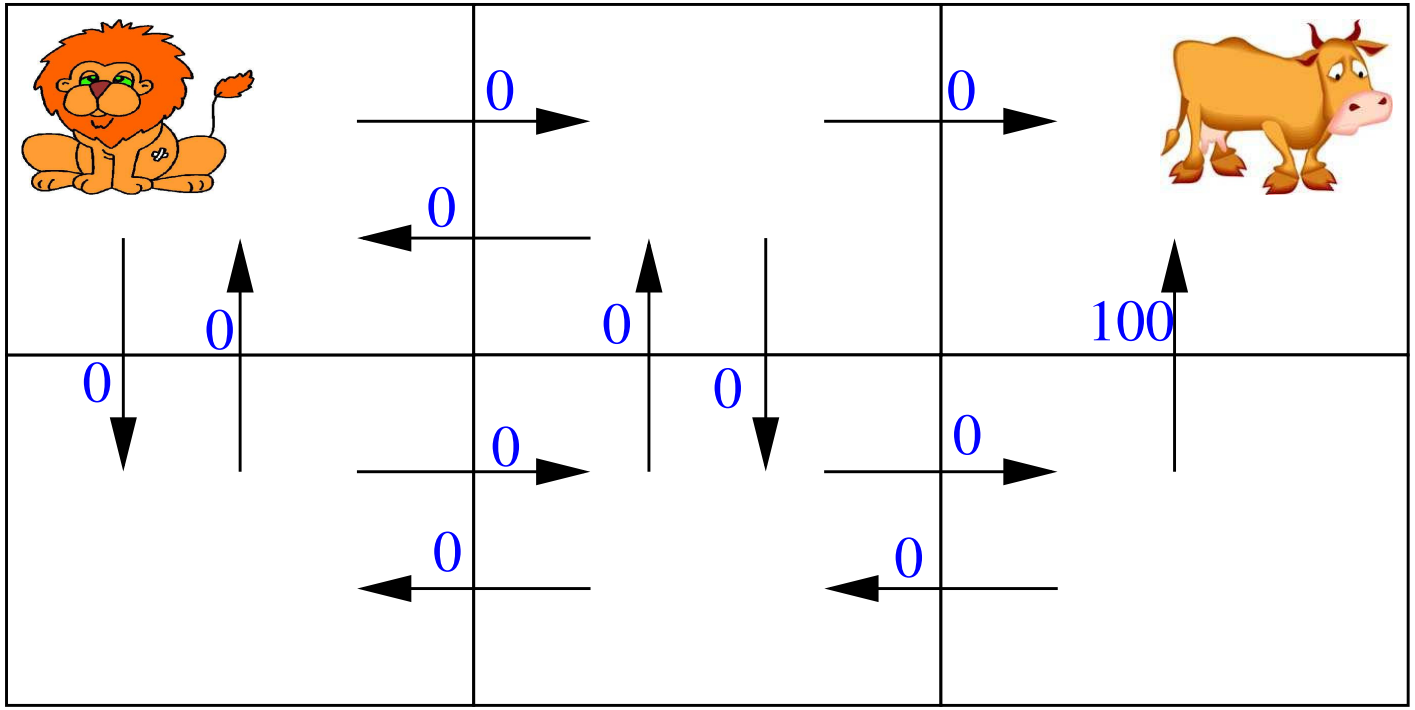
The lion makes another move to the right.

Q Learning: trace (4/)



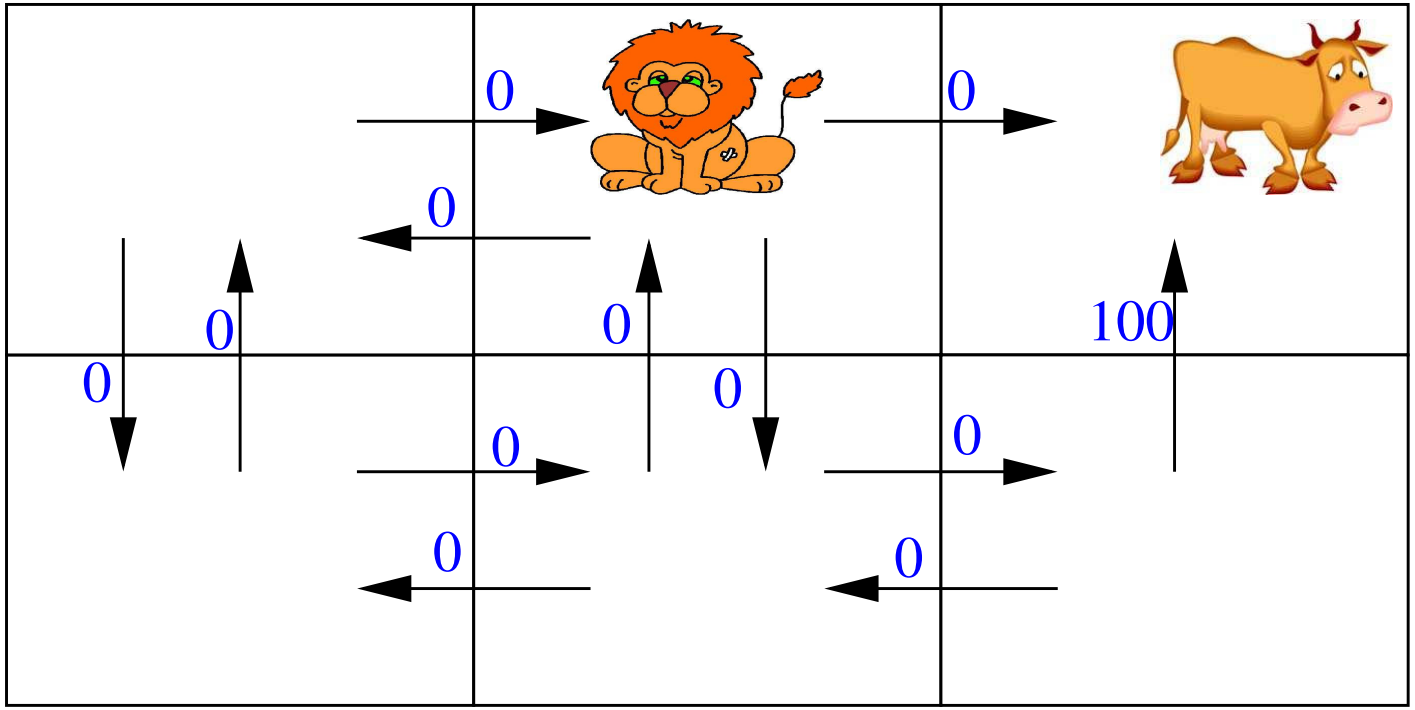
The lion goes up. Beef sandwich anyone?

Q Learning: trace (5/)



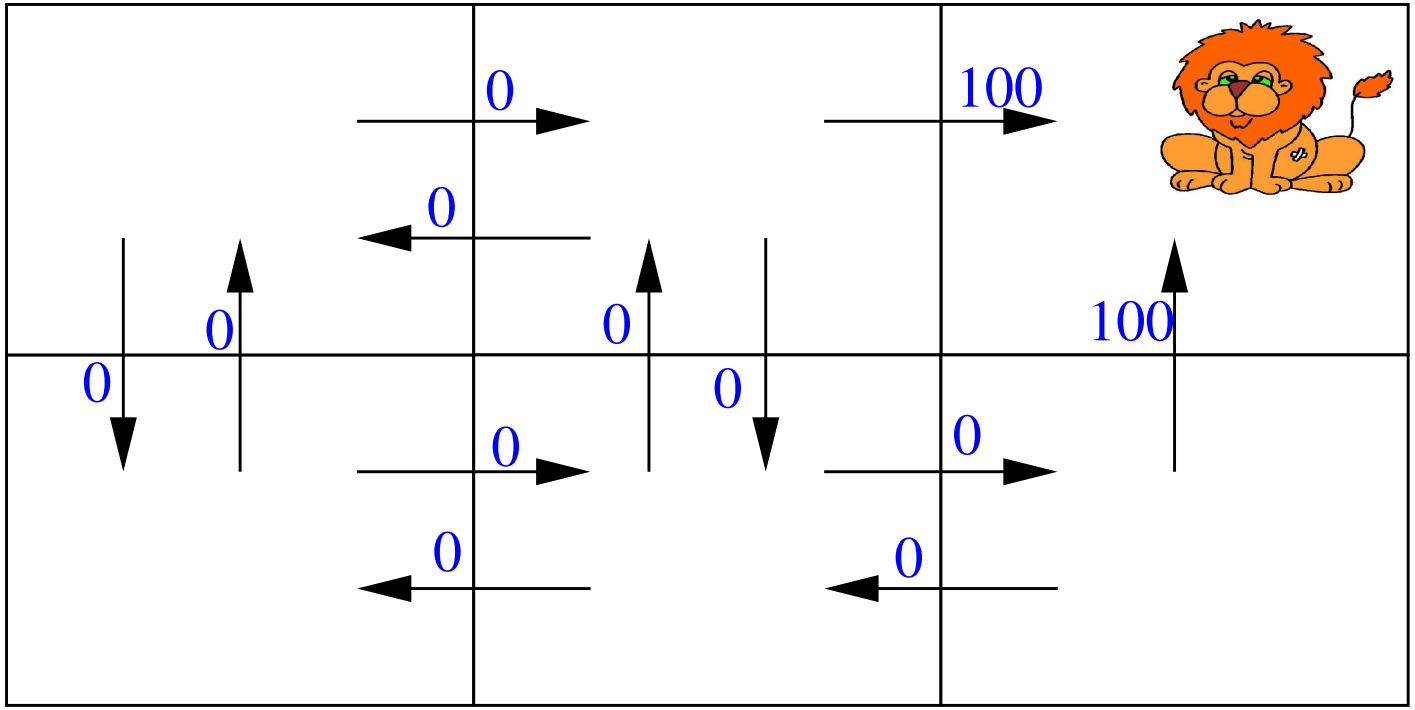
The lion starts in the top left corner.

Q Learning: trace (6/)



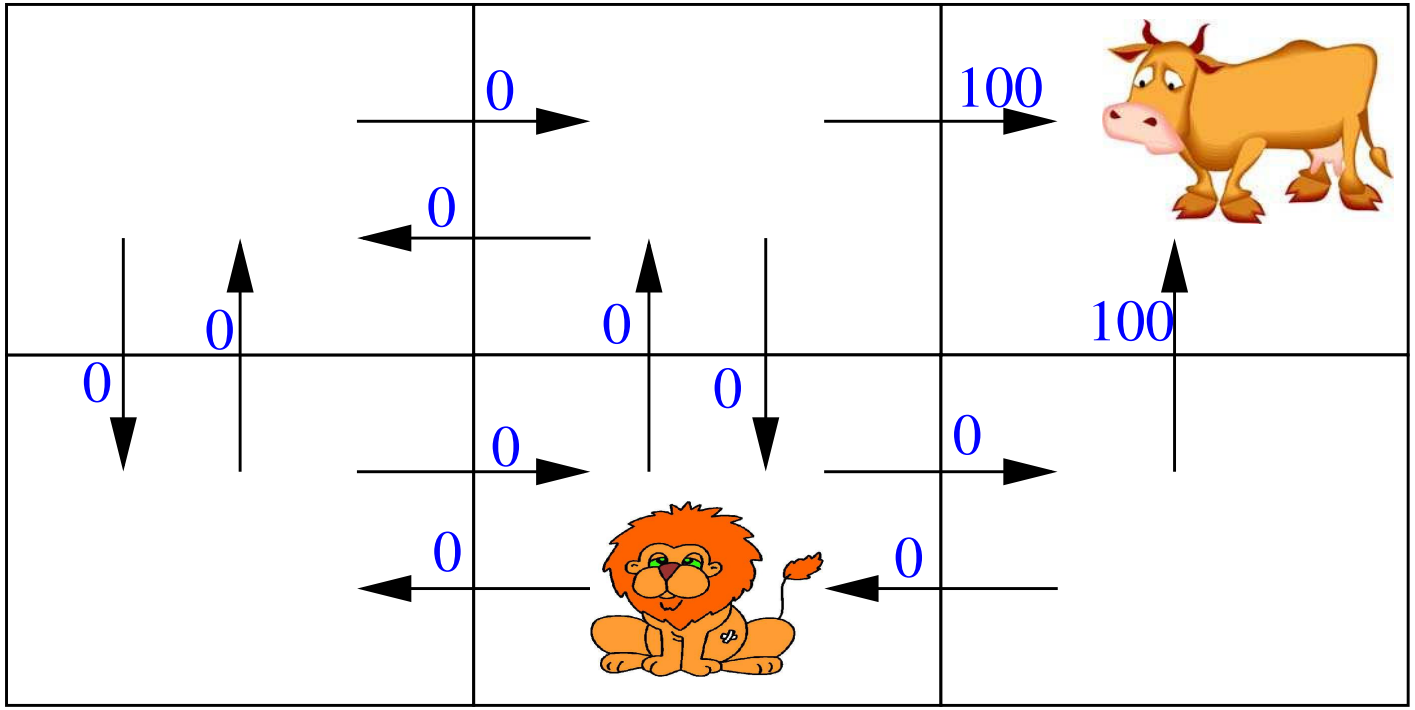
The lion moves to the right.

Q Learning: trace (7/)



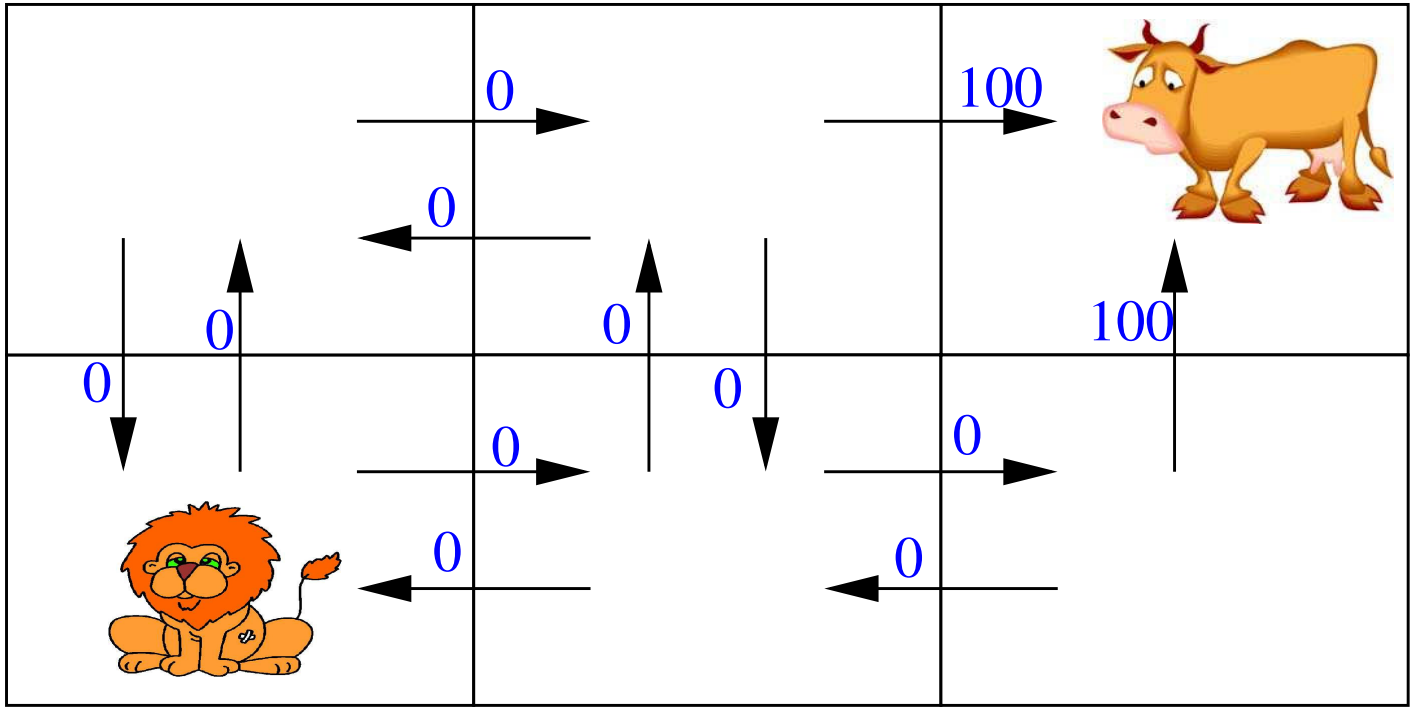
The lion goes right. It's still hungry!

Q Learning: trace (8/)



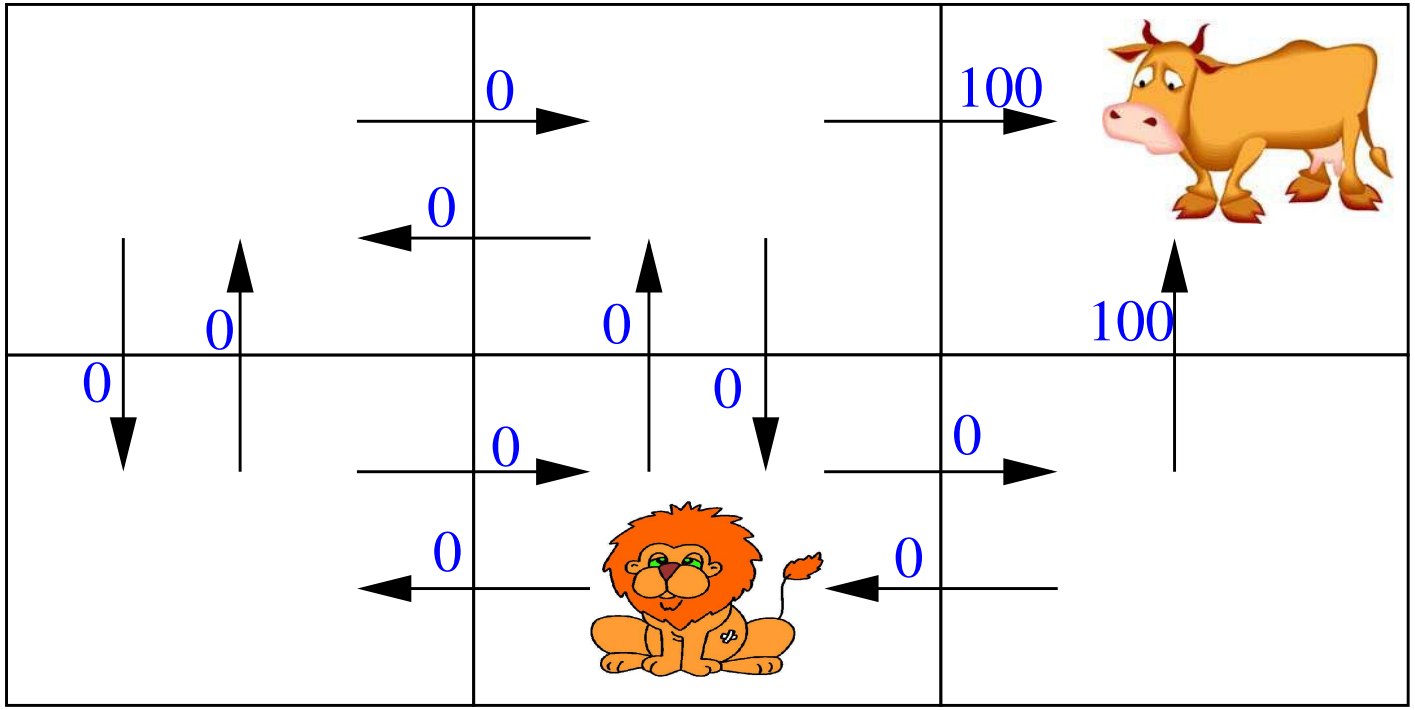
The lion starts in the bottom middle square.

Q Learning: trace (9/)



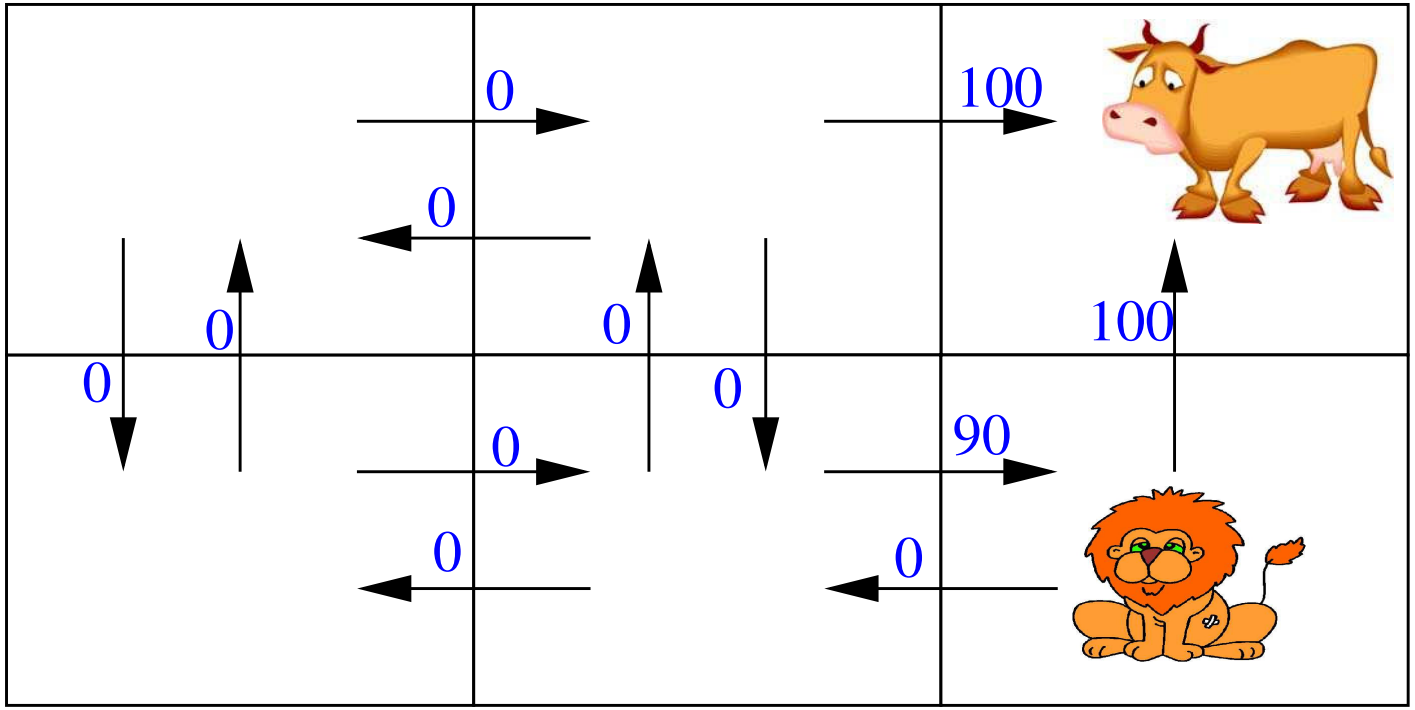
The lion goes left.

Q Learning: trace (10/)



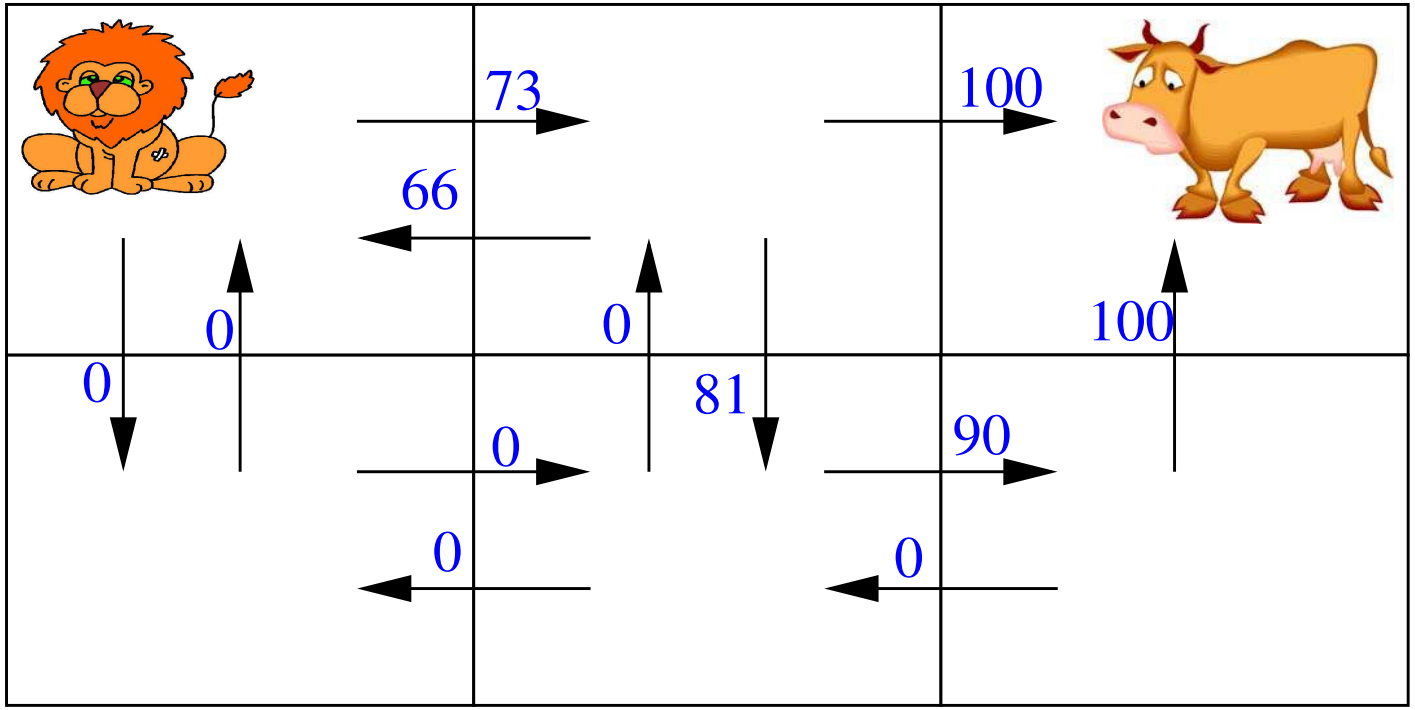
The lion goes right.

Q Learning: trace (11/)



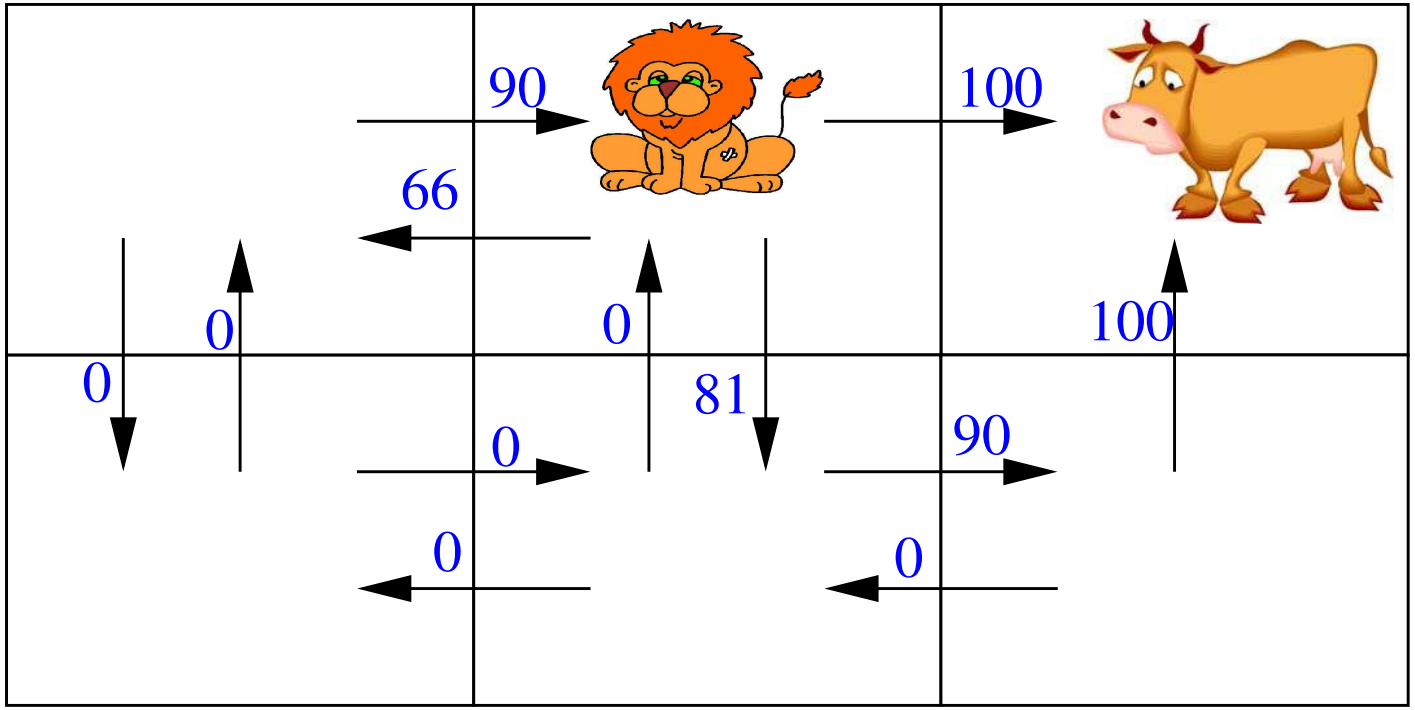
The lion goes right again.

Q Learning: trace (12/)



After a few episodes, this is the situation.

Q Learning: trace (13/13)



One extra move to the right.

The beauty of Q

$\hat{Q}$ converges to Q if:

- the MDP is deterministic (also works for stochastic MDPs with some changes)
- rewards have an upper bound
- *agent visits every state-action transition infinitely often*

In practice, the final point isn't necessary for Q learning to work.

Exploration strategies

How should the learning agent (e.g. lion) choose what action to perform next during learning ?

- choose seemingly optimal action (using $\hat{Q}$): may get stuck at local optimum
- Boltzmann exploration: $P(a_i|s) = \frac{k^{\hat{Q}(s,a)}}{\sum_{a_j} k^{\hat{Q}(s,a_j)}}$
- value of k determines how much the agent bases its decision on current $\hat{Q}$ estimate

Optimisation

How can the convergence speed of Q learning be improved?

- reordering the updates
- storing past state-action transitions together with observed rewards and retraining on them periodically
- still an open research area!

Generalisation

- Q learning needs to store values for all state-action pairs
- this couls be an astronomically large number!
- better: use function approximation of $\hat{Q}$
- example: use a neural network for storing the $\hat{Q}$ estimates and update network weights after each state-action transition
- similar to the approach taken by TD-Gammon

RL: advantages

- beatifully elegant formulation
- convergence guarantee for Q learning
- natural setting for reactive agents

RL: disadvantages

- only applicable to reactive agents (almost)
- often tricky to design the reward function
- theoretically guaranteed convergence of Q-learning can be too slow in practice
- state-space explosion with more agents
- learning result is non-transparent in general