Reinforcement Learning

Leslie Pack Kaelbling
Brown University

Michael L. Littman
Brown University and Bellcore

Andrew W. Moore
Carnegie Mellon University

Tutorial Structure

- Introduction to reinforcement learning (Leslie)
- Delayed reinforcement (Michael)
- Learning and using world models (Andrew)

Break

- Generalization and hierarchies (Leslie)
- Hidden state (Michael)
- Practical reinforcement learning (Andrew)

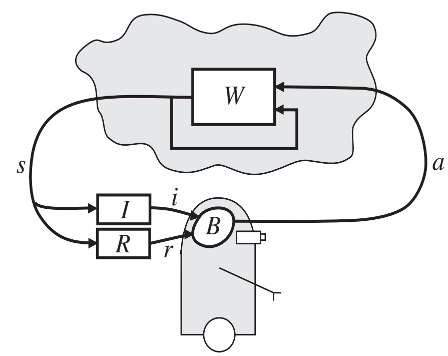
Discussion

Reinforcement Learning

Reinforcement learning is

- a popular model of the learning problems encountered by an agent that learns behavior through trial-and-error interactions with a dynamic environment
- a class of problems, rather than a technique

Reinforcement Learning Model



Reinforcement Learning Differs From Supervised Learning

- no presentation of input/output pairs
- agent chooses actions, receives reinforcement
- worlds are usually non-deterministic
- on-line performance is important
- system must explore the space of actions

RL = SL When

- there are two possible actions, *and*
- reinforcement is Boolean, *and*
- the world is deterministic

When action a is chosen in situation s

- if $r = 1$ then learn $f(s) = a$
- if $r = 0$ then learn $f(s) = \neg a$

Models of Optimal Behavior

- finite horizon

$$E\left(\sum_{t=0}^{k} r_t\right)$$

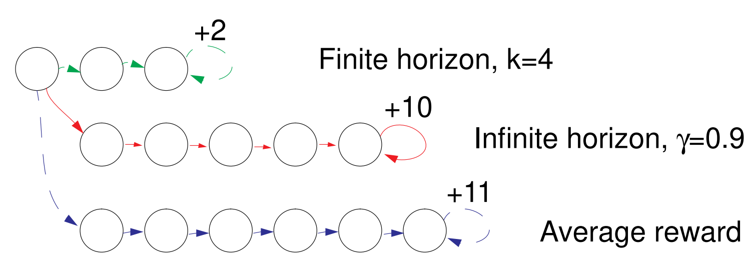
- infinite horizon discounted

$$E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right)$$

- average case

$$\lim_{k \to \infty} E\left(\frac{1}{k} \sum_{t=0}^{k} r_t\right)$$

Comparing Models of Optimality



Measuring Learning Performance

Having chosen a model of optimality, how can we characterize how close our algorithm comes?

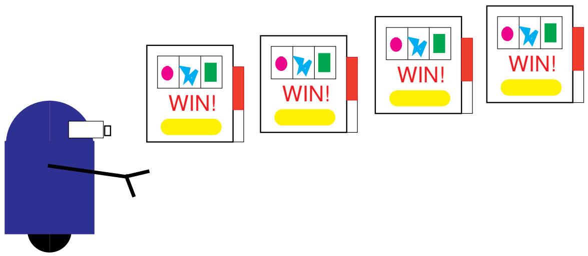
- asymptotic convergence to optimal behavior
- regret as a function of run-length, N

$$N\,p_{\max} - \sum_{t=1}^{N} E(r_t)$$

- PAC-like frameworks that divide into training and testing seem inappropriate for reinforcement learning

K-Armed Bandit

- world has only one state
- actions pay off independently on each trial
- world is stationary



How to trade off exploration versus exploitation?

Bayesian Approach

- start with distribution on underlying parameters
- in the finite horizon case
  - choose the best last action for every possible sequence of actions and observations using Bayesian updating of belief distribution
  - choose the best next-to-last action assuming that the best choice will be made on the last step
- this can be made more efficient in special cases

Berry & Fristedt, 1985

Minimax Algorithm

Assume the world antagonistically chooses the arms.

For finite horizon N, let $k = \sqrt{N}$

- choose arms alternately until a_i has k more successes than a_j
- pull a_i forevermore

This algorithm has regret $O(\sqrt{N})$, which is optimal.

For infinite-horizon, discounted optimality, set $k = \sqrt{\frac{1}{1-\gamma}}$.

Randomized Strategies

Estimate expected reinforcement of each action, $ER(a)$

- with probability $1 - p$, pick the action with the highest expected reinforcement; with probability p choose uniformly at random among the other actions
- choose actions according to the Boltzmann distribution:

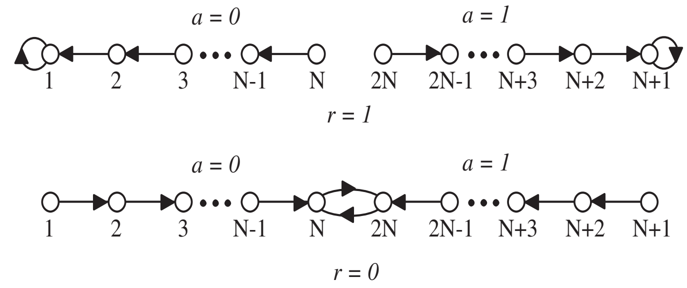
$$P(a) = \frac{e^{ER(a)/T}}{\sum_{a' \in A} e^{ER(a')/T}}$$

No formal guarantees, but in popular use.

Thrun, 1992

Learning Automata

Tsetlin Automaton *Tsetlin, 1973*



Shown to be ε-optimal as number of states is increased when one of the actions has success probability > .5

Linear Reward-Penalty

Internal state is a probability distribution, $\{p_1, \ldots, p_n\}$, over possible actions

- choose actions according to the distribution
- when action a_i succeeds

$$p_i = p_i + \alpha(1 - p_i)$$
$$p_j = p_j - \alpha p_j \qquad \text{for } j \neq i$$

- when action a_i fails

$$p_i = p_i - \beta p_i$$
$$p_j = p_j + \beta\left(\frac{1}{n-1} - p_j\right) \qquad \text{for } j \neq i$$

Narendra & Thathachar, 1989

Linear Reward-Inaction

Same as linear reward-penalty, but doesn't adjust the probabilities on failure

- converges w.p.1 to a unit vector
- ε-optimal: can increase probability of convergence to correct solution by decreasing α
- no understanding of the regret as a function of α

There are more complex schemes, such as the Lai-Robbins algorithm, that are asymptotically optimal.

Lai, 1987

REINFORCE Algorithms

Williams describes a broad class of network-based algorithms of the form

$$\Delta w_{ij} = \alpha (r - b_{ij}) \frac{\partial}{\partial w_{ij}} \ln(g_i)$$

where

- $g_i(\xi, w, x) = \Pr(y_i = \xi \mid \boldsymbol{w}, \boldsymbol{x})$
- b_{ij} is a reinforcement baseline

and shows they do gradient descent in r on average.

Linear reward-inaction is an instance of this scheme; these methods often suffer from slow convergence

Williams, 1992

Interval Estimation

Uncertainty-based exploration seems more efficient

- store statistics for each action a_i: number of successes, s_i, and trials, n_i
- compute upper bound of a $(1 - \alpha)\,\%$ confidence interval on the success probability of each action:

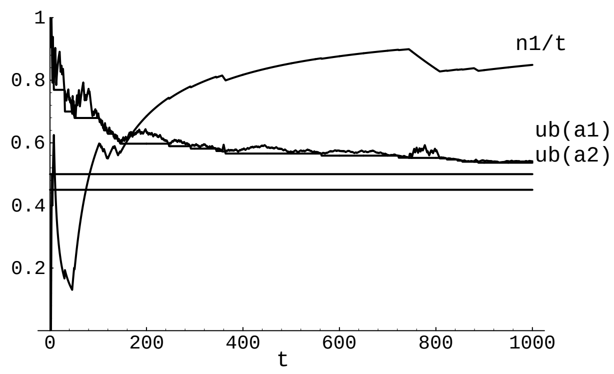
$$\mathrm{ub}(a_i) = \frac{s_i}{n_i} + z_{\alpha/2}\sqrt{\frac{1}{n_i}\,\frac{s_i}{n_i}\left(1 - \frac{s_i}{n_i}\right)}$$

- choose action with highest ub

Works best in empirical trials; no proof, but probably optimal (within constant factor) regret
Kaelbling, 1993

IE Sample Run

Payoff probability of $a_1 = 0.5$; of $a_2 = 0.45$



Extensions

- most methods can be extended to work with real-valued reinforcement
- in many cases, they can be extended in an *ad hoc* way to work in non-stationary environments; little or no theory available

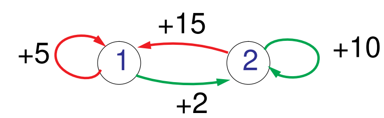
Multiple States

If the world has more than one possible state, then we can use these algorithms by making a copy for each state.

We'll see how to be smarter about this in the section on generalization.

Delayed Reinforcement

- actions determine immediate reward
- actions change situation
- like a network of bandit problems
- choose actions to maximize some measure of long run performance



Markov Decision Processes

- finite set of states S
- finite set of actions A
- discrete time
- actions result in immediate rewards

$$R: S \times A \rightarrow \Re$$

- actions result in state transitions

$$T: S \times A \rightarrow \Pi(S)$$

- transitions depend only on the last state and action

Bellman, 1957; Howard, 1960

Infinite Horizon Discounted MDPs

- a policy is a mapping from states to actions: $\pi: S \rightarrow A$
- optimal value of a state $V^*(s) = \max_\pi E_\pi\left(\sum_{t=0}^{\infty} \gamma^t r_t\right)$
- the optimal value function V^* is unique

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

- a stationary optimal policy exists

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

Find an Optimal Policy: Value Iteration

Value iteration algorithm:

initialize $V(s)$ **arbitrarily**
loop
loop for $s \in S$

$$V(s) := \max_a \left[R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V(s') \right]$$

end loop
end loop

An approximately optimal value function yields an arbitrarily good policy:

$$\left|V^* - \hat{V}\right|_{\sup} \le \varepsilon \Rightarrow \left|V^* - V_{\hat{\pi}}\right|_{\sup} \le 2\varepsilon \frac{\gamma}{1-\gamma}$$

Singh and Yee

More about Value Iteration

- error $\left|V^* - \hat{V}\right|_{\sup}$ decreases by a factor of γ on every iteration
- relaxations (assignments to V) can be done
  - asynchronously
  - in parallel

Bertsekas & Tsitsiklis, 1989

Another Method: Policy Iteration

choose an arbitrary policy π
loop
compute the value function under π**:**

$$V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V_\pi(s')$$

improve the policy at each state:

$$\pi'(s) \leftarrow \operatorname{argmax}_a \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V_\pi(s')\right)$$

π **:=** π'
until no improvement is possible

Guaranteed to terminate with optimal policy.
Howard, 1960

Learning an Optimal Policy

- first learn a model (the T and R functions)
- then find the optimal policy using value or policy iteration
- finally, execute that policy forever more

Not very satisfactory for an agent in a potentially changing world. But we'll talk more about this later.

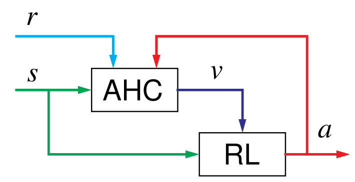
Instead, let's try to learn a policy without first learning a model.

Temporal Credit Assignment

How do we know whether the action we chose was a good one, when it might have far reaching effects?

- we could wait until the “end” and reward the actions if the result was good and punish them, if bad
- instead, we’ll use insights from value iteration to adjust the value of a state based on the immediate reward and the value of the next state
- AHC and Q-learning are both instances of this *temporal difference* learning method
- AHC is often called TD

Adaptive Heuristic Critic



- AHC learns value of states
- value serves as local reinforcement signal to RL, which only maximizes immediate reward
- like a version of policy iteration if RL changes more slowly than AHC

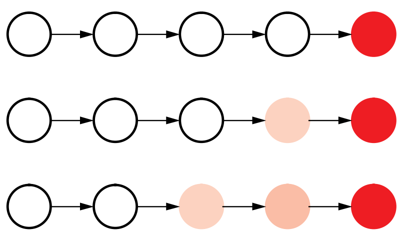
Implementing AHC: TD(0)

Update value function given “learning instance”: (s, a, s', r)

$$V(s) \leftarrow (1-\alpha) V(s) + \alpha (r + \gamma V(s'))$$

$$V(s) \leftarrow V(s) + \alpha (r + \gamma V(s') - V(s))$$

α is a learning rate



TD(λ)

More generally, we can update states other than the immediately previous one.

On every step, update all s

$$V(s) \leftarrow V(s) + \alpha (r + \gamma V(s') - V(s)) e(s)$$

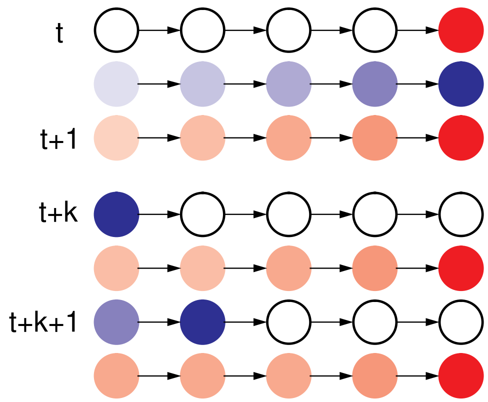
where $e(s)$ is the eligibility of s

$$e(s) = \sum_{k=1}^{t} (\lambda\gamma)^{t-k} \mathrm{I}(s = s_k)$$

Converges with probability 1.

Sutton, 1988; Dayan, 1992; Dayan & Sejnowski, 1994

TD(λ) Example



Q Values

Let $Q^*(s, a)$ be the expected discounted reinforcement of taking action a in state s, then continuing with the policy of choosing actions to maximize $Q*$

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

Theorem:

$\pi^*(s) = \operatorname{argmax}_a Q^*(s, a)$ is an optimal policy

Watkins, 1990

Q-Learning

Estimate the Q values on-line, finding the policy and the value function together:

$$Q(s, a) \leftarrow (1 - \alpha) Q(s, a) + \alpha (r + \gamma \max_{a'} Q(s', a'))$$

given learning instance: (s, a, s', r) .

Theorem: If each action is executed in each state an infinite number of times, and α is decayed, the *Q* values will converge to $Q*$, yielding an optimal policy.

Works reliably.

Watkins & Dayan, 1992

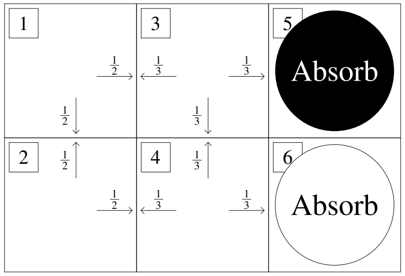
Learning and Using World Models

We've talked about

- finding a policy given a correct model *a priori*
- learning a policy without learning a model

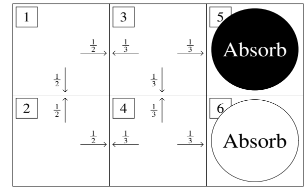
Will it be useful to learn models even if we can learn policies without them?

Stochastic Prediction



For each state, what is the probability of eventual absorption by black?

Straightforward Computation



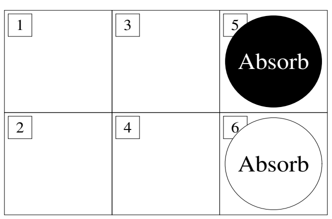
If $E^*(s)$ = Prob(eventually absorbed by black starting at s)

and $p(s)$ = Prob(absorbed by black in 1 step, from s)

and $t(s, s')$ =Prob(goto s'in 1 step, from s), then

$$E^*(s) = p(s) + \sum_{s' \in succs(s)} t(s, s') E^*(s')$$

Learning Stochastic Prediction



We see something like:

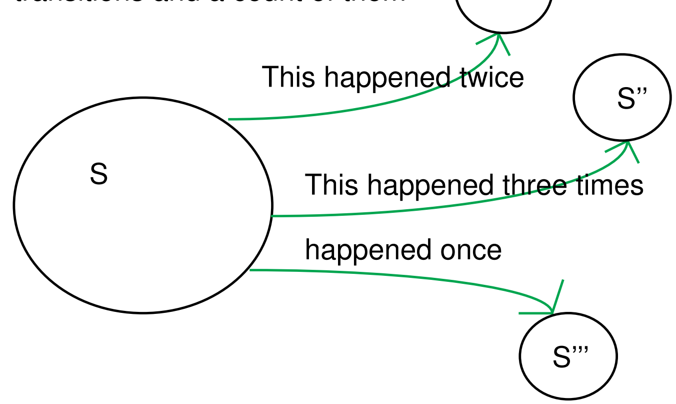
$3 \rightarrow 4 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 6$ (White absorb)

$3 \rightarrow 5$ (Black absorb)

$1 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 5$ (Black absorb)

Learning a Model

All states remember all their transitions and a count of them



Estimating Transition Probabilities

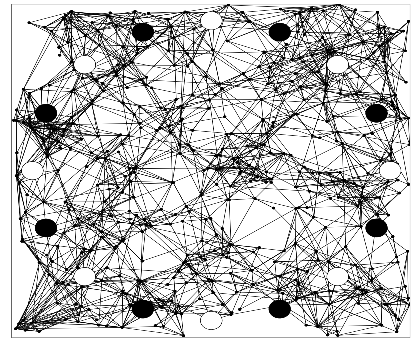
Count all transitions and estimate transition probabilities with Maximum Likelihood:

$$\hat{t}(s, s') = \frac{\text{Number-times-seen } (s \rightarrow s')}{\text{Number-times-started-in } (s)}$$

Compute absorption probabilites by solving

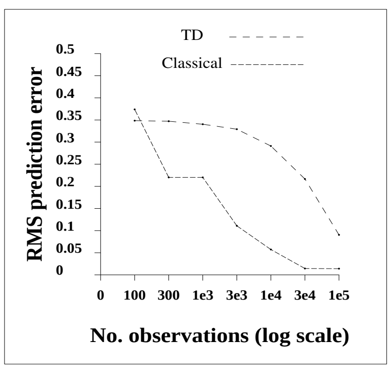
$$\hat{E}^*(s) = \hat{p}(s) + \sum_{s' \in \, succs(s)} \hat{t}(s, s')\hat{E}^*(s')$$

An Example Problem



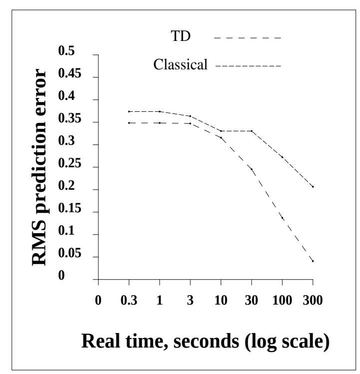
500 states, on average five stochastic successors per state.

Comparing TD and Model Methods



Average prediction error versus number of observations

Comparing TD and Model Errors



Average prediction accuracy versus real time

Comparison

Temporal differencing:

- inefficient use of data
- good real-time performance

Model-based method:

- efficient use of data
- terrible real-time performance

Comparison

The model-based method computes the best estimates for every state on every time step.

TD (and Q-learning) only do computation for the states the system is actually in.

Surely there must be some middle ground!

Dyna

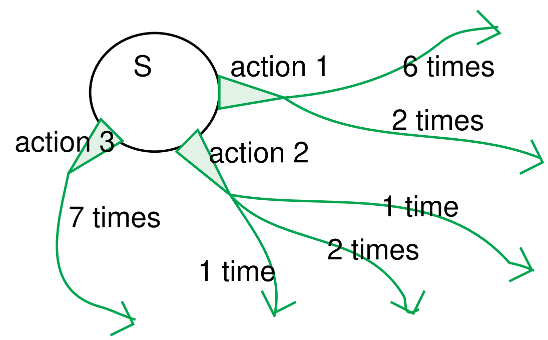
Uses the middle ground by combining model-based and TD methods in parallel:

- use experience to build a model
- use experience to adjust policy
- use model to adjust policy

Sutton, 1990

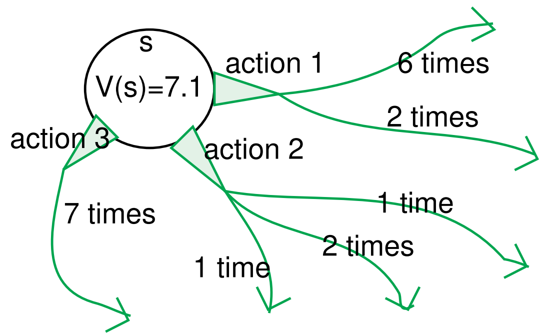
Dyna

Remember all actions, and all their outcomes



Dyna

Also record the current estimate of $V(s)$



Dyna's Model

Estimate transition probabilities from model

$$\hat{T}(s, a, s') = \frac{\text{number-times-seen}\,(s, a \rightarrow s')}{\text{number-times-tried}\,(s, a)}$$

Also remember the mean reward experienced so far for each state-action pair: $\hat{R}(s, a)$

The Dyna Algorithm

1. We are in state s
2. We choose action a
3. We observe the new state we enter, s', and our reward r
4. We add this information into our model
5. We update our estimate of

$$V(s) \leftarrow \max_a \left(\hat{R}(s, a) + \gamma \sum_{s'} \hat{T}(s, a, s') V(s') \right)$$

6. And then we do a little more computation before the next time step...

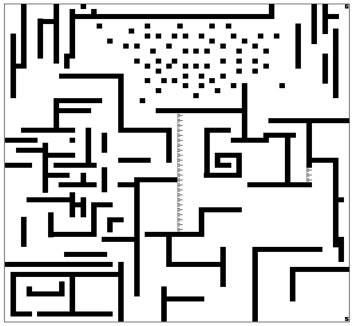
Dyna Algorithm continued

7. Pick k other states in our memory at random (e.g. $k = 50$). Call them $\{s_1, s_2, ..., s_k\}$.
8. For each k update the $V(s_k)$ estimate:

$$V(s_k) \leftarrow \max_a \left(\hat{R}(s_k, a) + \gamma \sum_{s'} \hat{T}(s_k, a, s') V(s') \right)$$

- About k times the computation of Q-learning
- Vastly less computation than full model-based method

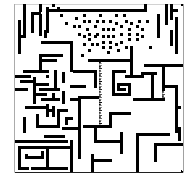
An Example Gridworld Task



Non-zero reward only at goal; some 1-way barriers

All exploration uses “optimism in face of uncertainty”

Q and Dyna Compared



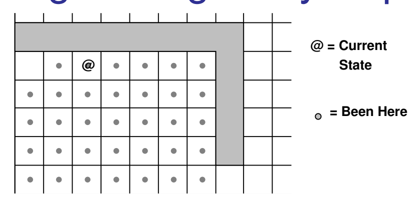
- Q-learning: 13,108 + 518,000 steps until optimal (531,000 backups)
- Dyna: 13,108 + 48,800 steps until optimal (3,055,000 backups)

Models can help converge to the correct value function more quickly; especially useful during initial exploration

What Are the Models Doing?

1. They help us converge to the correct $V^*(s)$ value function more quickly.
2. During initial exploration they can also help considerably.

Q-learning During Early Exploration



top left square still unexplored. Say it has these Q-values:

0	1	@	3	4	5	4
1	2	3	4	5	4	3
2	3	4	5	4	3	2
3	4	5	4	3	2	1
4	5	4	3	2	1	0

@ = Current State

After Some Transitions...

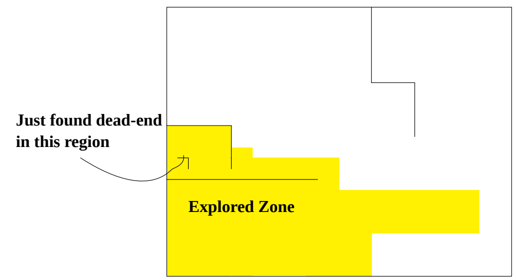
2	1	2	3	4	5	4
1	2	3	4	5	4	3
2	3	4	5	4	3	2
3	4	5	4	3	2	1
4	5	4	3	2	1	0

It discovers the dead end. But local Q values are still relatively small. Much more physical moving around is required before the Q values change sufficiently to guide the agent to the bottom right exit.

Dyna will exit the dead-end region more quickly.

Two Problems with Dyna

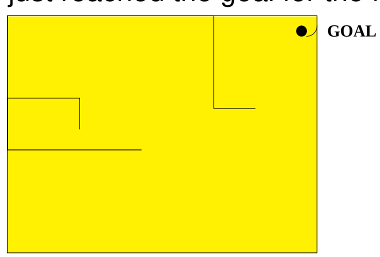
1. What if we are in this maze:



Then why waste backups on random parts of the state space? We should concentrate on exiting the dead end.

Two Problems with Dyna

2. Imagine we've just reached the goal for the first time:



- This is good news! But Dyna continues to update random states
- We'd prefer it to update states near the goal first, then work backwards.

A Solution

Don't update randomly. Update the states which are expected to be most usefully updated.

This idea is used by

- Prioritized Sweeping (*Moore and Atkeson, 1992, 1993*)
- Queue-Dyna (*Peng and Williams, 1992*)

Prioritized Sweeping / Queue-Dyna

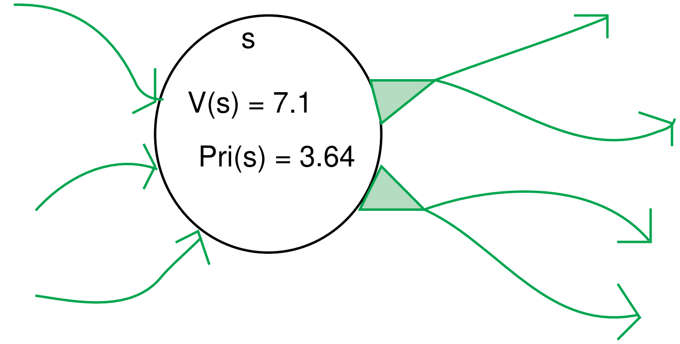
1. We are in state s
2. We choose action a
3. We observe the new state we enter, s', and our reward r
4. We add this information into our model
5. We update our estimate of

$$V(s) \leftarrow \max_a \left(\hat{R}(s, a) + \gamma \sum_{s'} \hat{T}(s, a, s') V(s') \right)$$

6. And then we do a little more computation before the next time step...

(These steps exactly the same as Dyna)

Extra Information in Model



- each state has a *priority*
- each state remembers its predecessors

Using the Priority Measures

On each step, perform backups on the k states which currently have highest priority. (e.g. $k = 50$)

For each state:

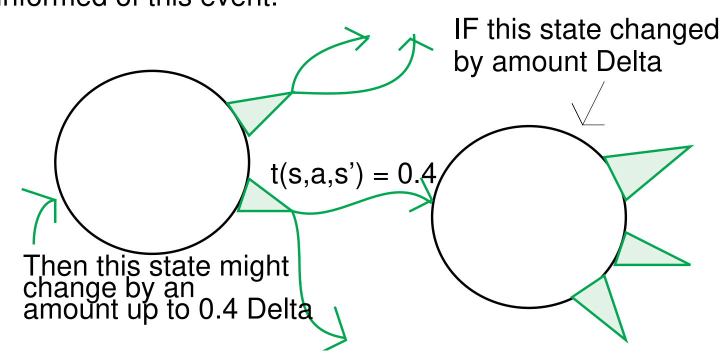
- Remember the current value of state $V_{old} \leftarrow V(s)$
- Update the state's value with Bellman's equation

$$V(s) \leftarrow \max_a \left(\hat{R}(s, a) + \gamma \sum_{s'} \hat{T}(s, a, s') V(s') \right)$$

- Set the state's priority back to 0
- Compute the value-change $\Delta = |V_{old} - V(s)|$
- Use Δ to possibly change the priorities of some of our predecessors.

Computing New Priorities

If we've done a backup on state s and it has changed by amount Δ then the immediate predecessors of s are informed of this event.



Global Behavior

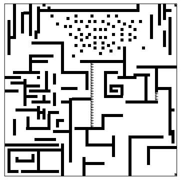
When a real-world transition is “surprising”:

- Lots of computation is directed to propagate this new information.

When a real-world transition is “boring”:

- The computation continues in the most deserving part of state space.

Q, Dyna, Prioritized Sweep Compared



- Q-learning: 13,108 + 518,000 steps until optimal (531,000 backups)
- Dyna: 13,108 + 48,800 steps until optimal (3,055,000 backups)
- PriSweep: 13,108 + 15,000 steps until optimal (1,010,000 backups)

Some Other Model-based Methods

- Real-time Dynamic Programming (*Barto, Bradtke, and Singh, 1991*)
- Prioritized Sweeping with Geometry (*Moore, 1993*)
- Generalization with Models. e.g.:
  - *Miller, 1987, 1989*
  - *Atkeson, 1989*
  - *Moore, 1991*
  - *Mahadevan, 1992*
  - *Schaal and Atkeson, 1994*

Real-time Dynamic Programming

- uses Q-learning to concentrate computational effort on the areas of state-space that the robot is likely to spend most of its life
- can result in significantly less computation than conventional value-iteration
- avoids computing in places robot unlikely to travel by considering start state

Barto, Bradtke & Singh, 1993

Prioritized Sweeping with Geometry

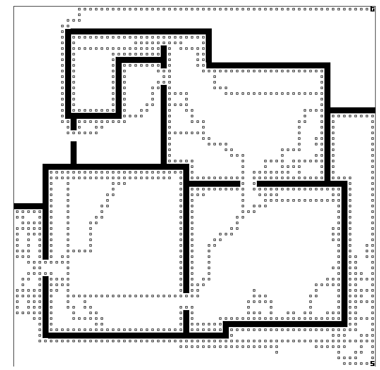
Additional assumptions can improve PriSweep:

- it is told what the reward function $R(s, a)$ is in advance
- it is told the geometry of state-space, and that the only possible paths through state space are continuous

These assumptions can be incorporated into prioritized sweeping fairly easily.

Moore, 1993

An Example



PriSweep: 19,800 steps until optimal (830K backups)
With Geometry: 3,800 steps until optimal (850K backups)

World Models Can Help Generalization

In (*Mahadevan, 1992*) a mapping is learned from 144-element certainty-grid sonar sensors to next state predictions.

Shallow lookahead search makes use of generalized state transition model.

Enabled substantial speed up on box pushing, wall following and unwedging.

Generalization was also applicable between different tasks.

Another Generalization Example

Juggling (devil-sticking) robot (*Schaal &Atkeson, 1994*)

- real-time juggling task
- six continuous state variables (6-d state space)
- six continuous action variables (6-d action space)
- impossible to discretize

Learned a next-state model, and generalized it using local regression.

Robot explicitly explores state space to find a locally linear region.

Local linear model used for automatic controller design.

Models and Continuous Spaces

So far, all the learning algorithms have used discrete states. What if the world is continuous?

- we could discretize the continuous state-space with a multidimensional array.
- can sometimes work
  - e.g. BOXES: *Michie and Chambers, 1968*
- but:
  - costs grow exponentially with dimensionality
  - cannot make boxes too coarse

Models and Continuous Spaces

So far, all the model learning algorithms have used discrete states. What if the world is continuous?

- we can use function approximation to generalize the model (as we have just seen). But how can the model be used?
  - search (e.g. *Mahadevan, 1992*) sometimes impractical
  - local linear control design often not applicable
  - do dynamic programming, using the generalized model to predict the results of transitions between cells? But then we'd need to discretize anyway.

Adaptive Resolution Models

There is a third alternative. Use a discretization of state space, but *learn* a good discretization of state space while we learn the model and the optimal value function.

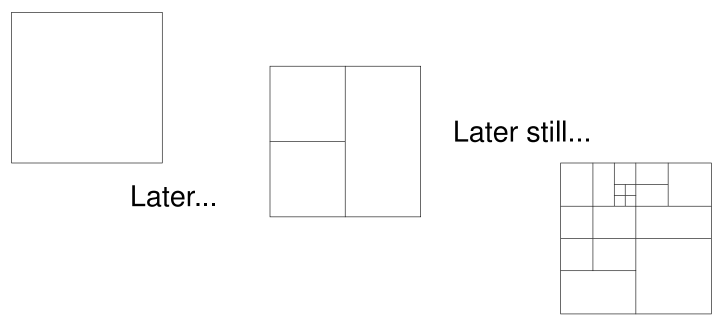
A few researchers have tried to do this:

- (*Simons, Van Brussel and De Schutter, 1982*)*
- Variable Resolution DP (*Moore 1991*)
- G-algorithm* (*Chapman and Kaelbling 1991*)
- Parti-Game Algorithm (*Moore 1993*)

* = did not use a model

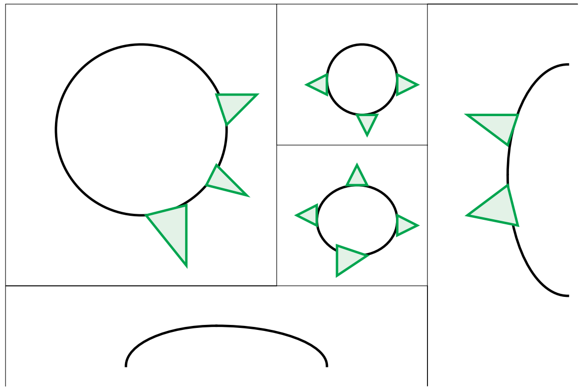
PartiGame Algorithm

Begin with one large partition. Increase the resolution by subdividing state-space where the learner predicts higher resolution is needed.



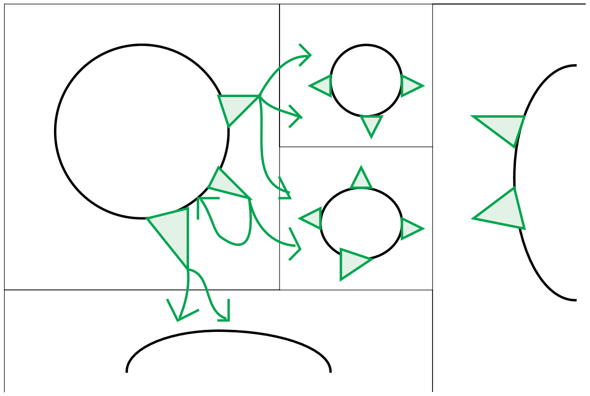
PartiGame's World Model

The actions of a cell consist of aiming at a neighbor cell



PartiGame's World Model

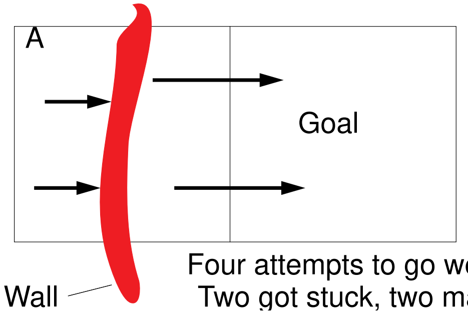
All previous transitions between cells are remembered.



Computing Optimal Policy

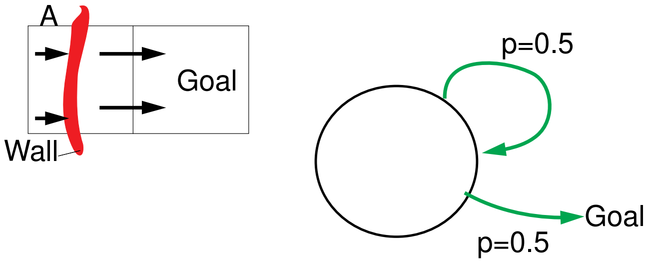
We cannot treat the problem as a Markov Decision Task.

Imagine we are solving a Maze problem, with two big cells:



Four attempts to go west from Cell A
Two got stuck, two made it

Markov Model is Wrong



Markov says, "Keep going East. I expect you to reach the goal in two steps."

Truth: If you keep going East, 50% chance you'll get to goal and 50% you'll get permanently stuck.

The Worst Case Assumption

To avoid this *hidden state* problem, PartiGame uses a worst case assumption.

- solves the system as a game
- when an action has been previously seen to have multiple outcomes, we assume the worst case

The Good News

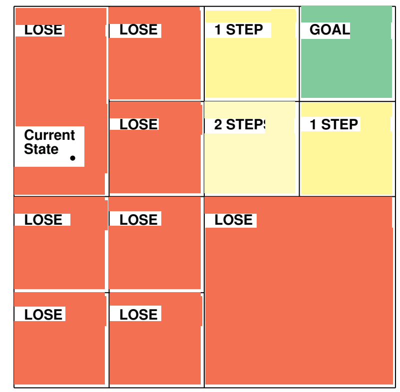
- robust. If we find a solution we can trust it.
- can identify which cells need to be split

The Bad News

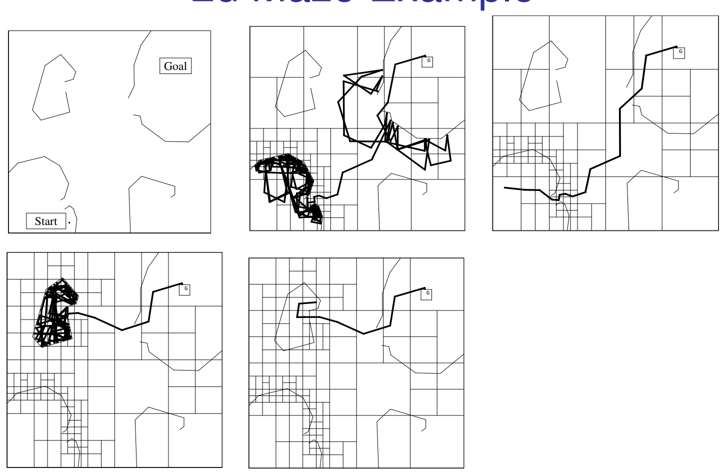
- we have to assume a deterministic system

Which Cells to Split?

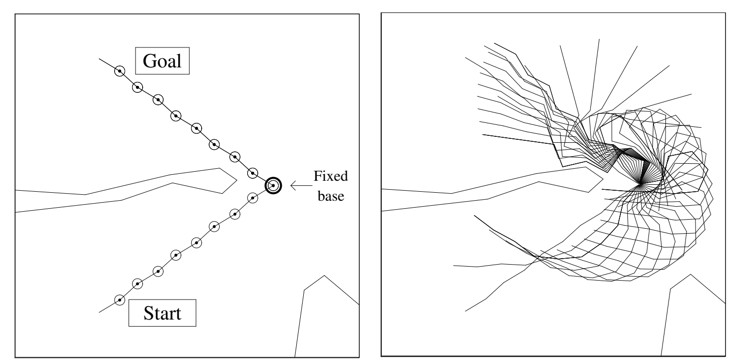
Answer: those on the border between winners and losers.



2d Maze Example



9-Joint Planar Manipulator Kinematics



World Models: Conclusion

- To get maximal information out of all the state transitions observed so far, the agent should solve the Markov system (e.g. by a system of linear equations) or the Markov decision task (e.g. by dynamic programming) after every new state transition is observed.
- This can learn far faster in terms of data needed.
- But is can require huge amounts of computation.
- So...

World Models: Conclusion

So...

- Dyna does more computation than Q-learning but far less than full DP, and it can reap large benefits
- exploration is enhanced by learning a model
- Prioritized Sweeping and Queue-Dyna do the same amount of computation as Dyna, but concentrate it in important places

Generalization and Hierarchy

What if we want to use something other than tables to store our knowledge?

Generalization

There are many mappings we might want to learn:

- $S \to A$
- $S \to V$
- $S \times A \to Q$
- $S \times A \to S$

Tables are not always a good representation

- scale badly in size
- do not offer transfer of knowledge

Generalization techniques lose representational power but gain efficiency and transfer.

Generalization Methods

We can generalize

- over input space S
- over action space A

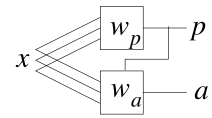
We can use a variety of techniques

- neural networks
- Boolean function learning
- CMAC
- specialized RL methods

Some illustrative examples follow...

LARC

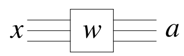
Linear Associative Reinforcement Comparison *Sutton, 1984*



- $p = w_p \cdot x$
- $a = \begin{cases} 1 & \text{if } w_a \cdot x + \nu > 0 \\ 0 & \text{otherwise} \end{cases}$; ν is normal noise
- $\Delta w_{a,i} = \alpha (r - p)(a - 1/2) x_i$
  $\Delta w_{p,i} = \alpha (r - p)$

CRBP

Complementary Reinforcement Back Propagation



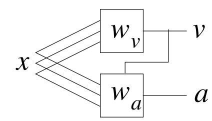
- $a_i = \begin{cases} 1 & \text{with probability } w_i \cdot x \\ 0 & \text{otherwise} \end{cases}$
- If $r = 1$ then train (x, a)
- If $r = 0$ then train $(x, \bar{a})$ with $\bar{a} = [1 - a_1, \ldots, 1 - a_n]$

Generalizes over actions; needs Boolean reinforcement

Ackley & Littman, 1990

LARC with TD

Train v with the TD learning rule:



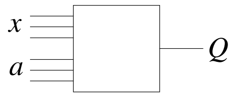
- $\Delta w_{v,i} = \alpha (r + \gamma v_{t+1} - v_t) x_i$
- $\Delta w_{a,i} = \beta (r + \gamma v_{t+1} - v_t)(a - 1/2) x_i$

Make v change slowly relative to a.

Used by *Barto, Sutton & Anderson, 1983*; *Lin, 1991*; many others
Sutton, 1984

Generalizing Q Learning

Generalize over both the input and action spaces:



The learning problem is

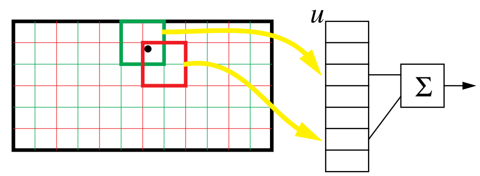
- supervised
- non-stationary

How to pick an action? Maybe gradient ascent on a to maximize $Q(x, a) + \frac{c}{N(x, a)}$

Lin, 1992

Using CMAC

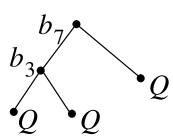
Developed by Albus; used by Watkins (*Watkins, 1990*)



- multiple tilings of the input space t_i
- finite number of parameters $u[j]$
- hash function, h, maps tiles to parameters
- $U(x) = \frac{1}{N}(u[h(t_1(x))] + \ldots + u[h(t_N(x))])$
- $\Delta u[t_i(x)] = \alpha(v - U(x))$

Decision Trees

G-learning algorithm (*Chapman and Kaelbling, 1991*) generalizes over input space



- start with no distinctions
- find input bit b such that $ER(b = 1) \neq ER(b = 0)$ or for some a, $Q(b = 1, a) \neq Q(b = 0, a)$ with high confidence
- split on that bit
- gather statistics and split the leaves

Logic-Based Methods

A policy with two actions can be thought of as a Boolean function

- learn functions in k-DNF with an algorithm like Valiant's
- learn functions expressible as small logical formulae through an on-line process of generate and test

Multiple Boolean-function learners can be combined to learn policies with many actions

Kaelbling, 1993, 1994

Hierarchy

Aggregation of states by

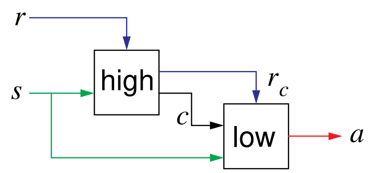
- same optimal action
- same subgoal
- same policy
- spatial proximity
- temporal proximity

What do we gain?

- smaller data structures
- less computation per learning instance
- fewer learning instances
- better asymptotic performance
- generalization
- better exploration

We typically make a trade-off; maybe worse asymptotic performance in exchange for smaller data structures and less computation

Feudal Reinforcement Learning

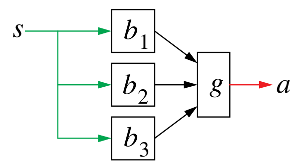


- high level knows a set of “commands” and can recognize their achievement
- high level rewards low level when command is achieved, independent of whether it was globally useful
- high level is rewarded by the world

Watkins, 89; Dayan &Hinton, 93

Partially Built Hierarchies

Behaviors mediated by a gating function:



- *Maes and Brooks, 1990*: fix the behaviors and learn the gating function
- *Mahadevan and Connell, 1991*: fix the gating function and provide individual reinforcement functions, then learn the behaviors

Goal-Based Decomposition

Compositional Learning (*Singh, 1992*)

- hierarchy based on temporal sequencing of subgoals
- decomposition given
- both elemental tasks and gating function are learned

Hierarchical Distance to Goal (*Kaelbling, 1993*)

- hierarchy based on spatial locality of states
- effective in very noisy environments
- efficient but asymptotically suboptimal
- decomposition is learned

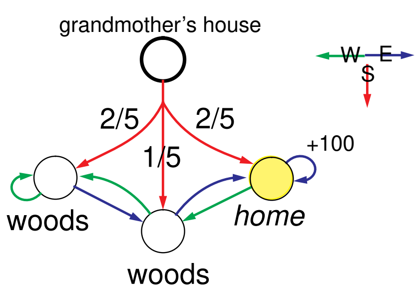
General Questions

- Are different “levels” learned in different phases? If so, how do we know when to switch?
- Is the hierarchical structure itself learned?
- Are there any convergence results? Convergence rate results?
- Is control given up by the high level when it “calls” a lower level?
- Does it apply to general reinforcement functions or just to goal-oriented problems?

Hidden State

aka: incomplete perception, perceptual aliasing...

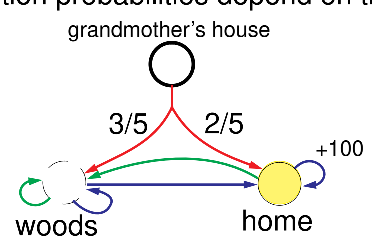
- agent gets observations only
- agent doesn't know current state



Observation graph

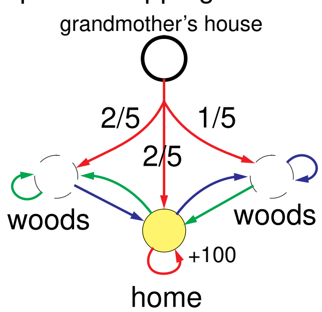
We could just treat the observations as the states and map them to actions.

But the transition probabilities depend on the policy...



Treating Observations as States

- Q-learning can be lead astray
- finding the optimal mapping is NP-hard



Chrisman, 92; Littman, 94

Probabilistic Policies

- Acting randomly can be better
- Q-learning-like algorithm converges to local max
- Optimal: right=$2 - \sqrt{2} \approx 0.6$, left=$\sqrt{2} - 1 \approx 0.4$
- Still NP-hard to find optimal policy

Jaakkola, Jordan, & Singh, 1994

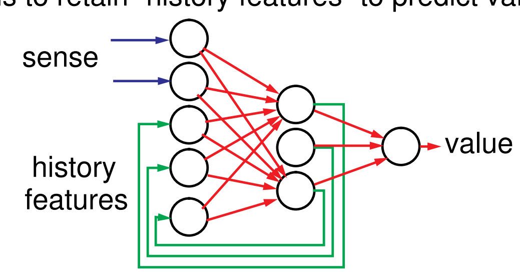
Memory

We could allow the policy to contain memory

- history window
- local memory (refreshed)
- registers
- program counter in a plan
- probability distribution over underlying world states (perfect memory)

Recurrent-Q

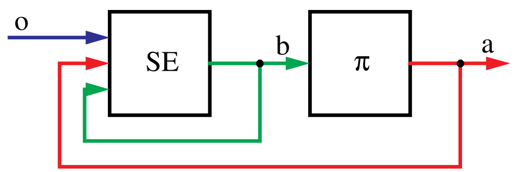
- one recurrent network for each action
- trained using backprop through time (unfolding)
- learns to retain “history features” to predict value



Lin and Mitchell, 92;Schmidhuber, 90, 91; Meeden, McGraw & Blank, 93

Perfect Memory

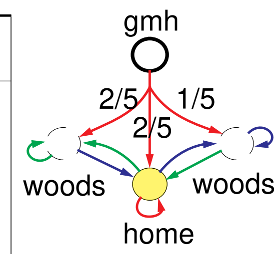
- makes the problem completely observable
- probability distribution over states (belief states)
- can be updated using a model



Belief-state Example

How do actions and observations affect the belief state?

	gmh	woods (west)	home	woods (east)
start	1	0	0	0
"south"	0	2/5	2/5	1/5
woods	0	2/3	0	1/3
"west"	0	2/3	1/3	0
woods	0	1	0	0



POMDP's

- use belief states as states
- good: belief state is sufficient for acting optimally
- bad: V is from n-dimensional vectors to values
- ok: these functions can be approximated using a simple representation

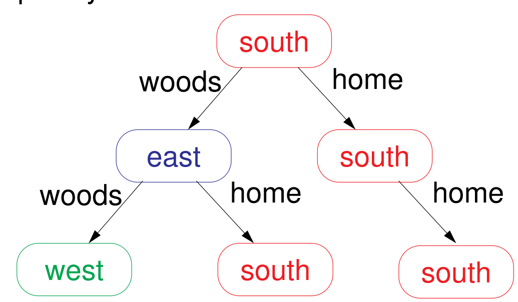
$$V(b) = \max_{l \in L}(l \cdot b)$$

- model can be learned (HMM's) (*Chrisman, 1992; McCallum, 1993*)

Cassandra, Kaelbling & Littman, 1994; Lovejoy, 1991

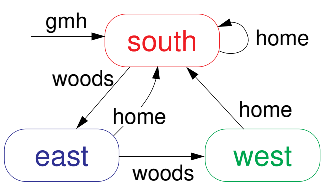
T-Step Policy Trees

- each linear function corresponds to a t-step policy
- l_s^k is the value for following tree k from state s
- completely-observable POMDP: l's are Q values



Policy graphs

- automatically derived from converged value function if optimal policy is "finitely transient"
- eliminates need for updating belief states



Hidden State: Conclusion

- lots of interesting new ideas
- understanding improving
- too hard for general approaches
- don't expect a silver bullet

Practical Reinforcement Learning

TD Gammon

Learned to play the game of Backgammon

- used temporal differencing in conjunction with a neural net
- two versions:
  - basic TD-Gammon used only raw features of the board (no hand-crafted features)
- the other version used several hand-crafted features

Tesauro, 1992

Basic TD-Gammon

Each “point” is a space on the board. There are 26 points, including two special locations.

Each point is represented by 6 inputs to the neural net.

- Are one or more black pieces on point?
- Are two or more black pieces on point?
- Are three or more black pieces on point?
- Are one or more white pieces on point?
- Are two or more white pieces on point?
- Are three or more white pieces on point?

Basic TD-Gammon

- Learned by self play. Continually played for several months on a high performance workstation.
- Used a neural net with tens of hidden units: thousands of connections.

Attained sufficient strength to be superior to all other computer implementations of backgammon.

Was easily beaten by backgammon experts.

But played at least as well as a typical, competent, human player.

Regular TD-Gammon

Used all the basic features (as above). But also had several handcrafted features (piece count, point count, race status etc.) Results:

	Training Games	Number Hidden units	Results against top players
Basic			Poor
TD 1.0	300,000	80	Lost 13 points in 51 games
TD 2.0	800,000	40	Lost 7 points in 38 games
TD 2.1	1,500,000	80	Lost 1 point in 40 games

TD-Gammon Questions

- Why so successful?
- no-one understands
- Experiments in the following games were less successful:
  - Connect-4 [*Tesauro, 1993*]
  - Go [*Schraudolph, Dayan, Sejnowski, 1994*]
  - Chess [*Thrun, 1993*]

Is it a special feature of Backgammon, or will other difficult combinatorial problems begin to fall to reinforcement learning?

Devil-Sticking Robot

Task: Juggle a devil stick

Method: Learn a model, apply optimal control design to what is learned.

Extra Help: Optimal control design assumes local linear dynamics. Sensible state variables were chosen.

- Learned to juggle for hundreds of successive hits.

Schaal & Atkeson, 1994

Box Pushing Robot

Task: Use a mobile robot to push boxes around the floor. (Keep pushing for as long as possible).

Method: Several variants of Q-learning.

Extra Help: Task broken into subtasks by hand.

- Resulting performance was of competitive quality with a simple set of handwritten rules.

Mahadevan & Connell, 1992

RatBot

Goal: Mail-delivery type mobile robot. Two components:

1. Navigate down corridors using sonar.
2. Build a map of the building.

Method: Q-learning with Neural Networks and further modifications.

Extra Help: Humans give example trajectories.

- Learned to navigate corridors quickly.

Lin & Hanson, 1993

Nerd Herd

Task: A group of four robots must cooperatively move a number of cylindrical pucks from a start region to a goal region.

Method: Distributed local controllers. Each controller uses Q-learning on a state-space discretized by a vector of condition signals.

Extra Help: Potentially useful intermediate subgoals were represented by a set of subgoal reward functions.

- learned very fast (within 15 minutes of operation)
- final performance within a few percentage points below that of a simple handwritten controller.

Mataric, 1994

An Industrial Application

Large industrial packaging task

- posed as an 18,000 state Markov decision problem
- one observation and one decision needed every few minutes
- need to learn within a couple of eight hour shifts
- without generalizing: would have taken several weeks
- with generalizing: Successful

Moore, 1992

Robust Reinforcement Learning

What if some states must never happen, even once, during learning?

Then extra knowledge of the system's dynamics (state transition function) *must* be supplied to the learning controller.

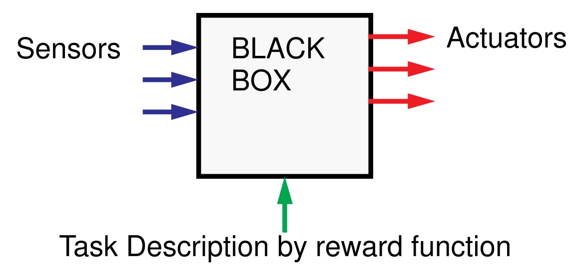
How can this extra knowledge be integrated into reinforcement learning?

Robust Reinforcement Learning

Used for Robot Motion Planning by *Singh, 1993*

- simulated dynamic mobile robot learns an optimal path to a goal without ever hitting a wall
- prior knowledge of location of walls and maximum capabilities of robot's dynamics given as a-priori knowledge

A Dream of Reinforcement Learning



A Black Box Reinforcement Learner. Are products like this going to appear in the near future?

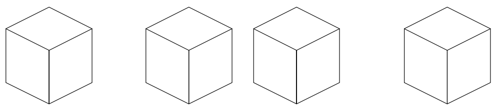
NO

Problem: Assumptionlessness

With few assumptions, the system is ***general*** (which is good) but it *has* to explore every possible state and will take impractical amounts of time.

An alternative to the black box:

- Reinforcement learning may become commercial by means of a wide number of specialized boxes. Each comes with a set of assumptions.



Expertise is needed to choose the best.

A Promising Location for Reinforcement Learning

Reinforcement living in the databases, Local Area Networks, and control loops of factories, automobiles, power plants, etc.

- these algorithms will watch the data streams and do modelling and controller design
- requires knowledge of system constraints and may need “permission” from human supervisors in order to test the resulting policies

Practical, Autonomous, Reinforcement Learning

- would have an enormous impact
- but is a long way away
- but is fairly unique in AI for having strong theoretical underpinnings
- reinforcement learning badly needs more examples of interesting, profitable learning control problems
- reinforcement learning is a promising area with many unexplored possibilities

The Future

Reinforcement Learning is a promising area with many unexplored possibilities.