Reinforcement Learning: A User's Guide

Bill Smart

Department of Computer Science and Engineering
Washington University in St. Louis
wds@cse.wustl.edu
http://www.cse.wustl.edu/~wds/

The Goal of this Tutorial

Provide answers to the following questions

- What is this thing called Reinforcement Learning?
- Why should I care about it?
- How does it work?
- What sort of problems can it solve?
- How is it being used?
- How is it being used in Autonomic Computing?
- Is it any use for my problems?
- Where can I find out more?

Find out what problems you are working on and see if RL can be applied to them

Overall Outline

Four parts

1. Basic reinforcement learning
2. Advanced reinforcement learning
3. Reinforcement learning in Autonomic Computing
4. Final Thoughts and Other Resources

Some Symbols

 Open Problem

 Glossing Over Details

 No Well-Understood Solution

 “Impossible” Problem

Part I:
Basic Reinforcement Learning

Outline for Part I

1. Basic intuitions about RL
2. Mathematics of RL
3. Learning value functions
4. Learning policies directly
5. Trade-offs
6. Example applications

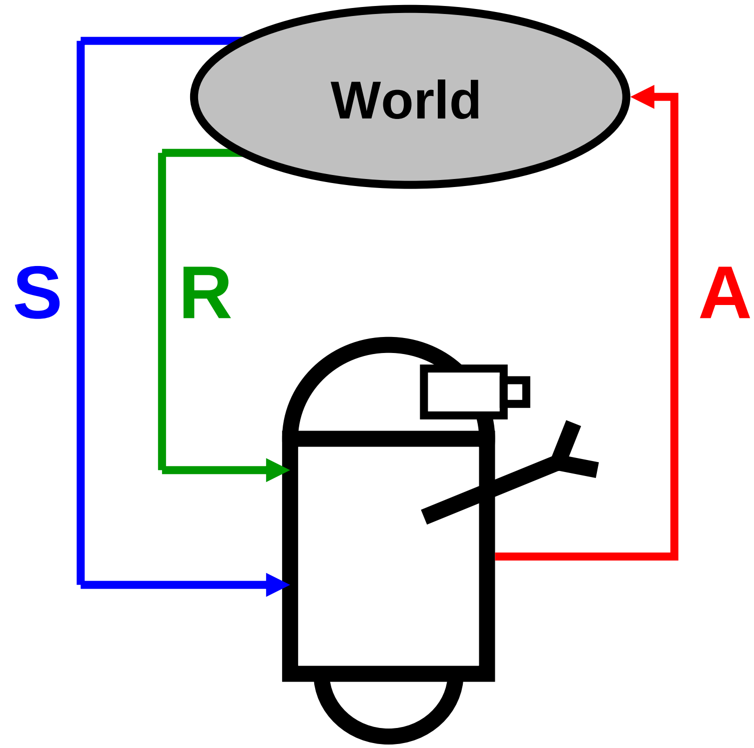
What is RL?

"a way of programming agents by reward and punishment without needing to specify *how* the task is to be achieved"

[Kaelbling, Littman, & Moore, 96]

Basic RL Model

1. Observe state, s_t
2. Decide on an action, a_t
3. Perform action
4. Observe new state, s_{t+1}
5. Observe reward, r_{t+1}
6. Learn from experience
7. Repeat

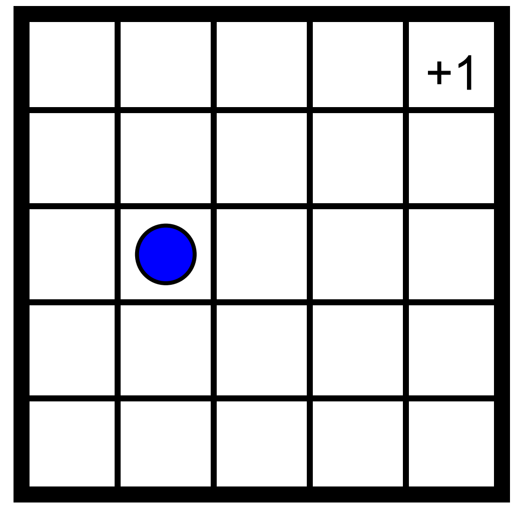


Goal: Find a control policy that will maximize the observed rewards over the lifetime of the agent

An Example: Gridworld

Canonical RL domain

- States are grid cells
- 4 actions: N, S, E, W
- Reward for entering top right cell
- -0.01 for every other move



Minimizing sum of rewards $\Rightarrow$ Shortest path

- In this instance

The Promise of Learning

The Promise of RL

Specify **what** to do, but not **how** to do it

- Through the reward function
- Learning "fills in the details"

Better final solutions

- Based of actual experiences, not programmer assumptions

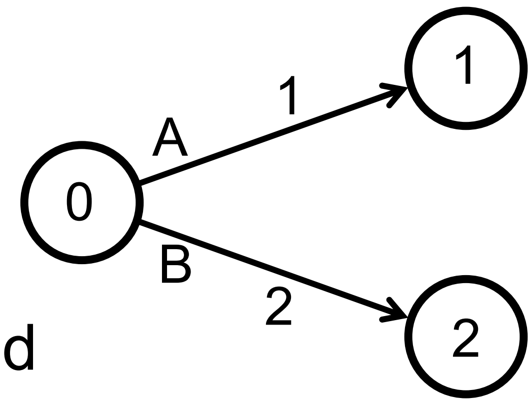
Less (human) time needed for a good solution

Mathematics of RL

Before we talk about RL, we need to cover some background material

- Some simple decision theory
- Markov Decision Processes
- Value functions
- Dynamic programming

Making Single Decisions



Single decision to be made
- Multiple discrete actions
- Each action has a reward associated with it

Goal is to maximize reward
- Not hard: just pick the action with the largest reward

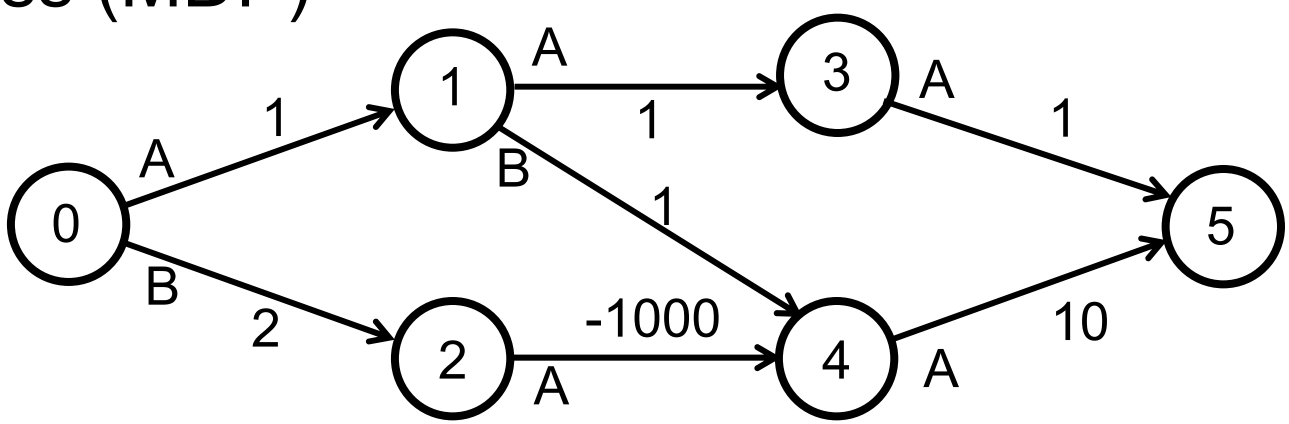
State 0 has a value of 2
- Sum of rewards from taking the best action from the state

Markov Decision Processes

We can generalize the previous example to multiple sequential decisions

- Each decision affects subsequent decisions

This is formally modeled by a Markov Decision Process (MDP)



Markov Decision Processes

Formally, an MDP is

[Puterman, 95]

- A set of states, $S = \{s_1, s_2, \ldots, s_n\}$
- A set of actions, $A = \{a_1, a_2, \ldots, a_m\}$
- A reward function, $R: S \times A \times S \rightarrow \Re$
- A transition function, $P_{ij}^{a} = P(s_{t+1} = j \mid s_t = i, a_t = a)$
  - Sometimes $T: S \times A \rightarrow S$

We want to learn a policy, $\pi: S \rightarrow A$

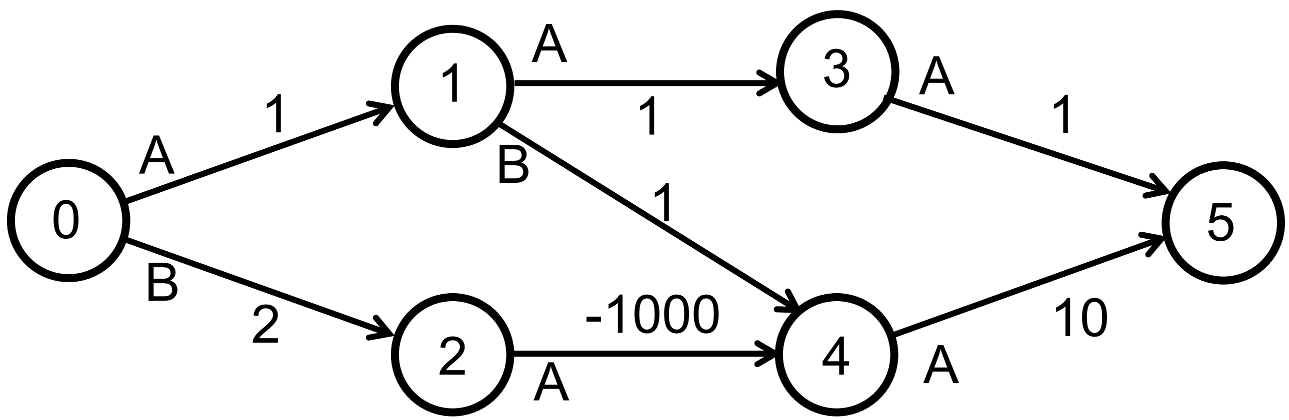
- Maximize sum of rewards we see over our lifetime

Policies

There are 3 policies for this MDP

1. 0 →1 →3 →5
2. 0 →1 →4 →5
3. 0 →2 →4 →5

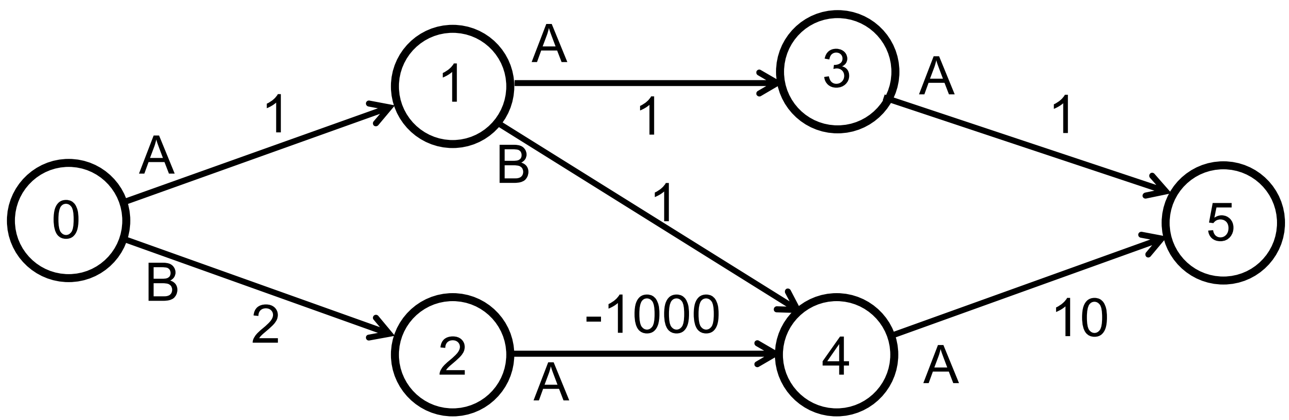
Which is the best one?



Comparing Policies

Order policies by how much reward they see

1. 0 →1 →3 →5 = 1 + 1 + 1 = 3
2. 0 →1 →4 →5 = 1 + 1 + 10 = 12
3. 0 →2 →4 →5 = 2 – 1000 + 10 = -988

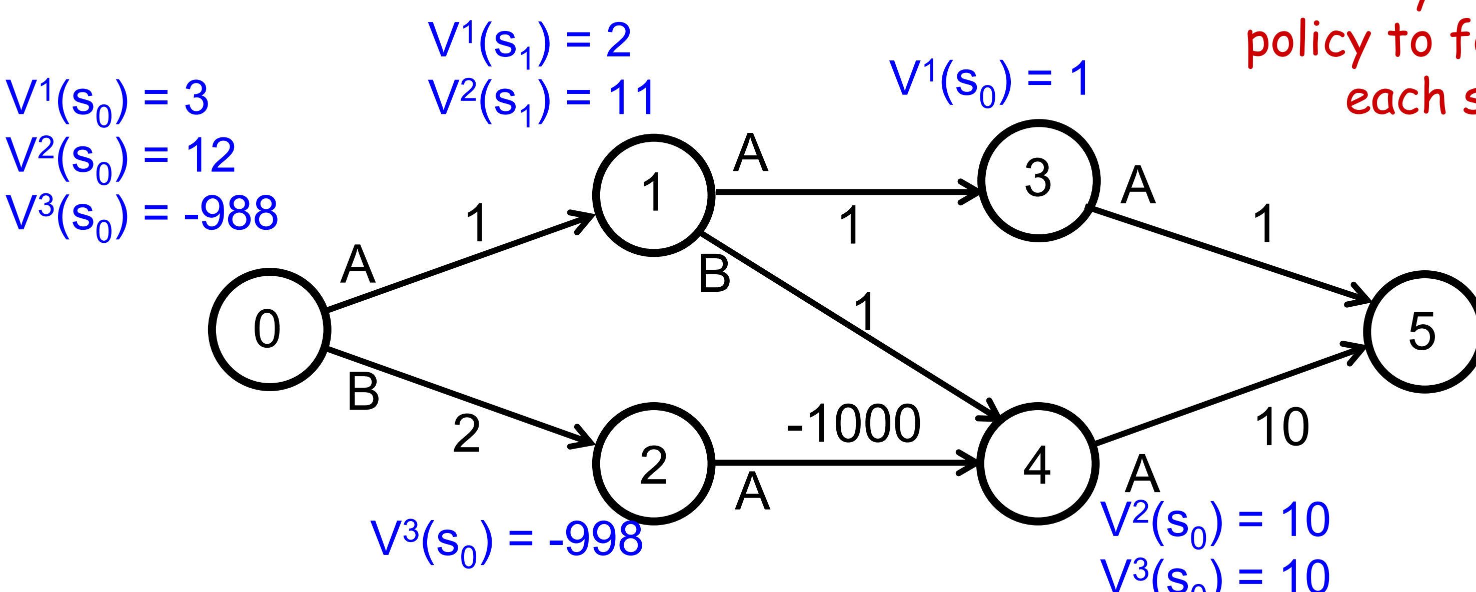


Value Functions

We can associate a value with each state

- For a fixed policy
- How good is it to run policy π from that state s
- This is the state value function, V

How do you tell which policy to follow from each state?

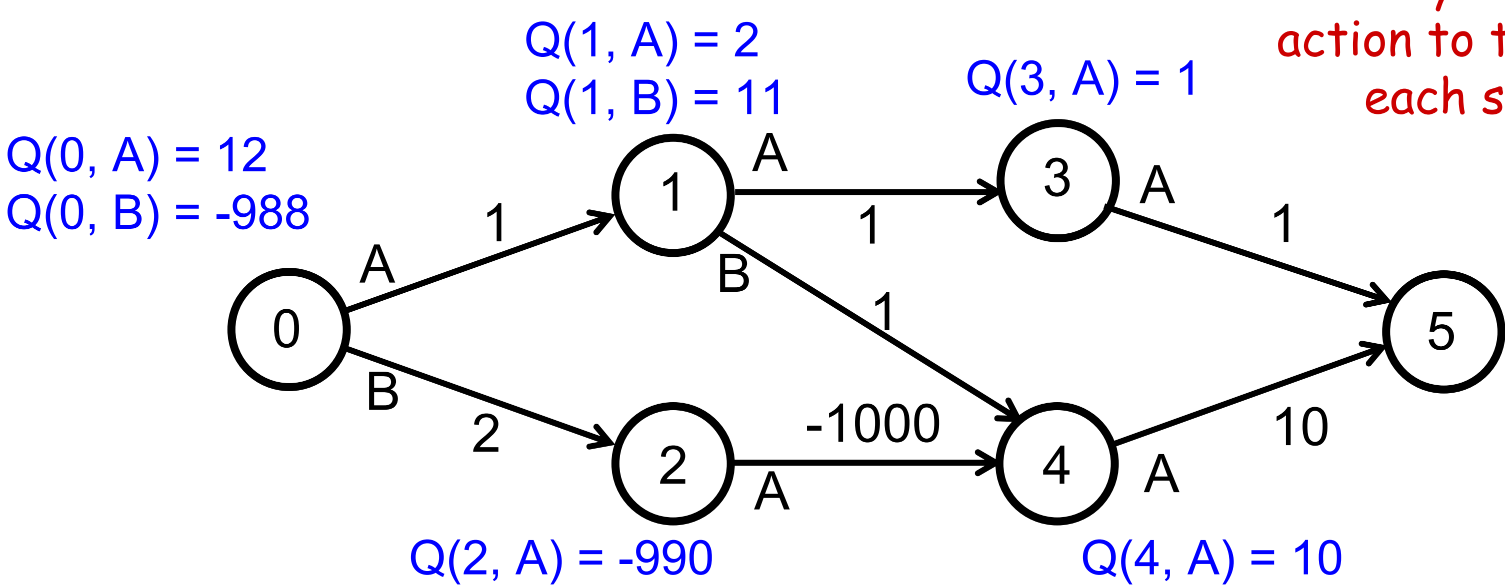


Value Functions

We can define value without specifying the policy

- Specify the value of taking action a from state s and then performing optimally
- This is the state-action value function, Q

How do you tell which action to take from each state?



Value Functions

So, we have two value functions

- $V^{\pi}(s) = R(s, \pi(s), s') + V^{\pi}(s')$
- $Q(s, a) = R(s, a, s') + \max_{a'} Q(s', a')$

s' is the next state

Both have the same form

- Next reward plus the best I can do from the next state

These extend to probabilistic actions

- $V^{\pi}(s) = \sum_{s'} P_{s,s'}^{\pi(s)} \left(R(s, \pi(s), s') + V^{\pi}(s') \right)$

- $Q(s,a) = \sum_{s'} P_{s,s'}^{a} \left(R(s, a, s') + \max_{a'} Q(s', a') \right)$

Getting the Policy

If we have the value function, then finding the best policy is easy

- $\pi(s) = \arg\max_a (R(s, a, s') + V^\pi(s'))$
- $\pi(s) = \arg\max_a Q(s, a)$

This generalizes to non-deterministic worlds

- Use expectations

Getting the Policy

We're looking for the optimal policy, $\pi^*(s)$

- No policy generates more reward than π^*

Optimal policy defines optimal value functions

- $V^*(s) = R(s, \pi(s), s') + V^*(s')$
- $Q^*(s,a) = R(s, a, s') + \mathrm{argmax}_{a'} Q^*(s', a')$

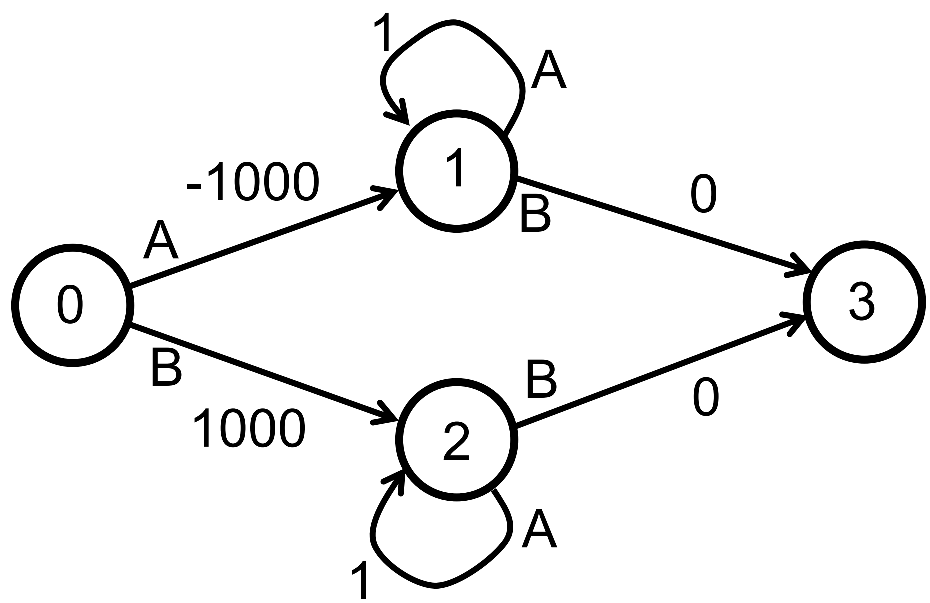
The easiest way to learn the optimal policy is to learn the optimal value function first

Problems with Our Functions

Consider this MDP

- Number of steps is now unlimited because of loops
- Value of states 1 and 2 is infinite for some policies

$$\begin{aligned} Q(1, A) &= 1 + Q(1, A) \\ &= 1 + 1 + Q(1, A) \\ &= 1 + 1 + 1 + Q(1, A) \\ &= \ldots \end{aligned}$$



This is bad

- All policies with a non-zero reward cycle have infinite value

Better Value Functions

We can introduce a term into the value function to get around the problem of infinite value

- Called the discount factor, γ
- Three interpretations
  - Probability of living to see the next time step
  - Measure of the uncertainty inherent in the world
  - Makes the mathematics work out nicely
- $0 \leq \gamma \leq 1$

We'll use these from now on

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

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

Better Value Functions

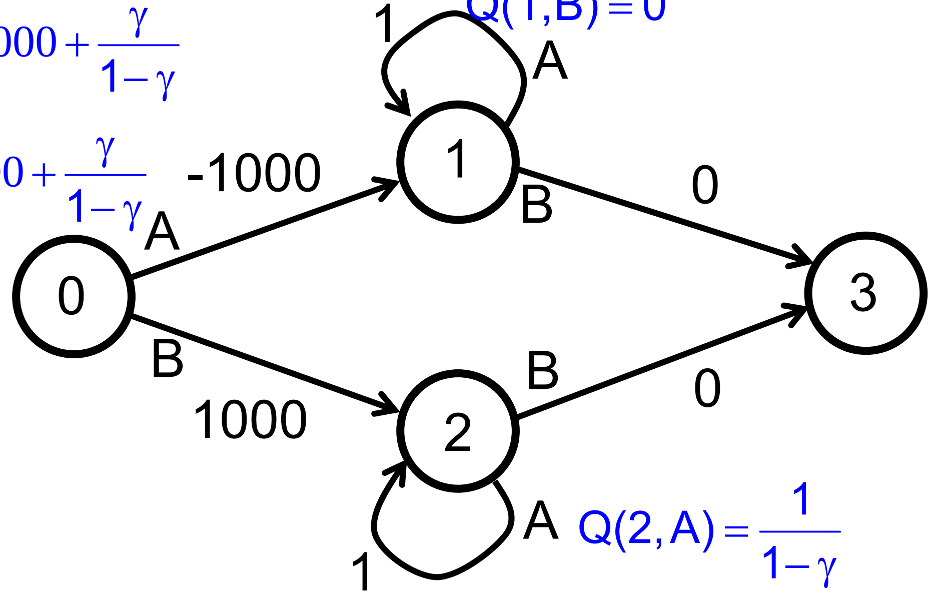
Value now depends on the discount, γ

$Q(0,A) = -1000 + \frac{\gamma}{1-\gamma}$

$Q(1,B) = 1000 + \frac{\gamma}{1-\gamma}$

$Q(1,A) = \frac{1}{1-\gamma}$

$Q(1,B) = 0$



$Q(2,A) = \frac{1}{1-\gamma}$

$Q(2,B) = 0$

Optimal Policy:

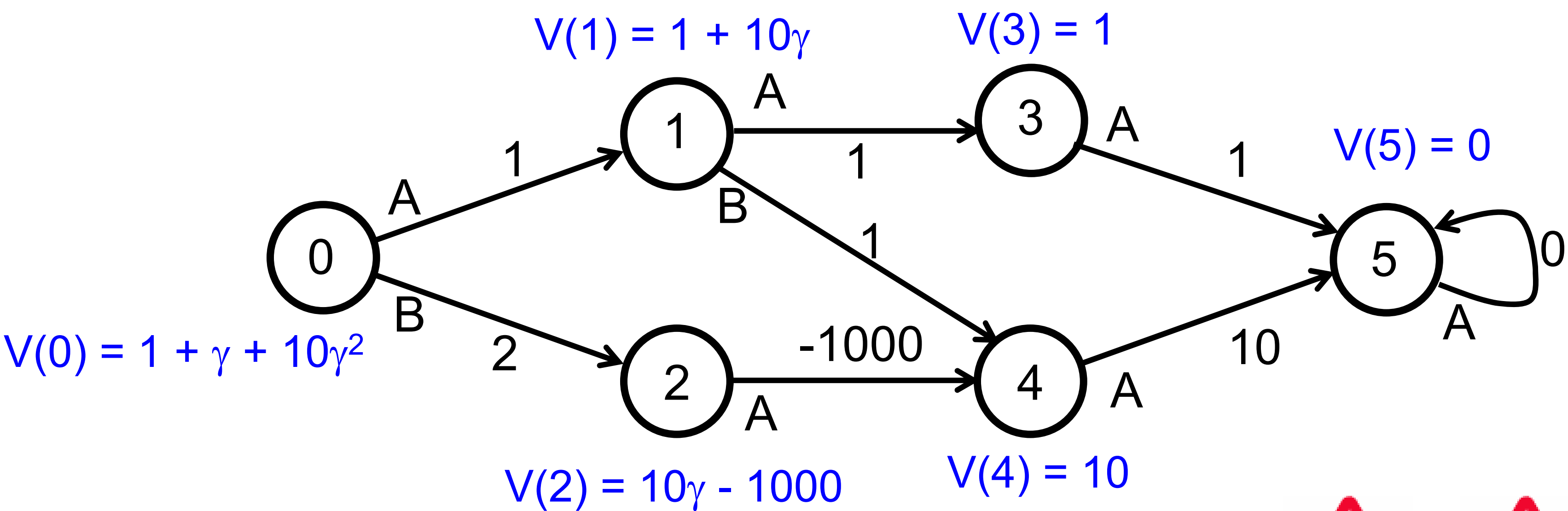
$\pi(0) = B$

$\pi(1) = A$

$\pi(2) = A$

Dynamic Programming

Given the complete MDP model, we can compute the optimal value function directly



[Bertsekas, 87, 95a, 95b]

Reinforcement Learning

What happens if we don't have the whole MDP?

- We know the states and actions
- We don't have the system model (transition function) or reward function

We're only allowed to sample from the MDP

- Can observe experiences (s, a, r, s')
- Need to perform actions to generate new experiences

This is Reinforcement Learning (RL)

- Sometimes called Approximate Dynamic Programming (ADP)

Learning Value Functions

We still want to learn a value function

- We're forced to approximate it iteratively
- Based on direct experience of the world

Four main algorithms

- Certainty equivalence
- Temporal Difference (TD) learning
- Q-learning
- SARSA

Certainty Equivalence

Collect experience by moving through the world

- $s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, a_3, r_4, s_4, a_4, r_5, s_5, \ldots$

Use these to estimate the underlying MDP

- Transition function, T: $S \times A \rightarrow S$
- Reward function, R: $S \times A \times S \rightarrow \Re$

Compute the optimal value function for this MDP

- And then compute the optimal policy from it

Temporal Difference (TD)

[Sutton, 88]

TD-learning estimates the value function directly

- Don't try to learn the underlying MDP

Keep an estimate of $V^{\pi}(s)$ in a table

- Update these estimates as we gather more experience
- Estimates depend on exploration policy, π
- TD is an on-policy method

TD-Learning Algorithm

1. Initialize $V^{\pi}(s)$ to 0, $\forall s$
2. Observe state, s
3. Perform action, $\pi(s)$
4. Observe new state, s', and reward, r
5. $V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha(r + \gamma V^{\pi}(s'))$
6. Go to 2

$0 \leq \alpha \leq 1$ is the learning rate

- How much attention do we pay to new experiences

TD-Learning

$V^{\pi}(s)$ is guaranteed to converge to $V^{*}(s)$

- After an infinite number of experiences
- If we decay the learning rate

$$\sum_{t=0}^{\infty} \alpha_t = \infty \qquad \sum_{t=0}^{\infty} \alpha_t^{\,2} < \infty$$

- $\alpha_t = \dfrac{c}{c+t}$ will work

In practice, we often don't need value convergence

- Policy convergence generally happens sooner

Actor-Critic Methods

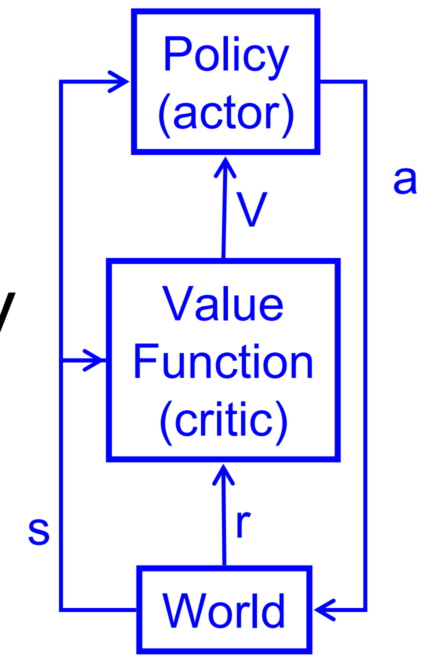
[Barto, Sutton, & Anderson, 83]

TD only evaluates a particular policy

- Does not learn a better policy

We can change the policy as we learn V

- Policy is the actor
- Value-function estimate is the critic



Success is generally dependent on the starting policy being "good enough"

Q-Learning

[Watkins & Dayan, 92]

Q-learning iteratively approximates the state-action value function, Q

- Again, we’re not going to estimate the MDP directly
- Learns the value function and policy simultaneously

Keep an estimate of Q(s, a) in a table

- Update these estimates as we gather more experience
- Estimates do not depend on exploration policy
- Q-learning is an off-policy method

Q-Learning Algorithm

1. Initialize $Q(s, a)$ to small random values, $\forall s, a$
2. Observe state, s
3. Pick an action, a, and do it
4. Observe next state, s', and reward, r
5. $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a'))$
6. Go to 2

$0 \leq \alpha \leq 1$ is the learning rate

- We need to decay this, just like TD

Picking Actions

We want to pick good actions most of the time, but also do some exploration

- Exploring means that we can learn better policies
- But, we want to balance known good actions with exploratory ones
- This is called the exploration/exploitation problem 

Picking Actions

ε-greedy

- Pick best (greedy) action with probability ε
- Otherwise, pick a random action

Boltzmann (Soft-Max)

- Pick an action based on its Q-value
- $P(a \mid s) = \dfrac{e^{\left(\frac{Q(s,a)}{\tau}\right)}}{\sum_{a'} e^{\left(\frac{Q(s,a')}{\tau}\right)}}$, where τ is the “temperature”

SARSA

SARSA iteratively approximates the state-action value function, Q

- Like Q-learning, SARSA learns the policy and the value function simultaneously

Keep an estimate of Q(s, a) in a table

- Update these estimates based on experiences
- Estimates depend on the exploration policy
- SARSA is an on-policy method
- Policy is derived from current value estimates

SARSA Algorithm

1. Initialize $Q(s, a)$ to small random values, $\forall s, a$
2. Observe state, s
3. Pick an action, a, and do it (just like Q-learning)
4. Observe next state, s', and reward, r
5. $Q(s, a) \leftarrow (1-\alpha)Q(s, a) + \alpha(r + \gamma Q(s', \pi(s')))$
6. Go to 2

$0 \leq \alpha \leq 1$ is the learning rate

- We need to decay this, just like TD

On-Policy vs. Off Policy

On-policy algorithms

- Final policy is influenced by the exploration policy
- Generally, the exploration policy needs to be "close" to the final policy
- Can get stuck in local maxima

Off-policy algorithms

Given enough experience

- Final policy is independent of exploration policy
- Can use arbitrary exploration policies
- Will not get stuck in local maxima

Convergence Guarantees

The convergence guarantees for RL are “in the limit”

- The word “infinite” crops up several times

Don’t let this put you off

- Value convergence is different than policy convergence
- We’re more interested in policy convergence
- If one action is really better than the others, policy convergence will happen relatively quickly

Rewards

Rewards measure how well the policy is doing

- Often correspond to events in the world
  - Current load on a machine
  - Reaching the coffee machine
  - Program crashing
- Everything else gets a 0 reward

These are sparse rewards

Things work better if the rewards are incremental

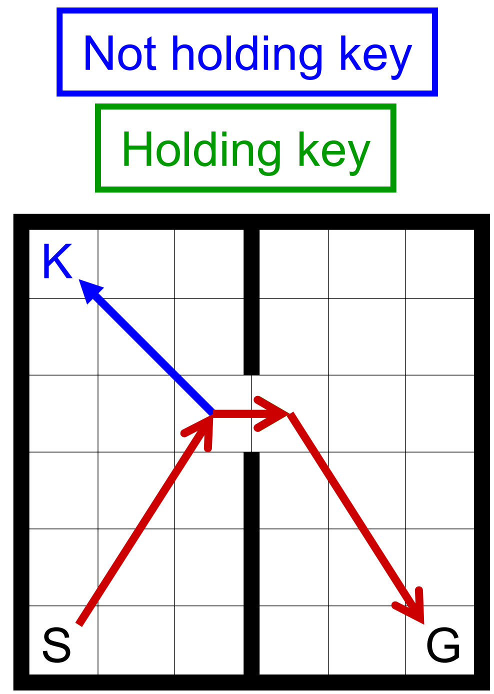
- For example, distance to goal at each step
- These reward functions are often hard to design

These are dense rewards

The Markov Property

RL needs a set of states that are Markov

- Everything you need to know to make a decision is included in the state
- Not allowed to consult the past



Rule-of-thumb

- If you can calculate the reward function from the state without any additional information, you're OK

But, What’s the Catch?

RL will solve all of your problems, but

- We need lots of experience to train from
- Taking random actions can be dangerous
- It can take a long time to learn
- Not all problems fit into the MDP framework

Learning Policies Directly

An alternative approach to RL is to reward whole policies, rather than individual actions

- Run whole policy, then receive a single reward
- Reward measures success of the whole policy

If there are a small number of policies, we can exhaustively try them all

- However, this is not possible in most interesting problems

Policy Gradient Methods

Assume that our policy, p, has a set of n real-valued parameters, q = $\{q_1, q_2, q_3, \ldots, q_n\}$

- Running the policy with a particular q results in a reward, r_q
- Estimate the reward gradient, $\frac{\partial R}{\partial \theta_i}$, for each q_i
- $\theta_i \leftarrow \theta_i + \alpha \frac{\partial R}{\partial \theta_i}$

This is another learning rate

Policy Gradient Methods

This results in hill-climbing in policy space

- So, it's subject to all the problems of hill-climbing
- But, we can also use tricks from search, like random restarts and momentum terms

This is a good approach if you have a parameterized policy

- Typically faster than value-based methods
- "Safe" exploration, if you have a good policy
- Learns locally-best parameters *for that policy*

An Example: Learning to Walk

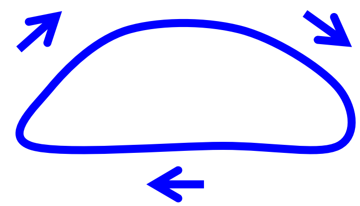
[Kohl & Stone, 04]

RoboCup legged league

- Walking quickly is a *big* advantage

Robots have a parameterized gait controller

- 11 parameters
- Controls step length, height, etc.



Robots walk across soccer pitch and are timed

- Reward is a function of the time taken

An Example: Learning to Walk

Basic idea

1. Pick an initial $\theta = \{\theta_1, \theta_2, \ldots, \theta_{11}\}$
2. Generate N testing parameter settings by perturbing θ

   $\theta^j = \{\theta_1 + \delta_1, \theta_2 + \delta_2, \ldots, \theta_{11} + \delta_{11}\}, \quad \delta_i \in \{-\varepsilon, 0, \varepsilon\}$
3. Test each setting, and observe rewards

   $\theta^j \rightarrow r_j$
4. For each $\theta_i \in \theta$

   Calculate $\theta_1^+, \theta_1^0, \theta_1^-$ and set $\theta'_i \leftarrow \theta_i + \begin{cases} \delta & \text{if } \theta_i^+ \text{ largest} \\ 0 & \text{if } \theta_i^0 \text{ largest} \\ -\delta & \text{if } \theta_i^- \text{ largest} \end{cases}$
5. Set $\theta \leftarrow \theta'$, and go to 2

Average reward when $q^n_i = q_i - d_i$

An Example: Learning to Walk

Initial

Final

Video: Nate Kohl & Peter Stone, UT Austin

Value Function or Policy Gradient?

When should I use policy gradient?

- When there’s a parameterized policy
- When there’s a high-dimensional state space
- When we expect the gradient to be smooth

When should I use a value-based method?

- When there is no parameterized policy
- When we have no idea how to solve the problem

Summary for Part I

Background

- MDPs, and how to solve them
- Solving MDPs with dynamic programming
- How RL is different from DP

Algorithms

- Certainty equivalence
- TD
- Q-learning
- SARSA
- Policy gradient

Part II:

Advanced Reinforcement Learning

Outline for Part II

1. Continuous state spaces
2. Continuous actions
3. All the stuff we’re not going to talk about

Continuous State Spaces

Many problems have a continuous, multi-dimensional state space

- Position in the world, for example
- But, standard RL algorithms only deal with discrete state spaces

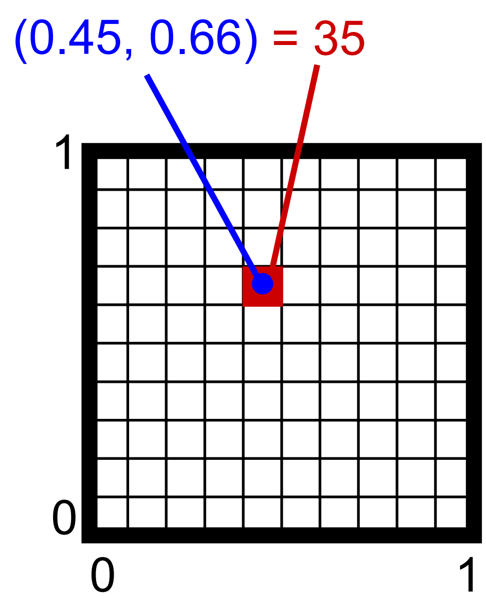
How can we modify the standard algorithms to deal with continuous state spaces?

- Discretization
- Value-function approximation

State Space Discretization

The simplest way to deal with continuous state spaces is to chop them up into discrete ones

- Uniformly discretize each dimension
- Every real point maps to a discrete state



If we know something about the problem, we can often make a more informed discretization

- This is likely to work better

State Space Discretization

Problems

- The Curse of Dimensionality
  - Exponentially many states
  - d^n states for n dimensions, with d partitions per dimension
- Introduces hidden state
- Removes Markov property

Works in practice for some (small) problems

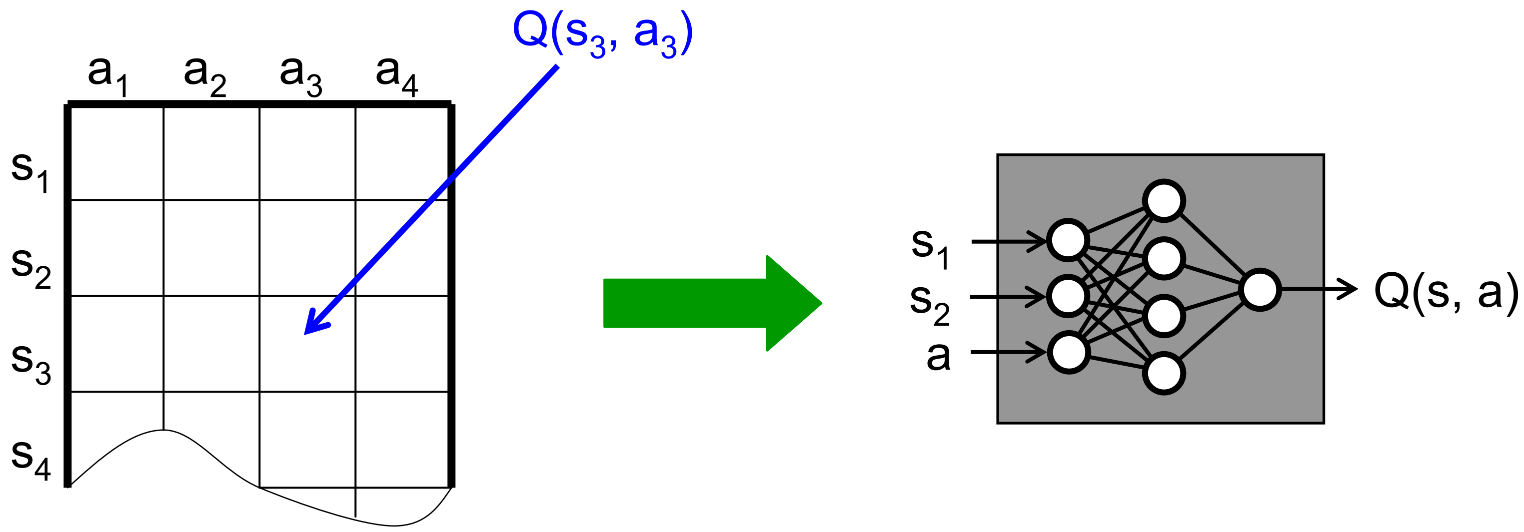
Better Discretization

We can be more clever about how we discretize the world

- Pay attention to the system dynamics
- Only use a fine discretization where it matters
- There are a number of ways to do this, based on samples
- But, then might still introduce hidden state

Value-Function Approximation

Another way to deal with continuous state is to replace the tabular value function representation with a general-purpose function approximator



Value-Function Approximation

VFA is good

- Deals naturally with continuous, multi-dimensional states
- Generalizes between states
  - Don’t have to see every state
  - Should result in faster learning
- Plenty of function approximators to choose from
  - Pick your favorite, artificial neural networks are popular

VFA is bad

- It doesn’t work

Value-Function Approximation

OK, it sometimes works 

- Several successful examples (see later)
- But also many failures, often in simple examples

[Boyan & Moore, 95]

Why does it often not work? 

- Convergence guarantees go away
- Small errors in approximation tend to “snowball”
  - Recall, we’re often taking the maximum of several values
- Euclidean distance metric is not always appropriate
  - Leads to incorrect value generalization

Warning: Author Speculation

Warning: Author Speculation

Continuous Actions

Some problems naturally have continuous actions

- Controlling a steering wheel, for example

In the standard algorithms, we maximize over a discrete set of actions

$$\pi(s) = \arg\max_a (R(s, a, s') + V^{\pi}(s'))$$

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

In the continuous case, this becomes a general optimization

Continuous Actions

If we have one continuous action, we can do a 1d optimization

- For a given state, treat value as a function of the (continuous) action, f(a)
- Use standard techniques to find the maximum
- This is *much* more expensive than maximizing over a discrete set, and might not find the true maximum
- We do this maximization a lot while learning

Continuous Actions

Things only get worse if we have multi-dimensional actions 

- Multi-dimensional sample-based optimization is hard

Biggest understatement of the tutorial

Usual solution is to discretize action space

- Might still suffer from the Curse of Dimensionality
- Again, knowledge of the problem domain can really help here

All the Other Stuff

We don't have time to talk about the other advanced techniques, but here are some buzzwords

- Continuous time/varying length values
- Hierarchical state spaces
- Partial observability
- Acceleration techniques

All of these are covered in the Sutton and Barto book

Summary for Part II

Extensions to the basic algorithms

- Continuous state spaces
- Continuous action spaces
- A set of buzzwords

Part III:

RL in Autonomic Compting

Outline for Part III

Some example applications

- Elevator scheduling
- Cell phone channel allocation
- Network packet routing

Audience participation time

- What to *you* want to use RL for?

Elevator Scheduling

[Crites & Barto, 95, 98]

Uses RL to learn controllers for a bank of elevators

- 4 elevators
- 10 floors
- Each elevator controlled independently

Simulation of one hour of “down-peak” traffic

- Most traffic heading to lobby
- 0% to 10% of traffic is inter-floor
- Realistic simulation

States

Continuous state space

- Includes elapsed times
- Could discretize it to 10^{22} states

State space is carefully crafted

- Builds in knowledge of the problem
- Designed to work well with VFA scheme

States

46 dimensions

• Hall button pushed?	9 binary
• Hall button elapsed time	9 real
• Car location/direction	16 binary
• Other car locations	10 binary
• Highest floor with waiting passenger	1 binary
• A floor of longest waiting passenger	1 binary

Actions

Discrete action space

- If stopped: “move up”, “move down”
- If moving: “stop at next floor”, “continue past next floor”

Additional constraints enforced

- Based on a knowledge of the problem
- Only two actions in final system: “stop”, “continue”

Actions selected with a Boltzmann distribution

Rewards

Different minimization objectives

- Wait time
- System time (wait + travel)
- %age of passengers waiting more than 60 seconds
- Sum of squared wait times

Different amounts of knowledge

- Omniscient: Reward calculated from simulator state
- Online: Only use information available to real car
  - Must estimate everything else

Values

Simulation is a discrete event, continuous-time system

- Actions take different lengths of time to execute
- Standard $\sum_{t=0}^{\infty} \gamma^{t} r_t$ formulation won't work
- Use $\int_0^{\infty} e^{-\beta\tau} r_\tau \, d\tau$ instead
    - Parameter β controls decay rate (like γ)

Value-Function Approximation

Used an an artificial neural network

- 47 input units
- 20 hidden units
- 2 output units

980 free parameters (weights)

Trained with backpropagation

- Learning rate is 0.01 or 0.001
- This makes the network conservative

Results

Trained for 60,000 simulated hours

- 4 days of computer time in 1995

Performed well

- Better than commonly-used algorithm (SECTOR)

Down only	Average Wait	Squared Wait	System Time	% > 60 s
SECTOR	21.4	674	47.7	1.12
Best Fixed	15.1	338	46.6	0.11
RL (shared)	14.8	320	41.8	0.09
RL (indep)	**14.7**	**313**	**41.7**	**0.07**

Results

Up 2	Average Wait	Squared Wait	System Time	% > 60 s
SECTOR	27.3	1252	54.8	9.24
Best Fixed	17.9	476	48.9	**0.50**
RL (shared)	**16.9**	476	**42.7**	1.53
RL (indep)	**16.9**	**468**	**42.7**	1.40

Up 4	Average Wait	Squared Wait	System Time	% > 60 s
SECTOR	30.3	1643	59.5	13.5
Best Fixed	20.1	667	52.3	3.10
RL (shared)	18.8	593	**45.4**	**2.40**
RL (indep)	**18.6**	**585**	45.7	2.49

Thoughts

Elevator system is simulated

- We could run it for real, but it would take a long time
- Assumes a sufficiently realistic simulation

State space was the result of “considerable experimentation”

- Machine learning (and RL) is all about the right representation

Thoughts

A lot of domain knowledge was incorporated into the RL system

- Improves learning performance
- Makes the problem tractable
- It pays to have a domain expert

Continuous definition of value is “close enough”

- Not really the same as standard value
- But it behaves similarly
- Actual values are less important that their ordering

Thoughts

When doing VFA with artificial neural networks, low (backprop) learning rates seem to work best

- Network is conservative about updates
- Seems to avoid over-estimation of values

RL system outperformed fixed algorithms consistently

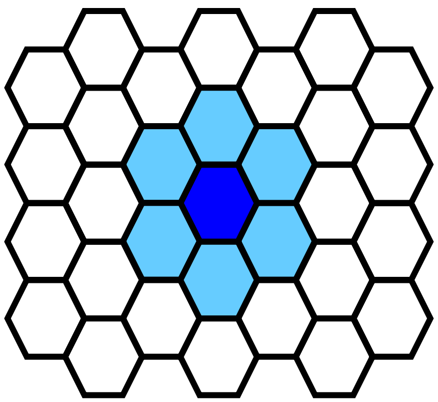
- So, why aren't the elevator companies using it?

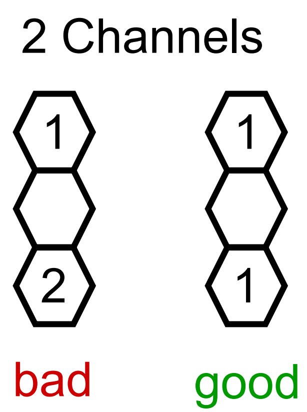
Cell Phone Channel Allocation

[Singh & Bertsekas, 97]

Learns channel allocations for cell phones

- Channels are limited
- Allocations affect adjacent cells
- Want to minimize dropped and blocked calls





States

State consists of two elements

- Occupied and unoccupied channels for each cell
  - Exponential in number of cells
- Last event (arrival, departure, handoff)

This is too large to use directly

- 70^{49} states for example in paper

States

State space actually used has two components

- Availability: Number of free channels in cell
- Packing: Number of times each channel is used within interference radius

Actions

Call arrival

- Evaluate possible next channels
- Assign one with highest value

Call termination

- Free channel
- Consider reassigning each ongoing call to just-released channel
- Perform reassignment (if any) with highest value

Rewards and Values

Reward is number of on-going calls

Again, this is a continuous-time system

- Value is $\int_0^\infty e^{-\beta t} c(t)\, dt$, where is the number of on-going calls at time t

Value-Function Approximation

The value function is represented by an artificial neural network

- Linear units
- Evaluates state and returns value
- Trained using the TD algorithm

Results

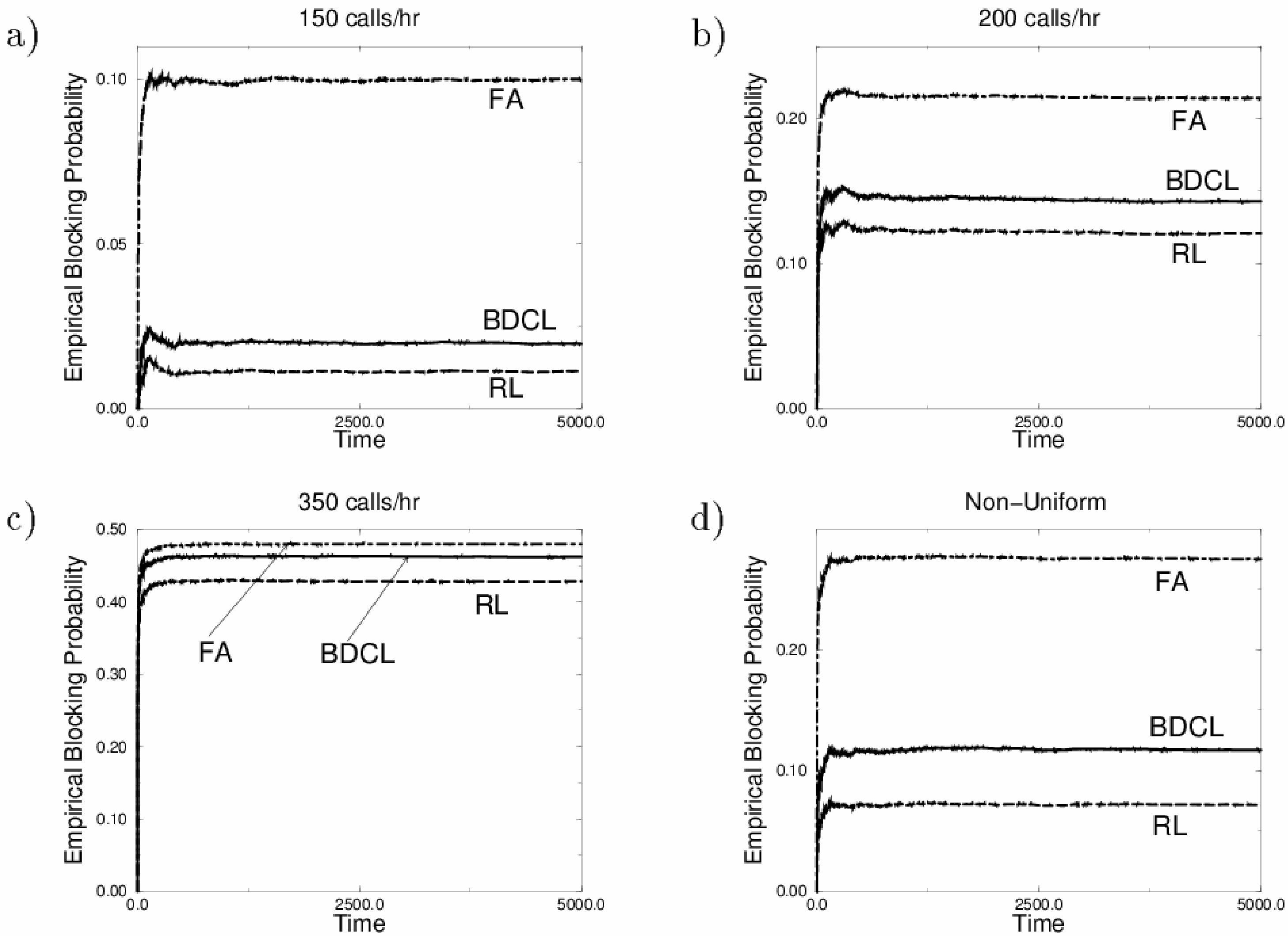
Compared to best fixed and adaptive algorithms from the literature

- FA: Fixed set of channels pre-computed and allocated to each cell.
- BDCL: Best adaptive algorithm from the literature

Tested at different call levels

- 150, 200, 350, variable calls/hour

Results



From: Singh & Bertsekas, 97

Thoughts

There is a discrete-state representation

- But, it’s too big to deal with

Lots of domain knowledge in the state vector

- Again, it’s all about the representation

State representation is relative for each cell

- Does not grow as number of cells increases

Thoughts

Each agent makes its own decisions

- Using the same learned policy
- Might be able to do better with explicit cooperation

Again, VFA took some tweaking to get right

- Different inputs for the neural network

Thoughts

Learning is (relatively) fast

- Best behavior after about 250 simulated minutes
- Learned behaviour is stable

Results are good

- Especially compared to currently deployed algorithm
- So, why don't the cell companies use an RL solution?

Network Packet Routing

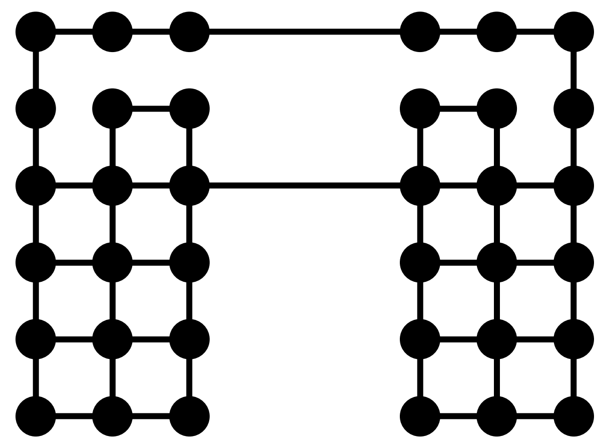
[Boyan & Littman, 94]

Uses Q-learning to route packets in a network

- Policy determines which adjacent node to send a packet to
- Learns a static routing policy

Each node has a queue

- One packet is dispatched on each time step



States and Actions

State is the destination of the current packet

Action is which adjacent node to route the packet through

Rewards and Values

Reward is the Q-value estimate of the state that the packet is routed to

Value function Q(d, y) is estimate of time needed to reach destination, d, for the current packet

- This is like setting $\gamma = 1$

Values

Values are represented in a table

- Since we have discrete states and actions

Tried VFA, but the results “proved inconclusive”

Results

Compared results to a standard shortest-path routing algorithm

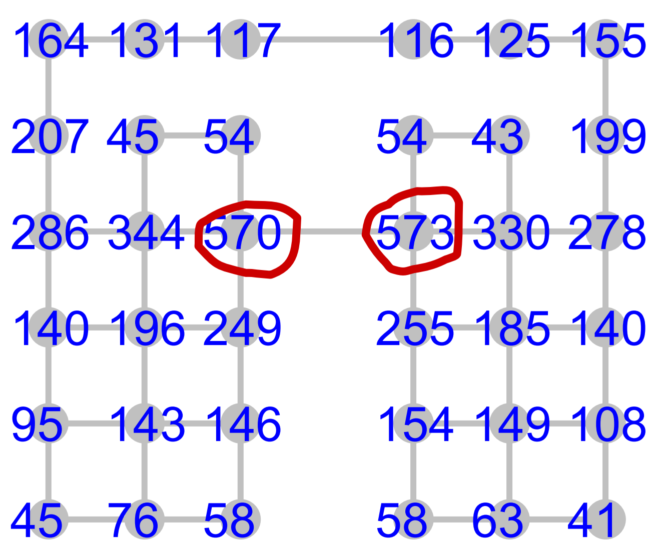
- Under several different load conditions

Q-routing was better in all cases

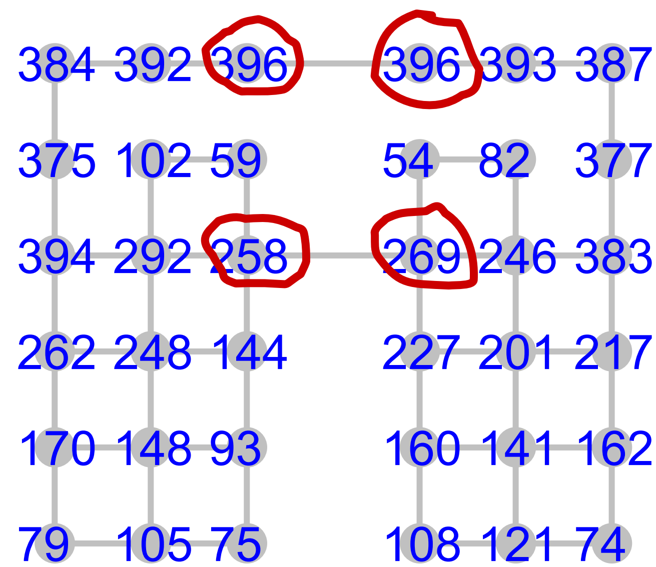
- Learned static routes that were more balanced across all of the nodes in the network
- Fewer “choke points” in the network

Results

High load conditions, number of routes passing through each node

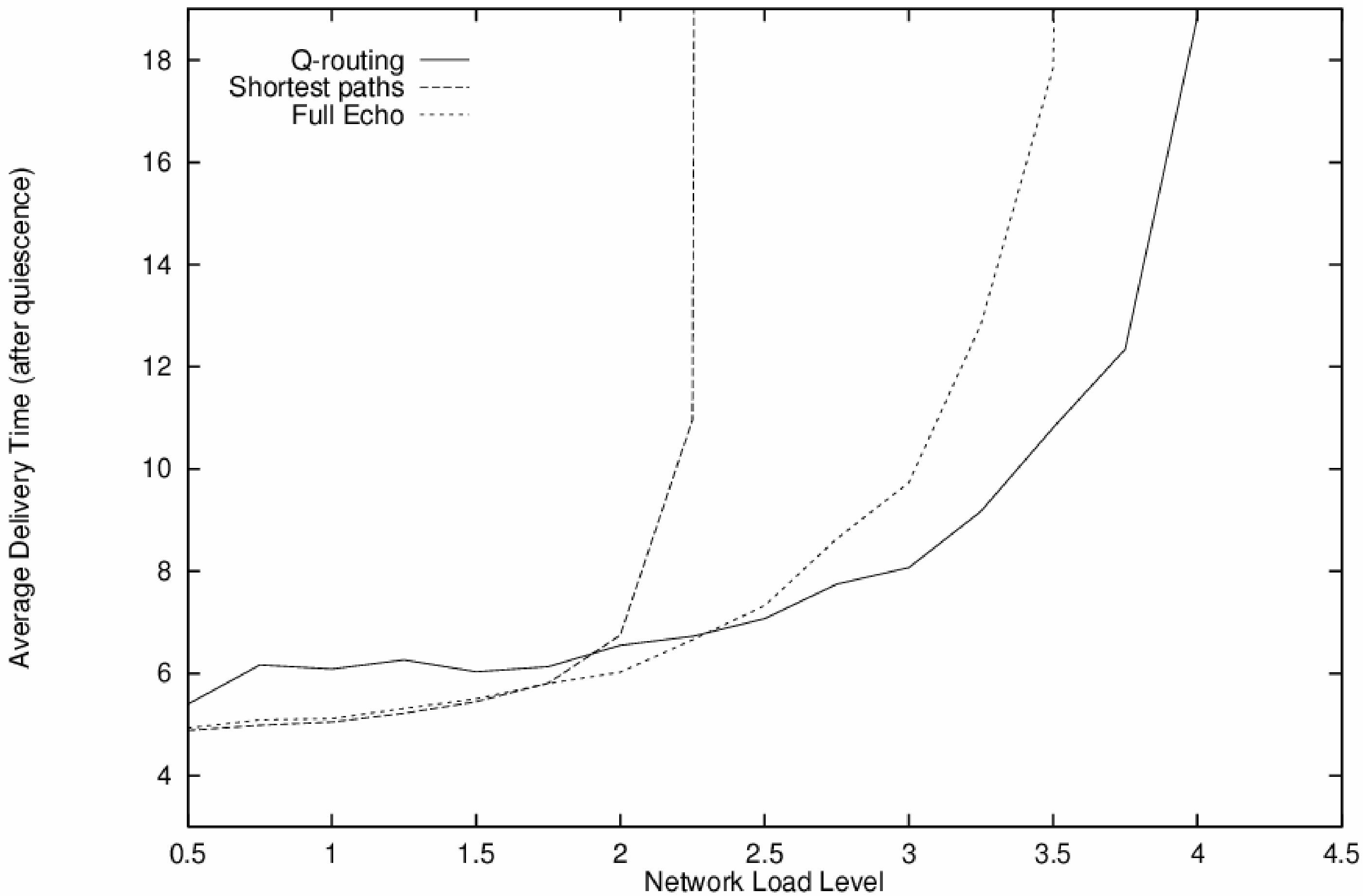


Shortest Path



Q-routing

Results



From: Boyan & Littman, 94

Thoughts

This application has a potentially unbounded Q-value, since it effectively sets $\gamma = 1$

- This is OK, since the Q-function is really a measure of cost
- All infinite-valued policies really are the same
- We're only interested in the smallest Q-value

Thoughts

VFA didn’t work, probably because there’s no notion of continuity between the states

- State number is nominal (has no order)
- If we renumbered the states, it wouldn’t affect the algorithm
- VFA comes with a built-in assumption about continuity between states (and their values)
- VFA generally fails at discontinuities in the value function (unless we’re careful)

Thoughts

Are you compelled by the results?

- Is shortest-path a reasonable comparison?
- How much is the network tailored to show Q-routing is better?
- These are important questions to ask in any RL application
  - RL researchers are often not experts in the application area
  - Straw men in ML papers are sometimes not the strongest that they could be

Audience Participation Time

In the acceptance letter for this tutorial was the following challenge:

- "You might even consider getting people to describe any applications they're working on currently that they think might benefit from RL, and you could pick a few of those and work through them on the fly – if you're willing to something that ~~risky~~." stupid

So, at the risk of falling flat on my face, does anyone have an application that RL might apply to?

Summary for Part III

RL has been successfully applied to a number of problems relevant to Autonomic Computing

- Elevator control
- Cell phone channel allocation
- Network packet routing

Some of you have (hopefully) got some ideas about how RL can be applied to your own applications

- Or there's just been 30 minutes of silence

Part IV:
Final Thoughts

Final Thoughts

RL seems well-suited to Autonomic Computing

- Techniques are starting to scale to deal with realistic problems
- RL papers are starting to appear in the AC literature

RL researchers are always looking for new, hard problems to work on

This includes me

- Especially if they're drawn from the "real world"
- Funding agencies are especially keen about this

Other RL Applications

We only talked about a few RL applications, but there are many other application areas

- Job shop scheduling
- Control of processes
  - Robots
  - Power systems
  - Bicycles
  - Sailboats
  - Helicopters

And, there are several papers at ICAC 2005

Standard References

The two standard references for RL are:

- “Reinforcement Learning: A Survey”, Leslie P. Kaelbling, Michael L. Littman, and Andrew W. Moore. Journal of Artificial Intelligence Research, 4:237-285, 1996.
- “Reinforcement Learning: An Introduction”, Richard S. Sutton and Andrew G. Barto. MIT Press, 1998.

Conferences

ICML
- International Conference on Machine Learning

NIPS
- Advances in Neural Information Processing Systems

AAAI
- National Conference on Artificial Intelligence

IJCAI
- International Joint Conference on Artificial Intelligence

IAAI
- Innovative Applications of Artificial Intelligence

Journals

Journal of Machine Learning Research

- http://www.jmlr.org/

Journal of Artificial Intelligence Research

- http://www.jair.org/

Machine Learning Journal

- Springer

Artificial Intelligence Journal

- Elsevier

Web Sites

Reinforcement Learning Repository

http://www-anw.cs.umass.edu/rlr/

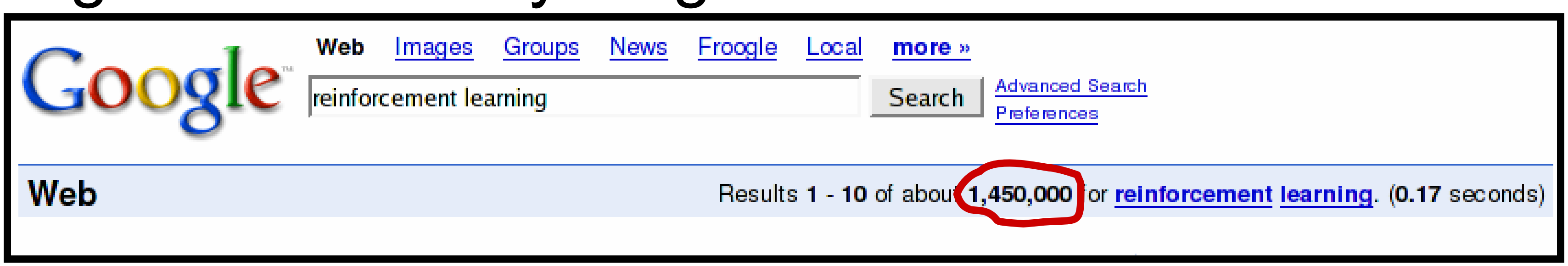
Rich Sutton's RL FAQ

http://www.cs.ualberta.ca/~sutton/RL-FAQ.html

Satinder Singh's RL wiki

http://neuromancer.eecs.umich.edu/cgi-bin/twiki/view/Main/

Google knows everything...



Questions?