Reinforcement Learning

Robotica Probabilistica

Reinforcement Learning

Learning from rewards (and punishments)

Learning to assess the value of states.

Learning goal directed behavior.

RL has been developed rather independently from two different fields:

1) Dynamic Programming and Machine Learning (Bellman Equation).
2) Psychology (Classical Conditioning) and later Neuroscience (Dopamine System in the brain)

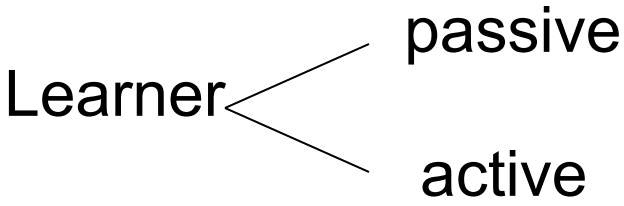
Reinforcement Learning

- Task
  - Learn how to behave successfully to achieve a goal while interacting with an external environment

    Learn through experience from trial and error

- Examples
  - Game playing: The agent knows it has won or lost, but it doesn't know the appropriate action in each state
  - Control: a traffic system can measure the delay of cars, but not know how to decrease it.

Reinforcement Learning



Sequential decision problems

Approaches:

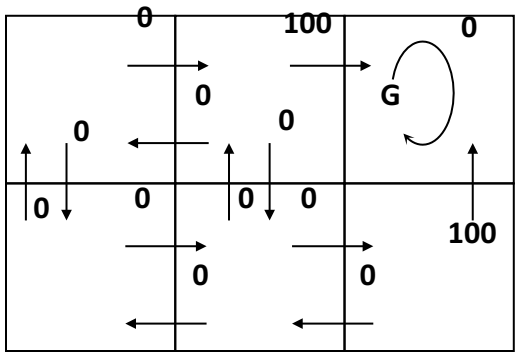
1. Learn values of states (or state histories) & try to maximize utility of their outcomes.
   - Need a model of the environment: what ops & what states they lead to
2. Learn values of state-action pairs
   - Does not require a model of the environment (except legal moves)
   - Cannot look ahead

Elements of RL

$$s_0 \xrightarrow{a_0 : r_0} s_1 \xrightarrow{a_1 : r_1} s_2 \xrightarrow{a_2 : r_2} \cdots$$

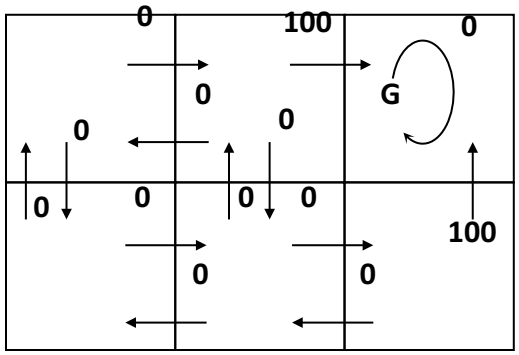
- **Transition model,** how action influence states
- **Reward R**, immediate value of state-action transition
- **Policy π,** maps states to actions

Elements of RL

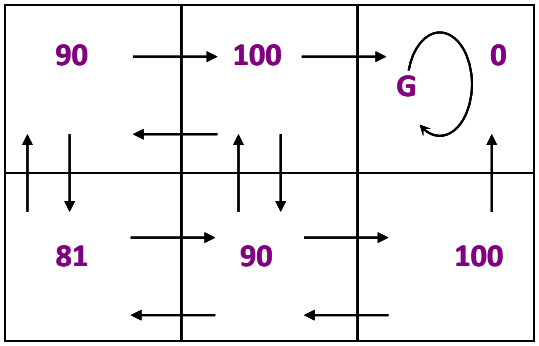


***r*(*state*, *action*)**
immediate reward values

Elements of RL



***r*(*state*, *action*)**
immediate reward values



V(**state**)* **values**

- **Value function**: maps states to state values

$$V^{\pi}(s) \equiv r(t) + \gamma r(t+1) + \gamma^2 r(t+1) + \ldots$$

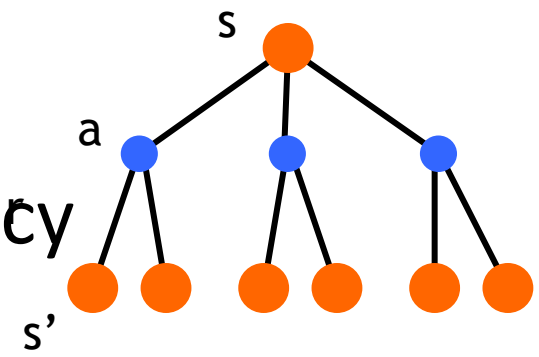
Discount factor $\gamma \in [0, 1)$ (here 0.9)

Computing return from rewards

- episodic (vs. continuing) tasks
  - "game over" after N steps
  - optimal policy depends on N; harder to analyze
- additive rewards
  - $V(s_0, s_1, \ldots) = r(s_0) + r(s_1) + r(s_2) + \ldots$
  - infinite value for continuing tasks
- discounted rewards
  - $V(s_0, s_1, \ldots) = r(s_0) + \gamma * r(s_1) + \gamma^2 * r(s_2) + \ldots$
  - value bounded if rewards bounded

Value functions

- state value function: $V^{\pi}(s)$
  - expected return when starting in *s* and following π
- state-action value function: $Q^{\pi}(s,a)$
  - expected return when starting in *s*, performing *a,* and following π



- useful for finding the optimal policy
  - can estimate from experience
  - pick the best action using $Q^{\pi}(s,a)$
- Bellman equation

Optimal value functions

- there's a set of *optimal* policies
  - V^π defines partial ordering on policies

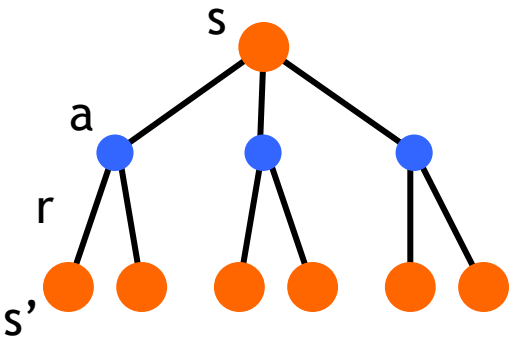
$$V^*(s) = \max_\pi V^\pi(s)$$

  - they share the same optimal value function
- Bellman optimality equation

$$V^*(s) = \max_a \sum_{s'} P^a_{ss'} \left[r^a_{ss'} + \gamma V^*(s') \right]$$

  - system of n non-linear equations
  - solve for V*(s)
  - easy to extract the optimal policy
- having Q*(s,a) makes it even simpler

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



Reinforcement Learning

- Execute actions in environment,

  observe results.

- Learn action policy π : *state* $\rightarrow$ *action* that maximizes <u>expected discounted reward</u>

  $E\,[r(t) + \gamma r(t+1) + \gamma^2 r(t+2) + \ldots]$

  from any starting state in *S*

Reinforcement Learning

- Target function is π : *state* $\rightarrow$ *action*
- However…
  - We have no training examples of form <*state*, *action*>
  - Training examples are of form <<*state*, *action*>, *reward*>

Utility-based agents

- Try to learn V^{π^*} (abbreviated V^*)
- Perform look ahead search to choose best action from any state s

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

- Works well if agent knows
  - $\delta : state \times action \rightarrow state$
  - $r : state \times action \rightarrow \mathrm{R}$
- When agent doesn't know δ and r, cannot choose actions this way

Q-values

- **Q-values**
  - Define new function very similar to *V**

$$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a))$$

  - If agent learns *Q*, it can choose optimal action even without knowing δ or R
- **Using *Q***

$$\pi^*(s) \equiv \arg\max_a Q(s, a)$$

Value Functions

State value function

V^{π}:S → Real

or

$V^{\pi}(s)$

The expected sum of discounted reward for following the policy π from state s to the end of time.

State-action value function

Q^{π}:S x A → Real

or

$Q^{\pi}(s, a)$

The expected sum of discounted reward for starting in state s, taking action a once then following the policy π from state s' to the end of time.

Solution Methods

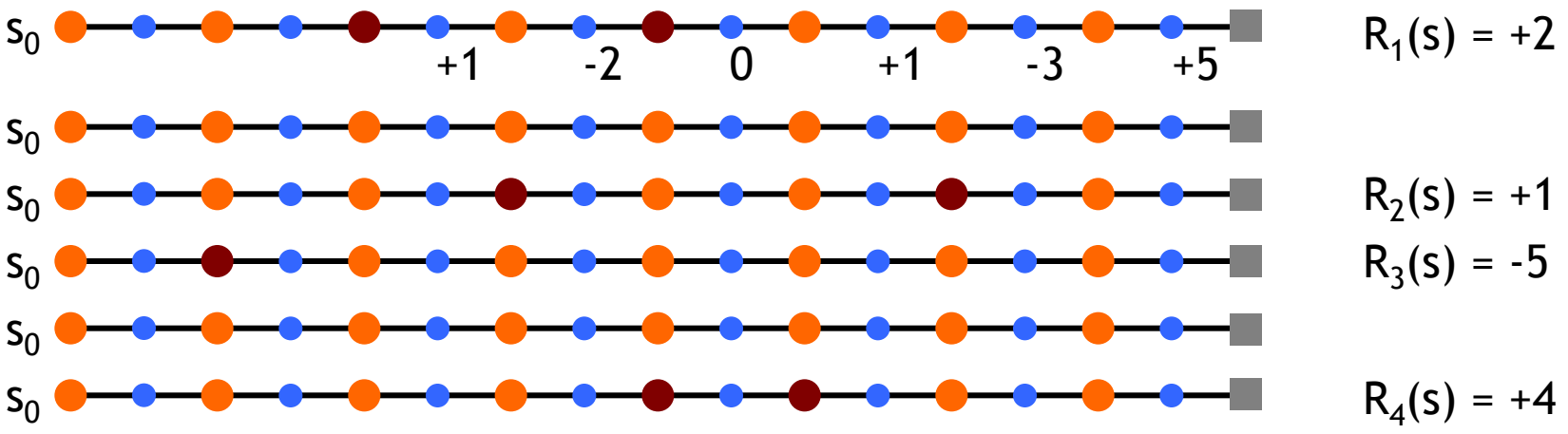
- **Model based**:
  - For example dynamic programming
  - Require a model (transition function) of the environment for learning
- **Model free**:
  - Learn from interaction with the environment without requiring a model
  - For example Q-learning…

Monte Carlo methods

- don’t need full knowledge of environment
  - just experience, or
  - simulated experience
- but similar to DP
  - policy evaluation, policy improvement
- averaging sample returns
  - defined only for episodic tasks

Monte Carlo policy evaluation

- want to estimate $V^{\pi}(s)$
  - = expected return starting from s and following π
  - estimate as average of observed returns in state s
- first-visit MC
  - average returns following the first visit to state s



$V^{\pi}(s) \approx (2 + 1 - 5 + 4)/4 = 0.5$

TD Learing

TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics.

Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

Whereas Monte Carlo methods must wait until the end of the episode to determine the increment of V (only then is known), TD methods need wait only until the next time step.

Initialize $V(s)$ arbitrarily, π to the policy to be evaluated
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    $a \leftarrow$ action given by π for s
    Take action a; observe reward, r, and next state, s'
    $V(s) \leftarrow V(s) + \alpha\left[r + \gamma V(s') - V(s)\right]$
    $s \leftarrow s'$
  until s is terminal

TD methods learn their estimates in part on the basis of other estimates. They learn a guess from a guess--they *bootstrap*

TD Learing

The next most obvious advantage of TD methods over Monte Carlo methods is that they are naturally implemented in an on-line, fully incremental fashion.

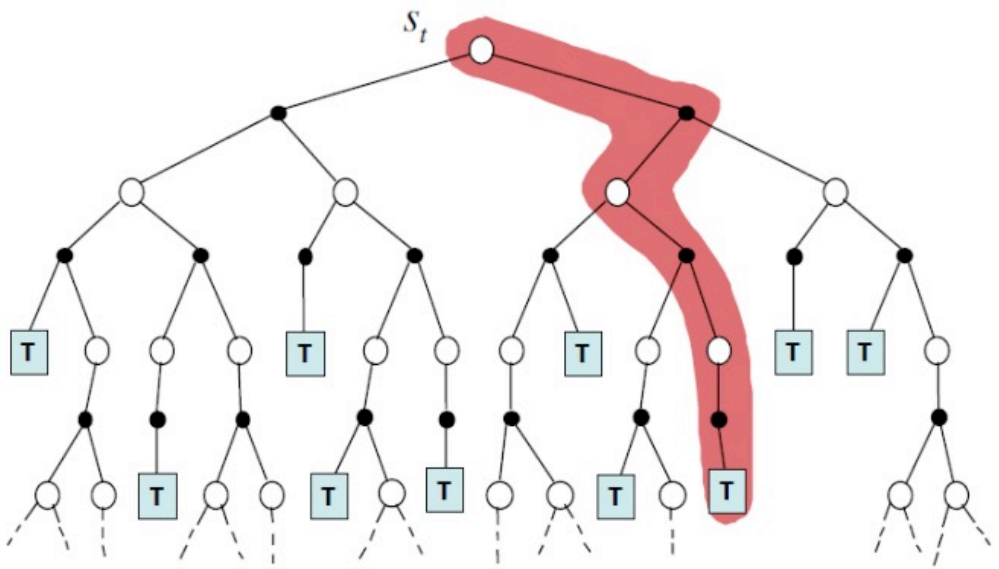
With Monte Carlo methods one must wait until the end of an episode, because only then is the return known, whereas with TD methods one need wait only one time step.

TD Learning

- MC learning

$$V(s_t) \leftarrow V(s_t) + \alpha\left[R_t - V(s_t)\right]$$

where R_t is the actual return following state s_t.

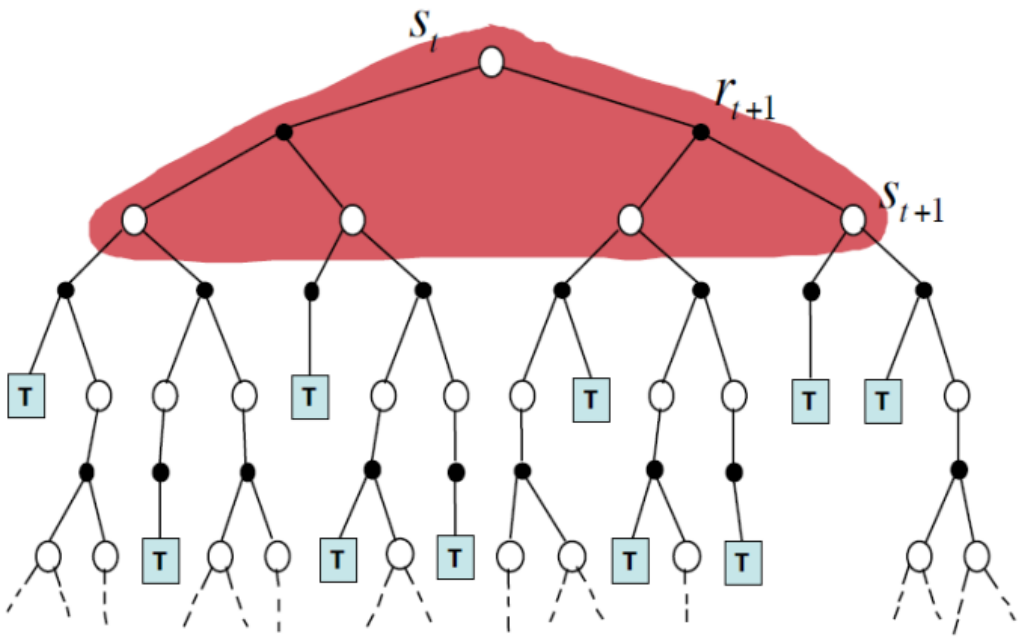


Monte Carlo uses an estimate of the actual return.

TD Learning

- DP method

$$V(s_t) \leftarrow E_\pi \left\{ r_{t+1} + \gamma V(s_t) \right\}$$

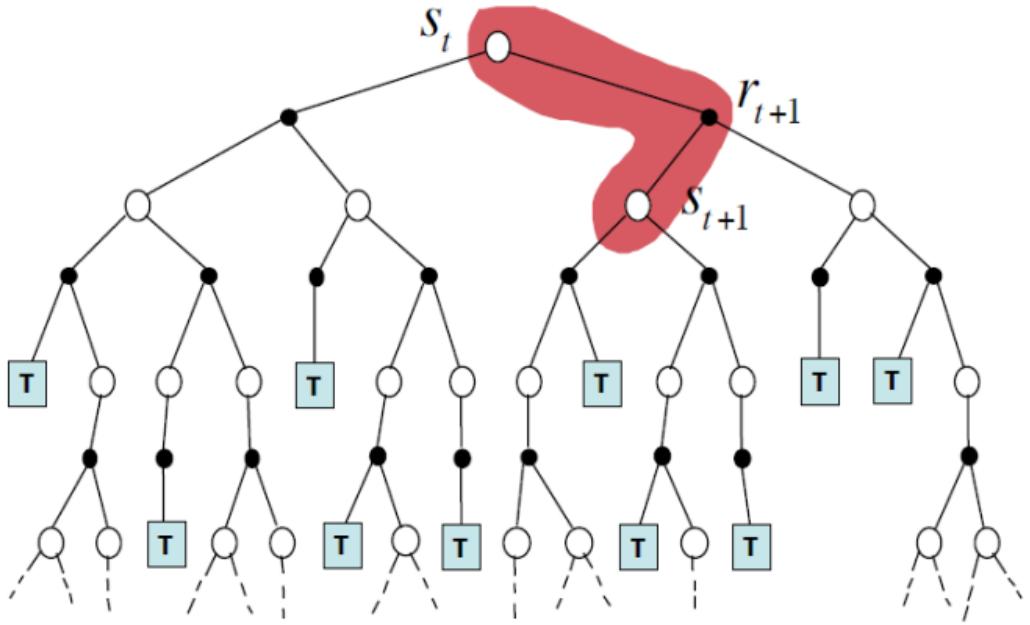


The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because V^π is not known and the current estimate is used instead.

TD Learning

- Simplest TD method

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$



TD samples the expected value and uses the current estimate of the value.

TD Learning

Tabular TD(0) for estimating V^π

Initialize $V(s)$ arbitrarily, π to the policy to be evaluated

Repeat (for each episode):

  Initialize s

  Repeat (for each step of the episode):

    $a \leftarrow$ action given by π for s

    Take action a; observe reward, r, and next state, s'

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

    $s \leftarrow s'$

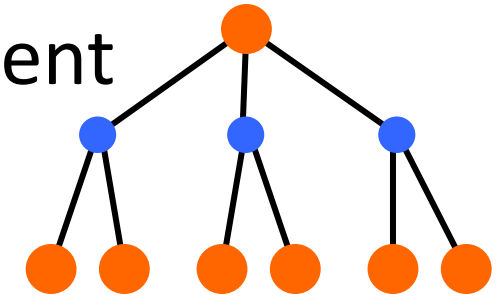
  until s is terminal

„temporal Difference“

Policy Improvement

- V^{π} not enough for policy improvement
  - need exact model of environment

$$\pi'(s) = \arg\max_a Q^{\pi}(s,a)$$



- estimate $Q^{\pi}(s,a)$

$$\pi_0 \rightarrow^{E} Q^{\pi_0} \rightarrow^{I} \pi_1 \rightarrow^{E} Q^{\pi_1} \rightarrow^{I} \ldots \rightarrow^{I} \pi^* \rightarrow^{E} Q^*$$

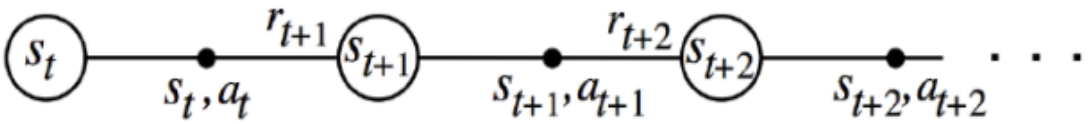
- MC control

$$V(s) \leftarrow V(s) + \alpha\,[R - V(s)]$$

  - update after each episode

SARSA: On-Policy TD

Estimate Q^{π} for the current behavior policy π.



After every transition from a nonterminal state s_t, do:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma\, Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right]$$

If s_{t+1} is terminal, then $Q(s_{t+1},a_{t+1}) = 0$.

SARSA: On-Policy TD

Turn this into a control method by always updating the policy to be greedy with respect to the current estimate:

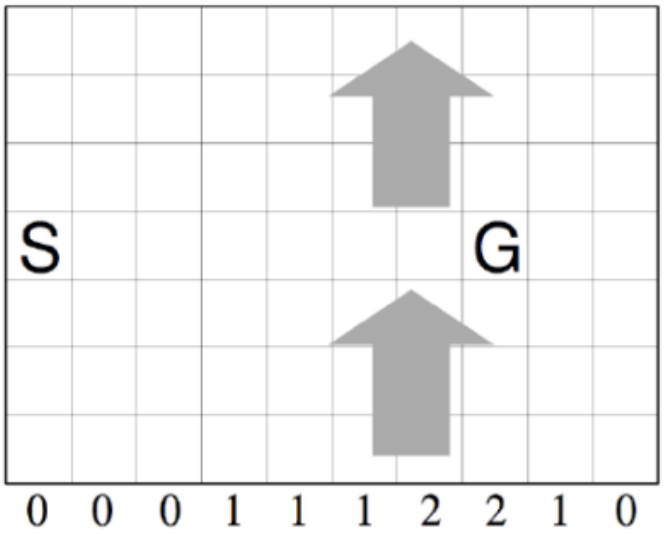
Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$; $a \leftarrow a'$;
  until s is terminal

On-Policy

$(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ Quintupel

SARSA: On-Policy TD

Move from S to G, but consider the crosswind that moves you upward. For example, if you are one cell to the right of the goal, then the action left takes you to the cell just above the goal.



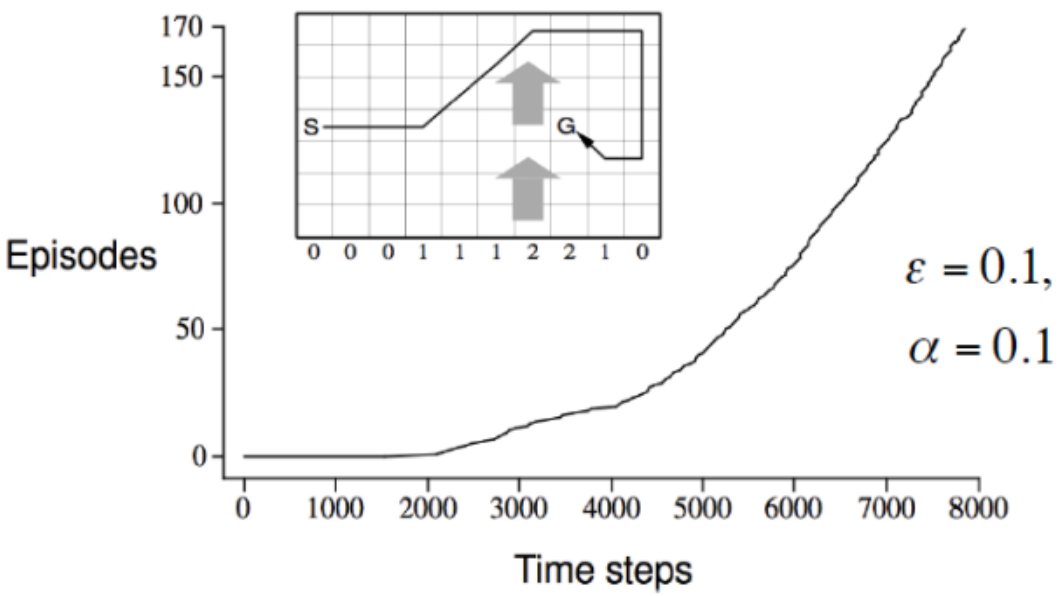
Windy Gridworld

standard moves

strength of the wind

undiscounted, episodic task with constant rewards
reward = –1 until goal

SARSA: On-Policy TD



Results of Sarsa on the Windy Gridworld

Can Monte Carlo methods be used on this task?

No, since termination is not guaranteed for all policies.

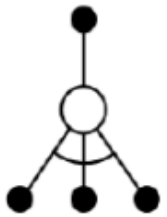
And Sarsa?

Step-by-step learning methods (e.g. Sarsa) do not have this problem. They quickly learn during the episode that such policies are poor, and switch to something else.

Q-learning: Off-Policy TD control

One - step Q - learning :

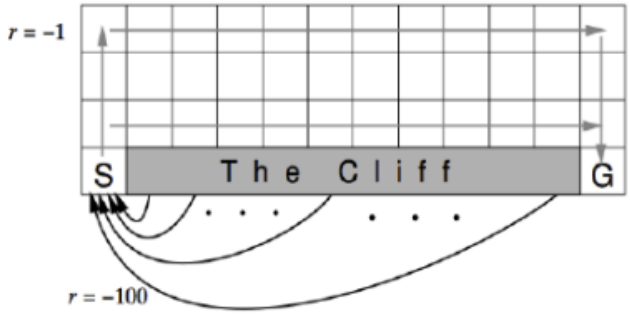
$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right]$$



Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$;
  until s is terminal

Off-Policy

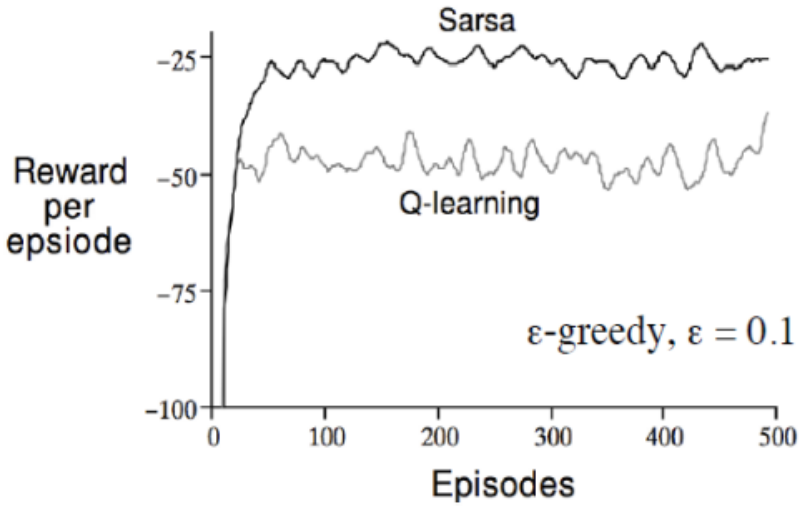
Q-learning and SARSA



safe path

optimal path

Reward is on all transitions -1 except those into the the region marked "The Cliff."



Q-learning learns quickly values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the ε-greedy action selection.

Sarsa takes the action selection into account and learns the longer but safer path through the upper part of the grid.

If ε were gradually reduced, then both methods would asymptotically converge to the optimal policy.

TD Algorithmic Components

- Q-learning:

$$Q_t(s,a) := (1-\alpha_t)Q_{t-1}(s,a) + \alpha_t[r + \gamma \max_{a'} Q_{t-1}(s',a')],$$

  - If infinitely often and $\lim_{T\to\infty}\sum_{t=1}^{T}\alpha_t = \infty \quad \text{and} \quad \lim_{T\to\infty}\sum_{t=1}^{T}\alpha_t^2 < \infty$ then converngence [Jaakkola,Jordan,Singh 94]

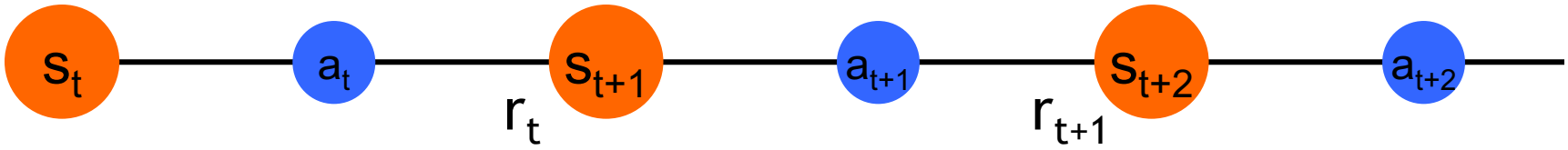
- SARSA(0):

$$Q_t(s,a) := (1-\alpha_t)Q_{t-1}(s,a) + \alpha_t[r + \gamma Q_{t-1}(s',a')],$$

  - Convergence if GLIE policy: infinitely often, in the limit action chosen w.r.t. Q

Sarsa

- again, need Q(s,a), not just V(s)



$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

- control
  - start with a random policy
  - update Q and π after each step
  - need ε-soft policies

SARSA

- SARSA(0) update: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\Big].$
- SARSA algorithm (on-policy):

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $Q(s, a) \leftarrow Q(s, a) + \alpha\big[r + \gamma Q(s', a') - Q(s, a)\big]$
    $s \leftarrow s'$; $a \leftarrow a'$;
  until s is terminal

Decaying vs. Persistent exploration.

GLIE ("greedy in the limit with infinite exploration"):
1. Each action is executed infinitely often in every state that is visited infinitely often;
2. In the limit, the learning policy is greedy with respect to the Q-value function with probability 1.

Q-Learning Algorithmic Components

- **Learning update** (to Q-Table):

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

or

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

- **Action selection** (from Q-Table):

$$a = \operatorname{argmax}_{a} f(Q(s, a))$$

What does an RL-agent do ?

Exploration – Exploitation Dilemma: The agent wants to get as much cumulative reward (also often called *return*) as possible. For this it should always perform the most rewarding action "exploiting" its (learned) knowledge of the state space. This way it might however miss an action which leads (a bit further on) to a much more rewarding path. Hence the agent must also "explore" into unknown parts of the state space. The agent must, thus, balance its policy to include exploitation and exploration.

Policies

1) Greedy Policy: The agent always exploits and selects the most rewarding action. This is sub-optimal as the agent never finds better new paths.

Policies

2) ε-Greedy Policy: With a small probability ε the agent will choose a non-optimal action. *All non-optimal actions are chosen with *equal* probability.* This can take very long as it is not known how big ε should be. One can also "anneal" the system by gradually lowering ε to become more and more greedy.

3) Softmax Policy: ε-greedy can be problematic because of (*). Softmax ranks the actions according to their values and chooses roughly following the ranking using for example:

$$\frac{\exp(\frac{Q_a}{T})}{\sum_{b=1}^{n} \exp(\frac{Q_b}{T})}$$

where Q_a is value of the currently to be evaluated action a and T is a temperature parameter. For large T all actions have approx. equal probability to get selected.

Exploration

Tradeoff between exploitation (control) and exploration (identification)

Extremes: greedy vs. random acting

- Randomly selecting actions is known to give rise to very poor performance.
-ϵ-Greedy, the agent chooses the action with the best long-term
effect with probability ϵ, and it chooses an action uniformly at random, otherwise
(1- ϵ) ϵ is a tuning parameter, which is sometimes changed, either according to a
fixed schedule (making the agent explore less as time goes by),
or adaptively based on some heuristics

Q-learning converges to optimal Q-values if
* Every state is visited infinitely often (due to exploration),
* The action selection becomes greedy as time approaches infinity, and
* The learning rate α is decreased fast enough but not too fast

Exploration

Tradeoff between exploitation (control) and exploration (identification)

Extremes: greedy vs. random acting

Q-learning converges to optimal Q-values if
* Every state is visited infinitely often (due to exploration),
* The action selection becomes greedy as time approaches infinity, and
* The learning rate α is decreased fast enough but not too fast

Exploration

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

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

Exploration

The specific control policy used is a standard one in the field and originally comes from [Watkins, 1989] and [Sutton, 1990]. The agent tries out actions probabilistically based on their Q-values using a Boltzmann or *soft max* distribution. Given a state *x*, it tries out action*a* with probability:

$$p_x(a) = \frac{e^{\frac{Q(x,a)}{T}}}{\sum_{b \in A} e^{\frac{Q(x,b)}{T}}}$$

The *temperature* T controls the amount of exploration (the probability of executing actions other than the one with the highest Q-value). If T is high, or if Q-values are all the same, this will pick a random action. If T is low and Q-values are different, it will tend to pick the action with the highest Q-value.

Exploration

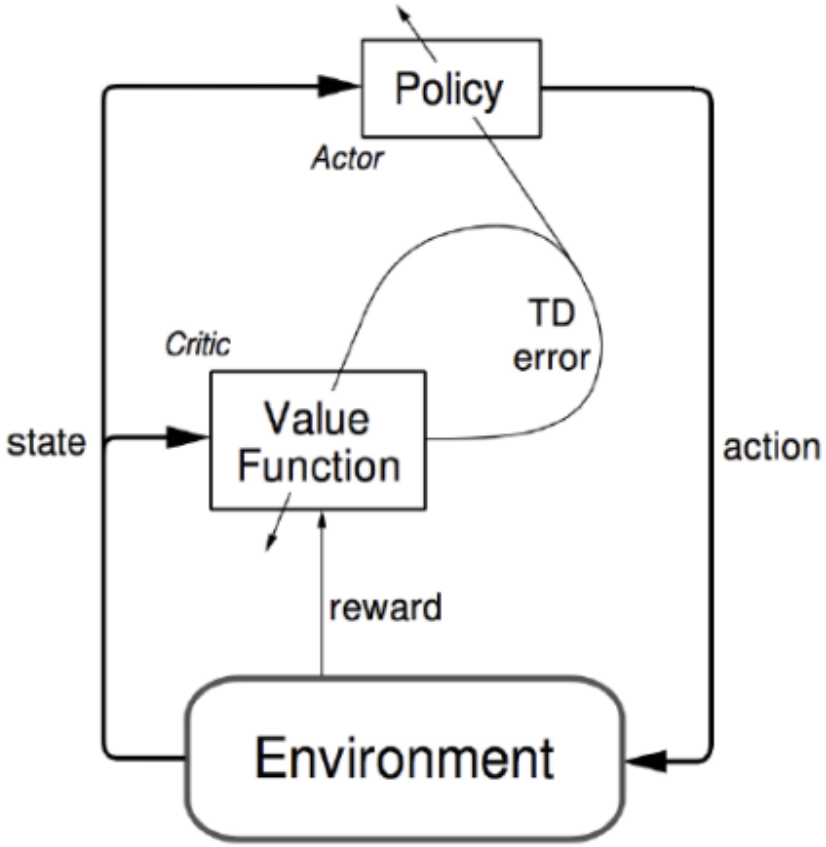
The specific control policy used is a standard one in the field and originally comes from [Watkins, 1989] and [Sutton, 1990]. The agent tries out actions probabilistically based on their Q-values using a Boltzmann or *soft max* distribution. Given a state *x*, it tries out action*a* with probability:

$$p_x(a) = \frac{e^{\frac{Q(x,a)}{T}}}{\sum_{b \in A} e^{\frac{Q(x,b)}{T}}}$$

1. At the start, Q is assumed to be totally inaccurate, so T is high (high exploration), and actions all have a roughly equal chance of being executed.
2. T decreases as time goes on. It becomes more and more likely to pick among the actions with the higher Q-values.
3. Finally, as we assume Q is converging, T approaches zero (pure exploitation) and we tend to only pick the action with the highest Q-value:

$$p_x(a) = \begin{cases} 1 & \text{if } Q(x,a) = \max_{b \in A} Q(x,b) \\ 0 & \text{otherwise} \end{cases}$$

Actor Critic Methods



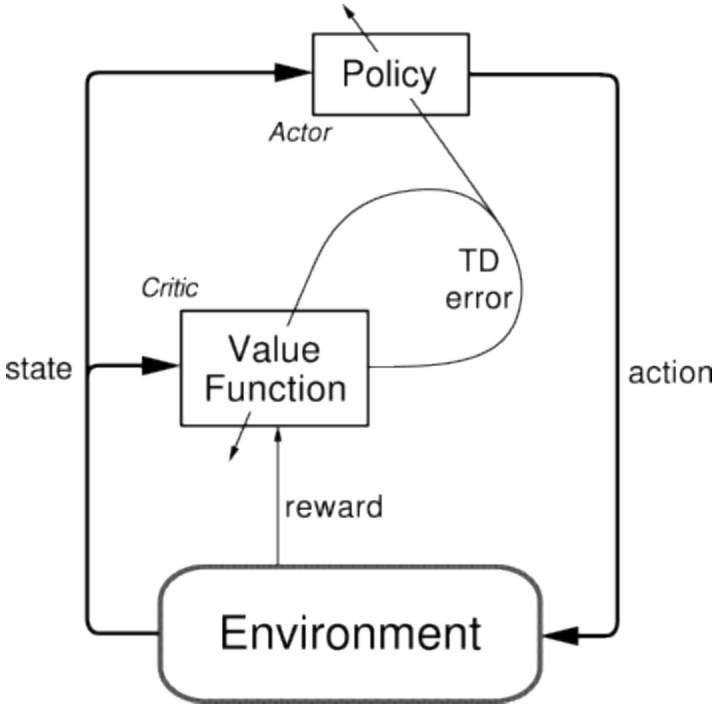
- Explicit representation of policy as well as value function
- Critic drives all learning
- On policy method
- Appealing as psychological and neural models

Separate structure to explicitly represent the policy. The policy is the *actor* (used to select actions). The estimated value function is the *critic*. Learning is on-policy: the critic must learn about and critique whatever policy is currently being followed by the actor. The critique is a scalar: TD error.

Actor Critic Methods

- Policy and value function
- After each action selection, the critic evaluates the new state to determine whether things have gone better or worse than expected.
- That evaluation is the TD error:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t),$$



- TD error can be used to evaluate the action just selected:

$$\pi_t(s,a) = Pr\{a_t = a \mid s_t = s\} = \frac{e^{p(s,a)}}{\sum_b e^{p(s,b)}}, \qquad p(s_t,a_t) \leftarrow p(s_t,a_t) + \beta\delta_t,$$

Another variant is $p(s_t,a_t) \leftarrow p(s_t,a_t) + \beta\delta_t\Big[1 - \pi_t(s_t,a_t)\Big].$

Actor Critic Methods

Typically, the critic is a state-value function. After each action selection, the critic evaluates the new state to determine whether things have gone better or worse than expected. That evaluation is the TD error:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

Let's assume actions are determined by preferences, p(*s*,*a*), as follows:

$$\pi_t(s,a) = \Pr\{a_t = a \mid s_t = s\} = \frac{e^{p(s,a)}}{\sum_b e^{p(s,b)}},$$

Then strengthening or weakening the preferences, p(*s*,*a*), depends on the TD error (β – step size parameter):

$$p(s_t,a_t) \leftarrow p(s_t,a_t) + \beta\delta_t$$

Another variant is $p(s_t,a_t) \leftarrow p(s_t,a_t) + \beta\delta_t\left[1 - \pi_t(s_t,a_t)\right].$

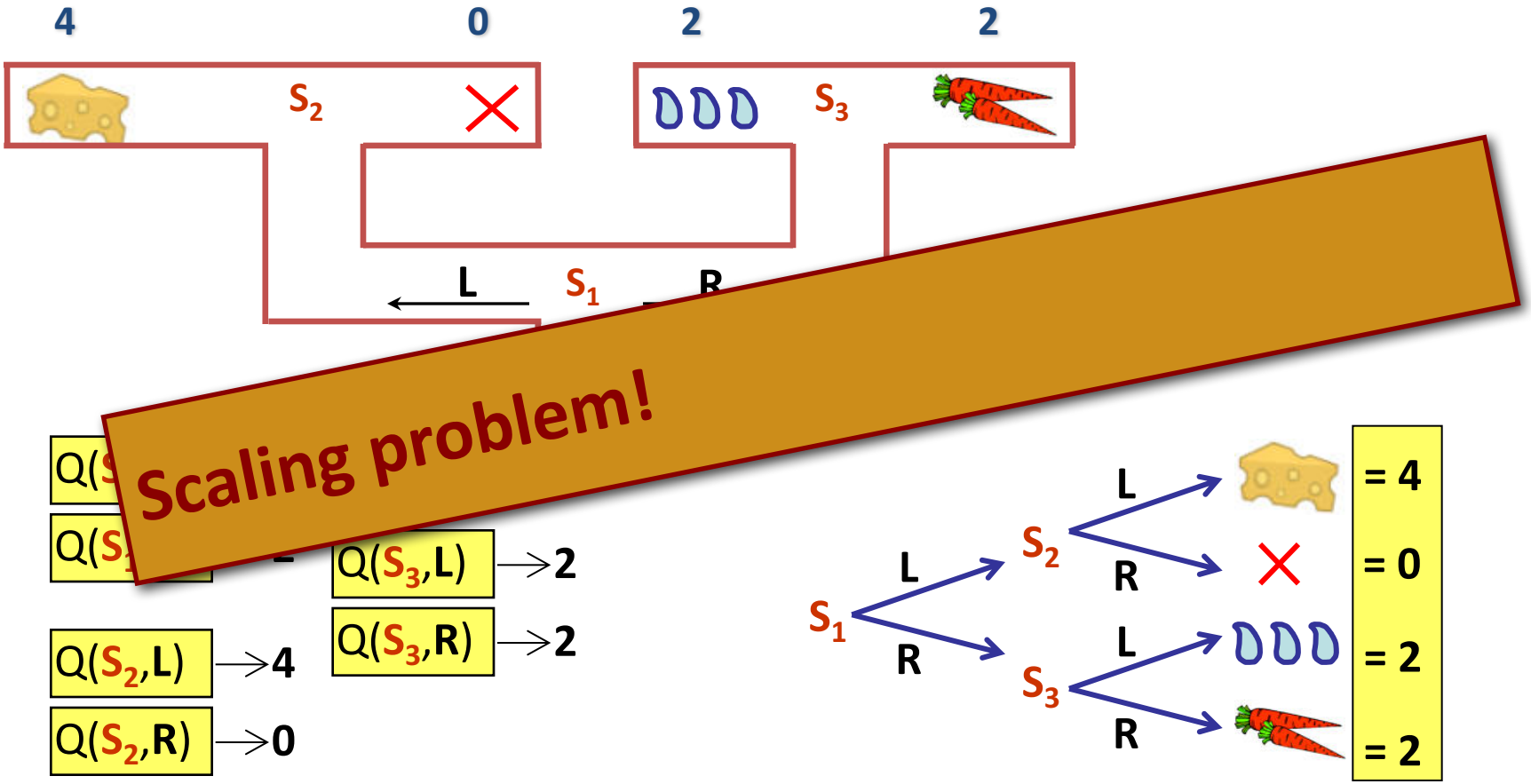
Applications of RL

- Checker's [Samuel 59]
- TD-Gammon [Tesauro 92]
- World's best downpeak elevator dispatcher [Crites at al ~95]
- Inventory management [Bertsekas et al ~95]
  - 10-15% better than industry standard
- Dynamic channel assignment [Singh & Bertsekas, Nie&Haykin ~95]
  - Outperforms best heuristics in the literature
- Cart-pole [Michie&Chambers 68-] with bang-bang control
- Robotic manipulation [Grupen et al. 93-]
- Path planning
- Robot docking [Lin 93]
- Parking
- Football
- Tetris
- Multiagent RL [Tan 93, Sandholm&Crites 95, Sen 94-, Carmel&Markovitch 95-, lots of work since]
- Combinatorial optimization: maintenance & repair
  - Control of reasoning [Zhang & Dietterich IJCAI-95]

Proprietà

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

RL in real world tasks...



model based vs. model free learning and control

Problems of RL

Curse of Dimensionality
In real world problems ist difficult/impossible to define discrete state-action spaces.

(Temporal) Credit Assignment Problem
RL cannot handle large state action spaces as the reward gets too much dilited along the way.

Partial Observability Problem
In a real-world scenario an RL-agent will often not know exactly in what state it will end up after performing an action. Furthermore states must be history independent.

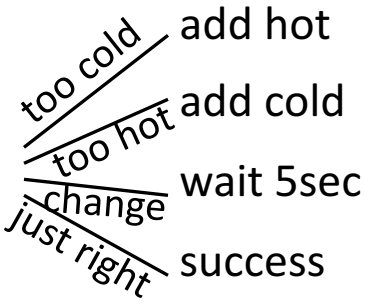
State-Action Space Tiling
Deciding about the actual state- and action-space tiling is difficult as it is often critical for the convergence of RL-methods. Alternatively one could employ a continuous version of RL, but these methods are equally difficult to handle.

Non-Stationary Environments
As for other learning methods, RL will only work quasi stationary environments.

Real-world behavior is hierarchical

1. pour coffee
2. add sugar
3. add milk
4. stir

1. set water temp
2. get wet
3. shampoo
4. soap
5. turn off water
6. dry off



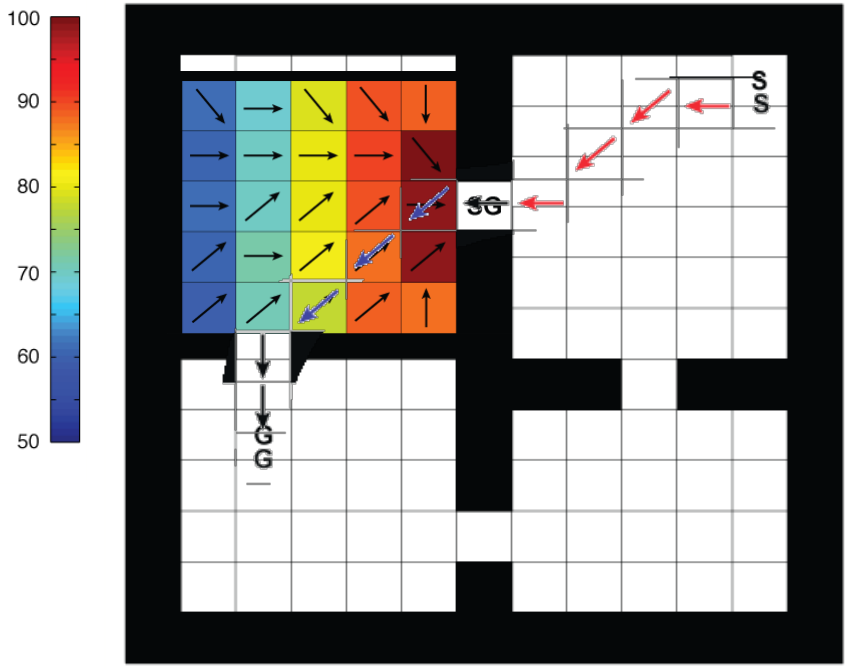
Hierarchical RL: What is it?

simplified control, disambiguation, encapsulation

Hierarchical Reinforcement Learning

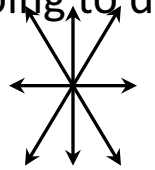
- Exploits domain structure to facilitate learning
  - Policy constraints
  - State abstraction
- Paradigms: Options, HAMs, MaxQ
- MaxQ task hierarchy
  - Directed acyclic graph of subtasks
  - Leaves are the primitive MDP actions
- Traditionally, task structure is provided as prior knowledge to the learning agent

HRL: a toy example



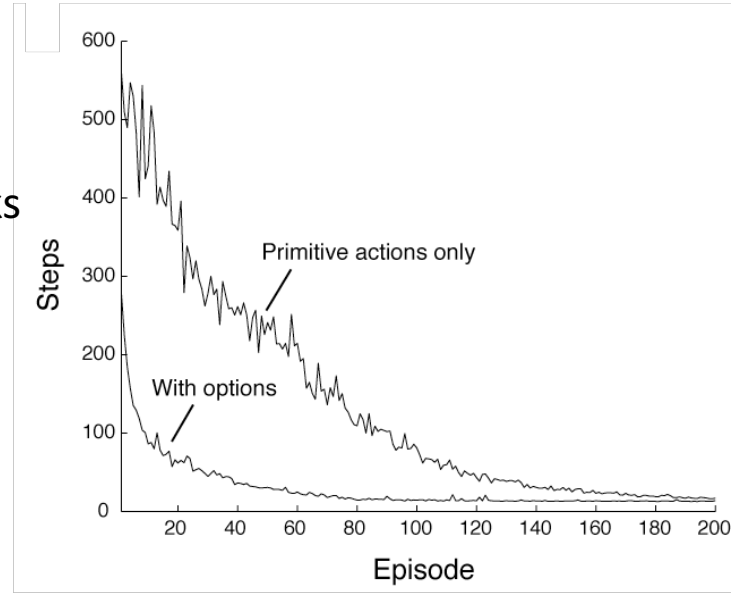
S: start G: goal

Options: going to doors

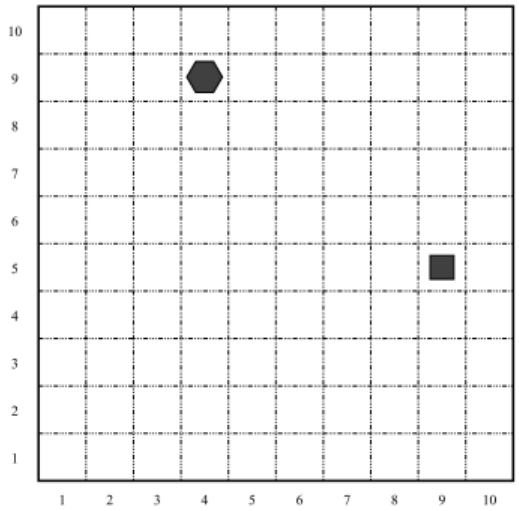


Advantages of HRL

1. Faster learning
   (mitigates scaling problem)
2. *Transfer of knowledge* from previous tasks
   (generalization, shaping)



Example 1: Hierarchical Distance to Goal (Kaelbling)

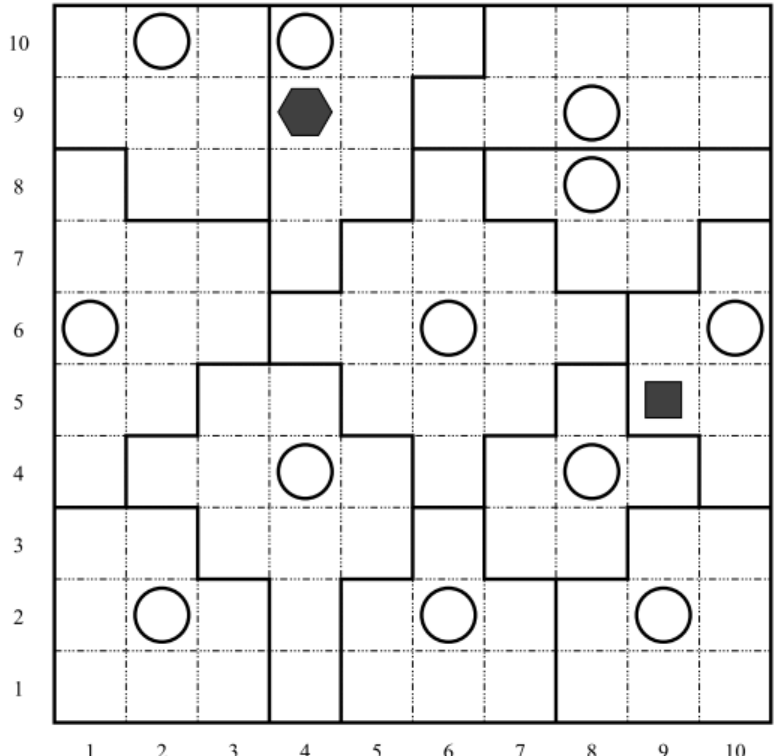


- **States:** 10,000 combinations of location of agent and location of goal.
- **Actions:** North, South, East, West. Each action succeeds with probability 0.8 and fails (moving perpendicularly) with probability 0.2.
- **Reward Function:** Cost of 1 unit for each action until goal is reached.

alue function requires representing 10,000 values (or 40,000 values using Q learning).

Example 1: Decomposition Strategy

- **Impose a set of landmark states**. Partitions the state space into Voronoi cells.
- **Constrain the policy**. The policy will go from the starting cell via a series of landmarks until it reaches the landmark in the goal cell. From there, it will go to the goal.
- **Decompose the value function**.



Example 1: Formulation of Subtasks

Subtasks:

- GotoLmk(x, l): Go from current location x to landmark l, where l is the landmark defining the current cell or any of its neighboring cells. $[V_1(x, l)]$
- LmktoLmk(l_1, l_2): Go from landmark l_1 to landmark l_2. $[V_2(l_1, l_2)]$ (uses GotoLmk as a subroutine)
- GotoGoal(x, g): Go from current location x to the goal location g (within current cell). $[V_3(x, g)]$

The cost of getting to the goal is now the cost of getting from x to one of the neighbor landmarks l_1, then from l_1 to the landmark in the goal cell l_g, and then from l_g to the goal g:

$$V(x, g) = min_{l \in N(NL(x))}[V_1(x, l) + V_2(l, NL(g)) + V_3(NL(g), g)]$$

This requires only 6,070 values to store as a set of Q functions (compared to 40,000 for the original problem).

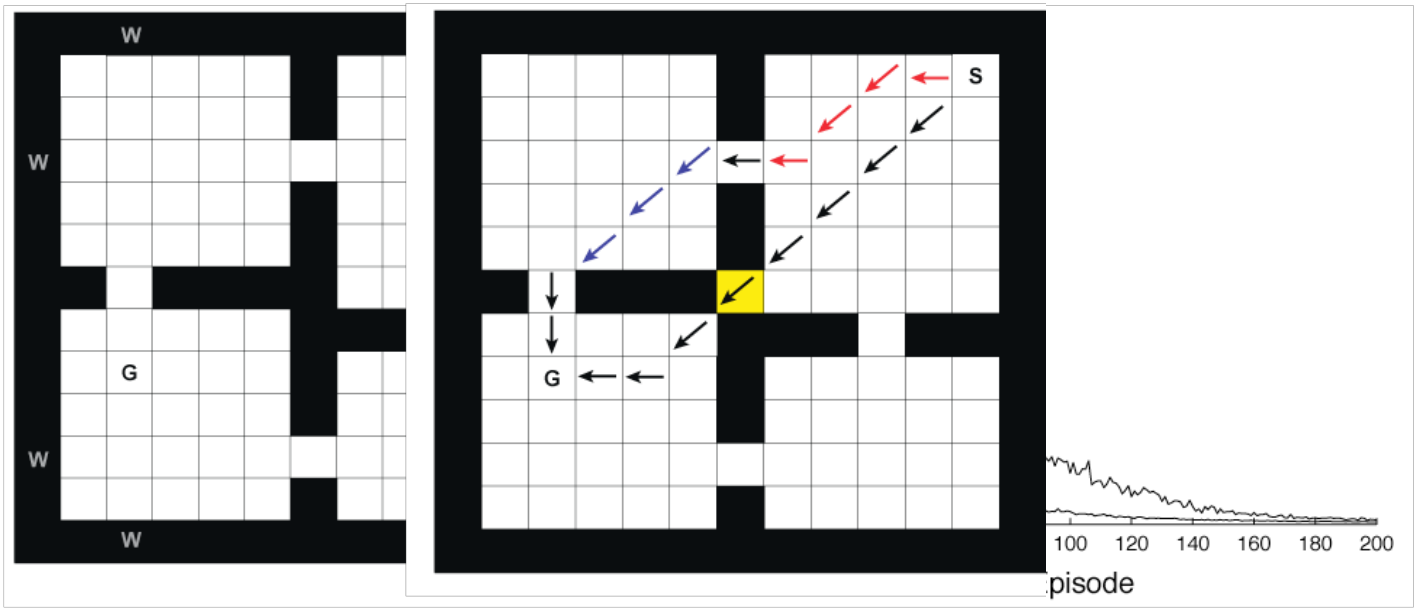
Each of these subtasks can be shared by many combinations of initial states and goal states.

Solution

- **Solve all possible** GotoLmk **and** GotoGoal **subtasks.**
- **Solve** LmktoLmk **task using** GotoLmk **as a subroutine.**
- **Choose actions by using combined value function.**

Disadvantages (or: the cost) of HRL

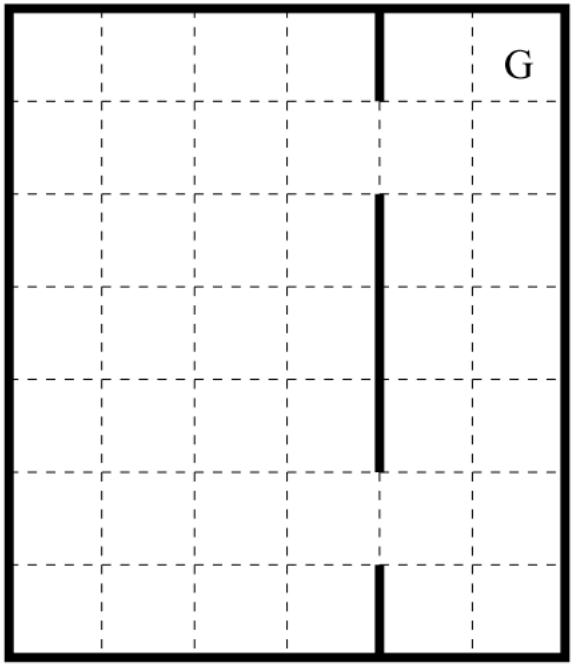
1. Need ‘right’ options - how to learn them?
2. Suboptimal behavior (“negative transfer”; habits)
3. More complex learning/control structure



no free lunches…

Example problem

- **actions:** North, South, East, West
- **rewards:** Each action costs -1. Goal gives reward of 0.



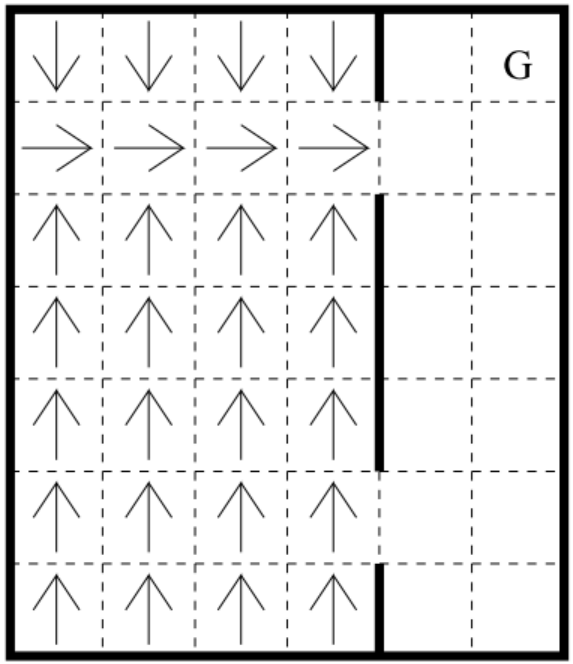
Options (Sutton; Precup; Singh)

An option is a macro action defined as follows:

- **A region of the state space**. The option can begin execution from any state in this region.
- **A policy** π. This tells for all states in the problem, what action the option will perform.
- **A termination condition**. This tells, for each state, the probability that the option will terminate if it enters that state.

Example: "Exit room by upper door"

- **Initiation region:** Any point in left room.
- **Policy:** See figure.
- **Termination condition:** Terminate with probability 1 in any state outside the room; 0 inside the room.



Partial Policies (Parr and Russell)

Partial Policy

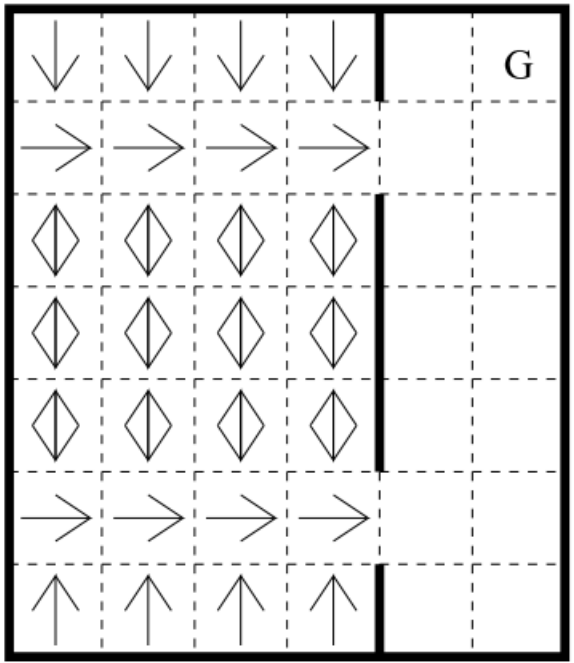
- **Mapping from states to sets of possible actions**.

  Example:

  $\pi(s_1) = \{\mathsf{South}\}$

  $\pi(s_2) = \{\mathsf{North}, \mathsf{South}\}$

Only need to learn what to do when the partial policy lists more than one possible action to perform.



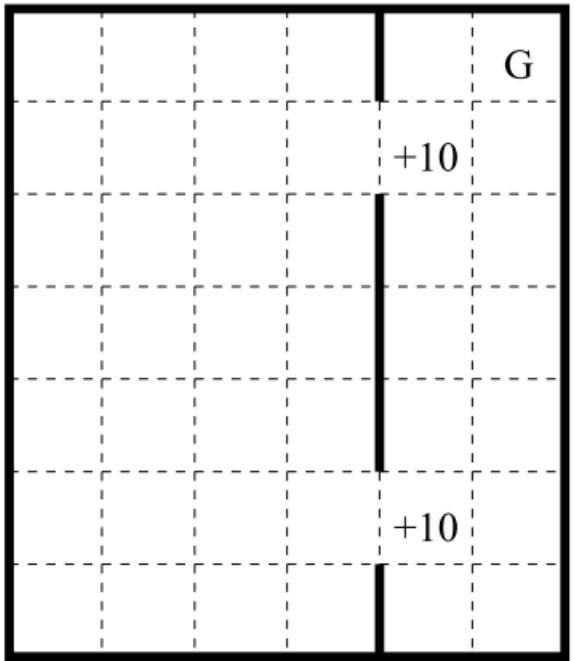
Subtasks

A subtask is defined by

- **A region of state space**. The subtask is only "active" within this region.
- **A termination condition**. This indicates when the subtask has completed execution.
- **A pseudo-reward function**. This determines the value of each of the terminal states.

Example: Exit by nearest door

- **Region:** Left room.
- **Termination condition:** Outside left room.
- **Pseudo-reward:** 0 inside left room; +10 in both "boundary states".



Semi-Markov Decision Process

- Generalizes MDPs
- Action **a** takes **N** steps to complete in **s**
- P(**s'**,**n** | **a**, **s**), R(**s'**, **N** | **a**, **s**)
- Bellman equation:

$$V^{\pi}(s) = \sum_{s',N} P(s', N|s, \pi(s)) \left[R(s', N|s, \pi(s)) + \gamma^{N} V^{\pi}(s') \right].$$

$$V^{\pi}(s) = \overline{R}(s, \pi(s)) + \sum_{s',N} P(s', N|s, \pi(s)) \gamma^{N} V^{\pi}(s').$$

Task 1: Learning a policy over options

Basic idea: Treat each option as a primitive action.

Complication: The actions take different amounts of time.

Fundamental Observation: MDP + options = Semi-Markov Decision Problem (Parr)

Semi-Markov Q learning (Bradke/Duff, Parr)

- **In s, choose option a and perform it**
- **Observe resulting state s', reward r, and number of time steps N**
- **Perform the following update:**

$$Q(s,a) := (1-\alpha)Q(s,a) + \alpha \cdot [r + \gamma^N \max_{a'} Q(s',a')]$$

Under suitable conditions, this will converge to the best possible policy definable over the options.

Observations

- **Only need to learn Q values for a subset of the states:** All possible initial states. All states where some option could terminate.
- **Learned policy may not be optimal**.
  The optimal policy may not be representable by some combination of given options. For example, if we only have the option Exit-by-nearest-door, then this will give a suboptimal result for states one move above the level of the lower door.
- **If $\gamma = 1$ (no discounting), Semi-Markov Q learning = ordinary Q learning**
- **Model-based algorithms are possible.** The model must predict the probability distribution over the possible result states (and the expected rewards that will be received).

Learning with partial policies

- **Basic Idea: Execute the partial policy until it reaches a "choice state"** (i.e., a state with more than one possible action)
- **This defines a Semi-MDP**
  - **States:** all initial states and all choice state
  - **Actions:** actions given by $\pi(s)$
  - **Reward:** sum of rewards until next choice state
- **Apply Semi-Markov Q learning**

  Converges to best possible refinement of given policy.

Hierarchies of Abstract Machines (HAM)

Parr extended the partial policy idea to work with hierarchies of partial policies. Within a partial policy, an action can be any of:

- **Primitive action**
- **Call another partial policy as a subroutine**
- RETURN (return to caller)

Convert the hierarchy into a flat SMDP:

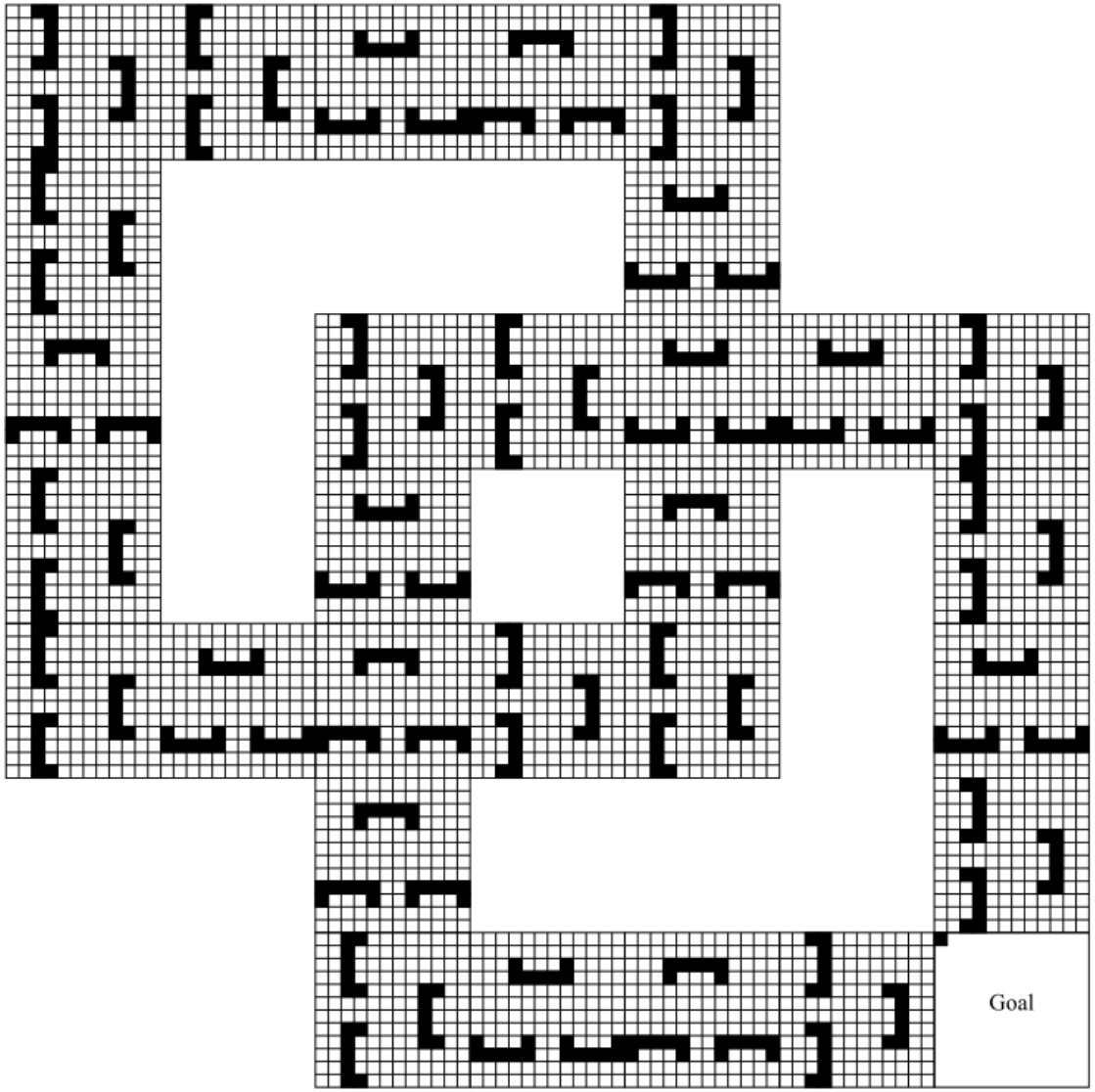
- **States:** pairs of [state, call-stack] pairs
- **Actions:** as dictated by partial policy
- **Reward function:** same as original reward function.

Apply SMDP Q learning

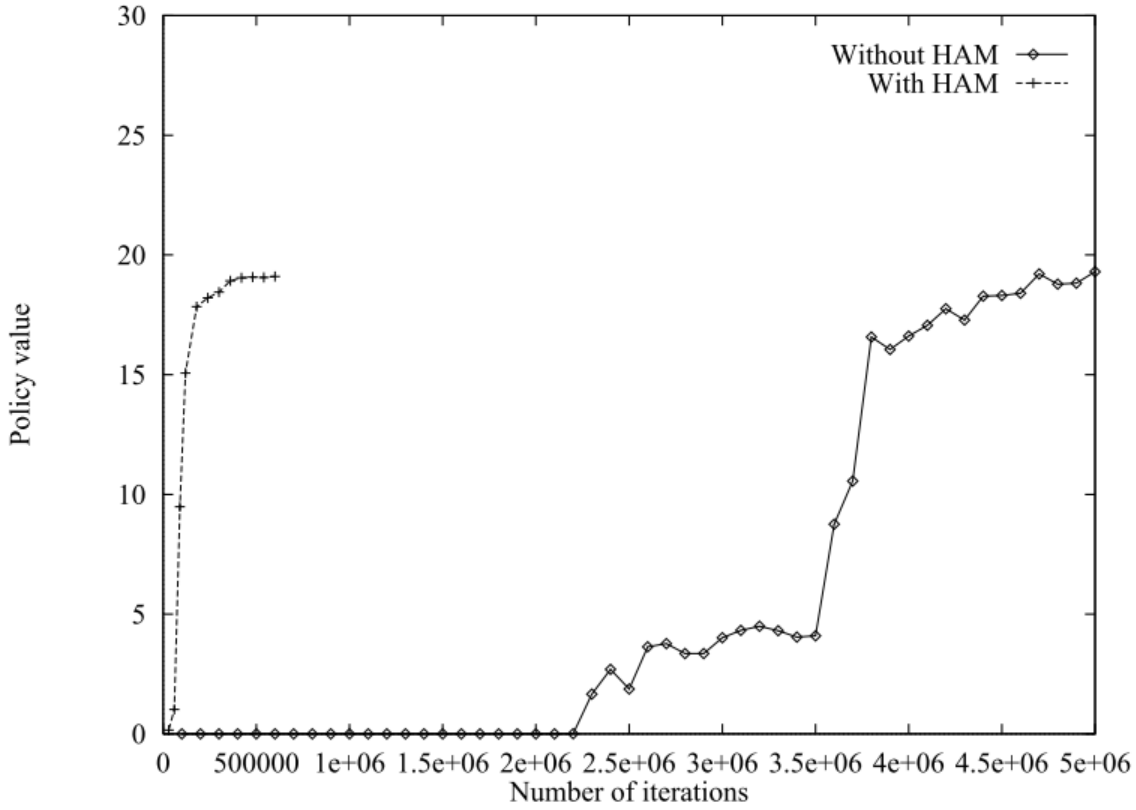
Task 2: Example: Parr's Maze Problem

artial Policies

- TraverseHallway(d)
  calls ToWallBouncing and BackOut.
- ToWallBouncing(d_1, d_2)
  calls ToWall, FollowWall
- FollowWall(d)
- ToWall(d)
- BackOut(d_1, d_2)
  calls BackOne, PerpFive
- BackOne(d)
- PerpFive(d_1, d_2)



Task 2: Results (Parr)



Value of starting state. Flat Q learning ultimately gives a better policy.

Learn policies for a given set of sub-tasks

Each subtask is an MDP

- **States:** All non-terminated states.
- **Actions:** The primitive actions in the domain.
- **Reward:** Sum of original reward function and pseudo-reward function.

Learning hierarchical sub-tasks

- **At state s inside subtask i, choose child subtask j and invoke it (recursively)**
- **When it returns, observe resulting state s', total reward r, and number of time steps N**

$$Q(i,s,j) := (1-\alpha)Q(i,s,j) + \alpha[r + \tilde{R}_i(s') + \gamma^N \max_{a'} Q(i,s',a')]$$

If each subtask executes a GLIE policy (Greedy in the Limit with Infinite Exploration), then this will converge (Dietterich).

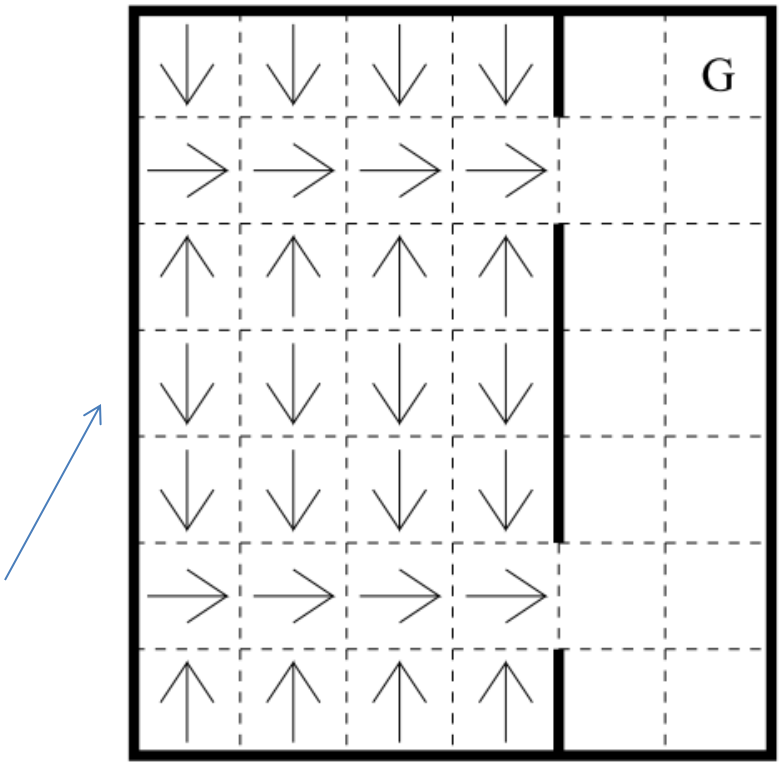
However, it converges to only a *locally optimal* policy.

Hierarchical Optimality versus Recursive Optimality

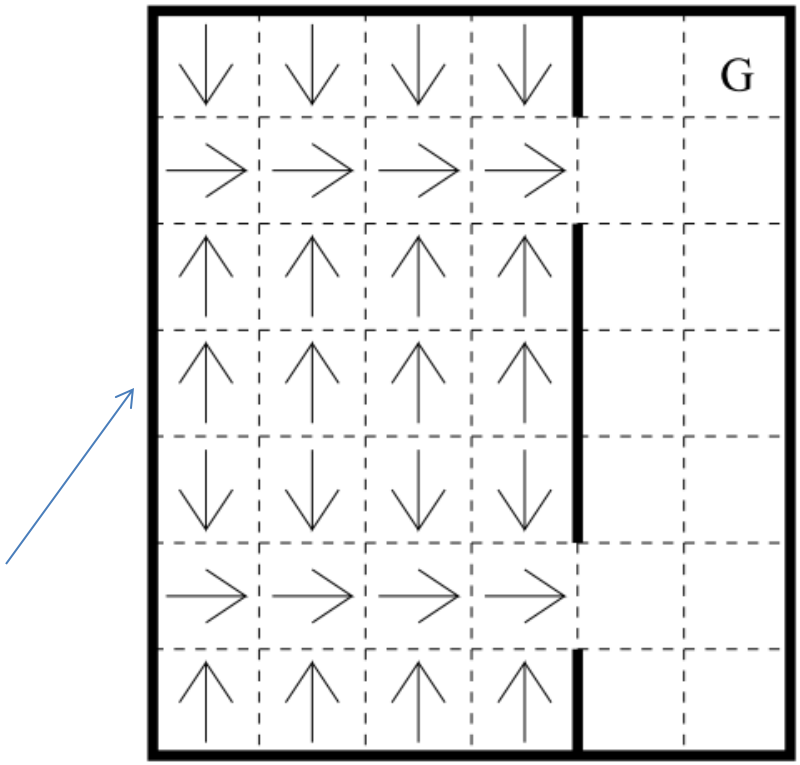
- **Hierarchical Optimality: The overall learned policy is the best policy consistent with the hierarchy**
- **Recursive Optimality: The policy for a task is optimal *given* the policies learned by its children**

- **Parr's partial policy method learns hierarchically optimal policies**. Information about the value of the result states can propagate "into" the subproblem.
- **Hierarchical SMDP Q learning converges to a recursively optimal policy**. Information about the value of result states is blocked from flowing "into" the subtask.

Example



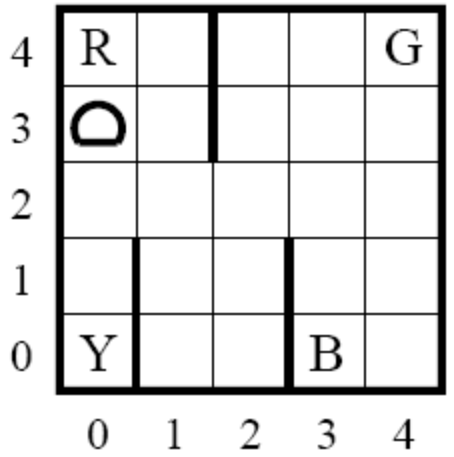
Locally optimal

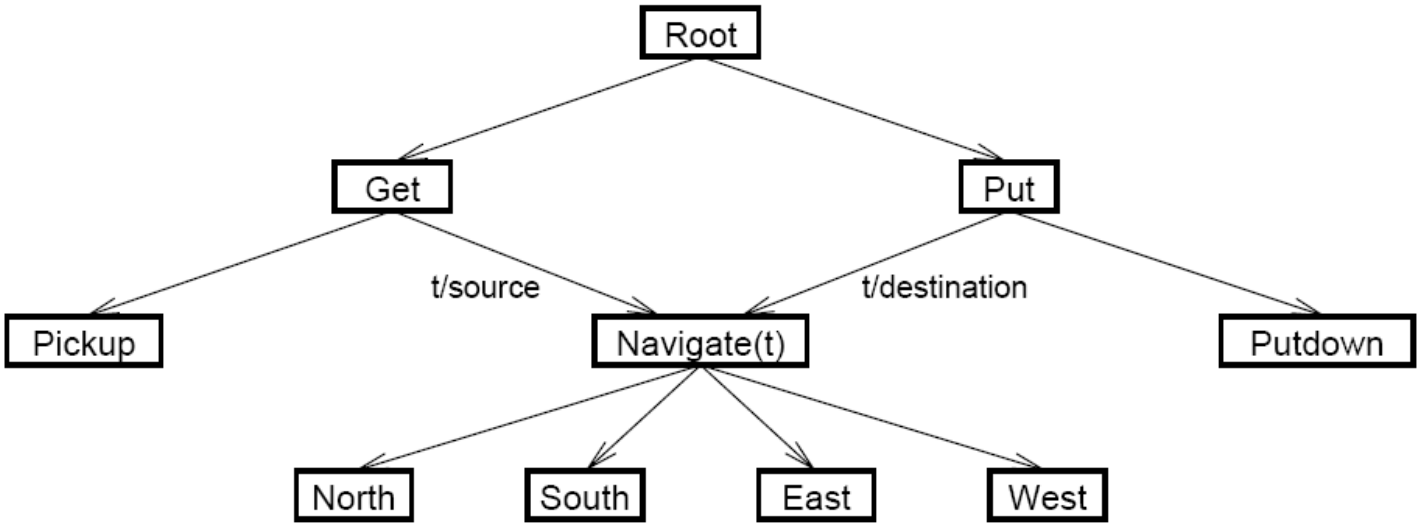


Optimal for the entire task

Taxi Domain

- Motivational Example
- Reward: -1 actions,
  -10 illegal, 20 mission.
- 500 states
- Task Graph:





HSMQ Alg. (Task Decomposition)

function HSMQ(state s, subtask p) **returns** float
    Let $TotalReward = 0$
    while p is not terminated **do**
        Choose action $a = \pi_x(s)$ according to exploration policy π_x
        Execute a.
            if a is primitive, Observe one-step reward r
            else $r := HSMQ(s, a)$, which invokes subroutine a and
                returns the total reward received while a executed.
        $TotalReward := TotalReward + r$
        Observe resulting state s'
        Update $Q(p, s, a) := (1 - \alpha)Q(p, s, a) + \alpha \left[r + \max_{a'} Q(p, s', a') \right]$
    end // while
    return $TotalReward$
end

MAXQ

- Break original MDP into multiple sub-MDP's
- Each sub-MDP is treated as a temporally extended action
- Define a hierarchy of sub-MDP's (sub-tasks)

- Each sub-task M_i defined by:
  - T = Set of terminal states
  - A_i = Set of child actions (may be other sub-tasks)
  - R'_i = Local reward function

MAXQ Alg. (Value Fun. Decomposition)

- Want to obtain some sharing (compactness) in the representation of the value function.
- Re-write Q(p, s, a) as

$$Q(p, s, a) = V(a, s) + C(p, s, a)$$

$$V(p, s) = \max_a \left[V(a, s) + C(p, s, a) \right]$$

where *V(a, s)* is the expected total reward while executing action a, and *C(p, s, a)* is the expected reward of completing parent task *p* after *a* has returned

Hierarchical Structure

- MDP decomposed in task M0, … , Mn

Theorem 1 *Given a task graph over tasks* $M_0, \ldots, M_n$ *and a hierarchical policy* π, *each subtask* M_i *defines a semi-Markov decision process with states* S_i, *actions* A_i, *probability transition function* $P_i^\pi(s', N|s, a)$, *and* expected *reward function* $\overline{R}(s, a) = V^\pi(a, s)$, *where* $V^\pi(a, s)$ *is the projected value function for child task* M_a *in state* s. *If* a *is a primitive action,* $V^\pi(a, s)$ *is defined as the expected immediate reward of executing* a *in* s: $V^\pi(a, s) = \sum_{s'} P(s'|s, a)R(s'|s, a)$.

- Q for the subtask i

$$Q^\pi(i, s, a) = V^\pi(a, s) + \sum_{s', N} P_i^\pi(s', N|s, a)\gamma^N Q^\pi(i, s', \pi(s')),$$

$$Q^\pi(i, s, a) = V^\pi(a, s) + C^\pi(i, s, a).$$

Value Decomposition

Definition 6 *The* completion function, $C^\pi(i,s,a)$, *is the expected discounted cumulative reward of* completing *subtask* M_i *after invoking the subroutine for subtask* M_a *in state* s. *The reward is discounted back to the point in time where* a *begins execution.*

$$C^\pi(i,s,a) = \sum_{s',N} P_i^\pi(s',N|s,a)\gamma^N Q^\pi(i,s',\pi(s')) \tag{9}$$

With this definition, we can express the Q function recursively as

$$Q^\pi(i,s,a) = V^\pi(a,s) + C^\pi(i,s,a). \tag{10}$$

Finally, we can re-express the definition for $V^\pi(i,s)$ as

$$V^\pi(i,s) = \begin{cases} Q^\pi(i,s,\pi_i(s)) & \text{if } i \text{ is composite} \\ \sum_{s'} P(s'|s,i)R(s'|s,i) & \text{if } i \text{ is primitive} \end{cases} \tag{11}$$

MAXQ Alg.

- An example

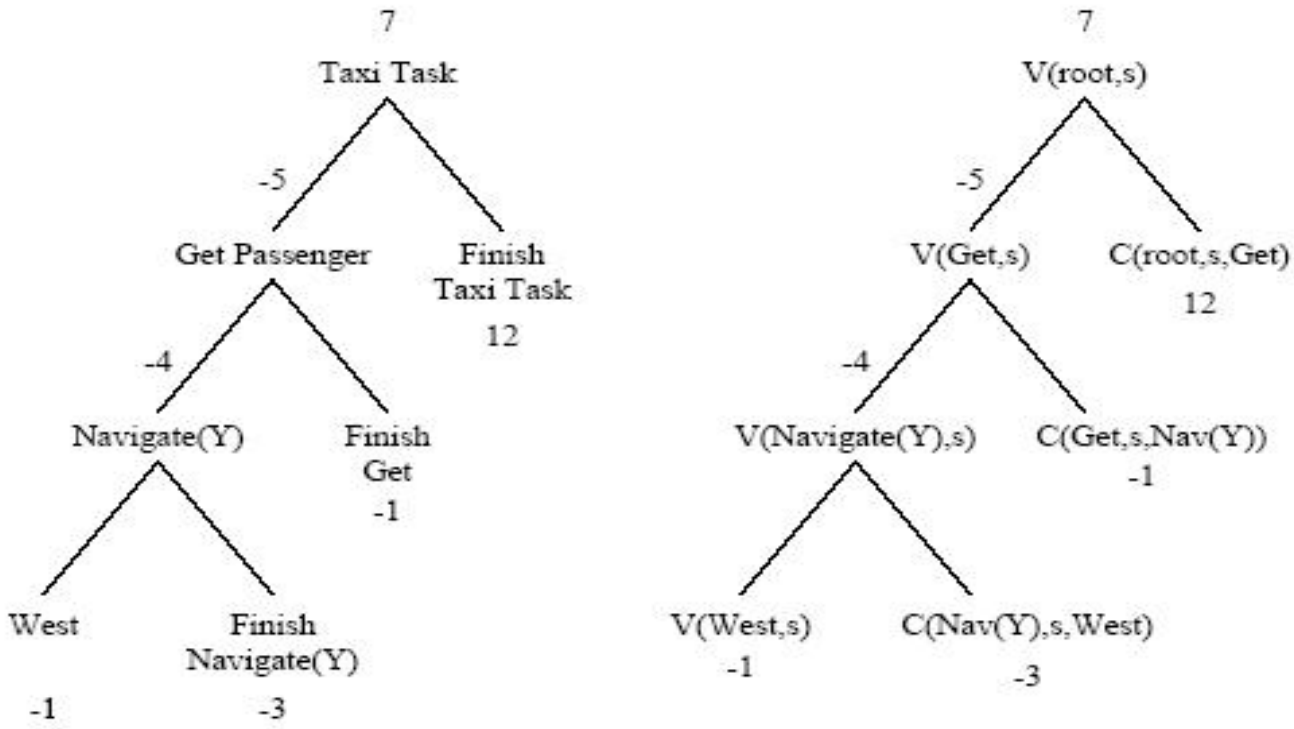


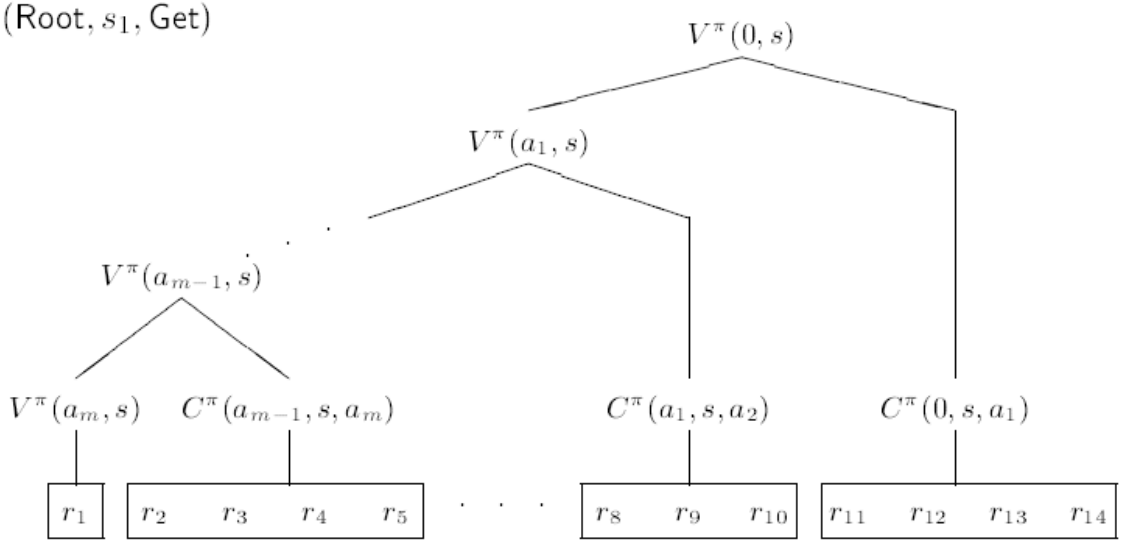
Fig. 5. An example of the MAXQ value function decomposition for the state in which the taxi is at location (2,2), the passenger is at (0,0), and wishes to get to (3,0). The left tree gives English descriptions, and the right tree uses formal notation.

Value Decomposition

- The value function can be decomposed as follows

$$V^{\pi}(0,s) = V^{\pi}(a_m,s) + C^{\pi}(a_{m-1},s,a_m) + \ldots + C^{\pi}(a_1,s,a_2) + C^{\pi}(0,s,a_1)$$

$$\begin{aligned} V^{\pi}(\text{Root},s_1) &= V^{\pi}(\text{North},s_1) + C^{\pi}(\text{Navigate}(R),s_1,\text{North}) + \\ &\quad C^{\pi}(\text{Get},s_1,\text{Navigate}(R)) + C^{\pi}(\text{Root},s_1,\text{Get}) \\ &= -1+0+-1+12 \\ &= 10 \end{aligned}$$



MAXQ Alg. (cont'd)

$$V(\text{root}, s) = V(\text{west}, s) + C(\text{navigate}(Y), s, \text{west}) + C(\text{get}, s, \text{navigate}(Y)) + C(\text{root}, s, \text{get}).$$

Passenger at Y

	0	1	2	3	4
4	10	9	8	7	6
3	11	10	9	8	7
2	12	11	10	9	8
1	13	10	9	8	7
0	14	9	8	7	6

Passenger In Taxi

	0	1	2	3	4
4	G	18	13	12	11
3	18	17	14	13	12
2	17	16	15	14	13
1	16	15	14	13	12
0	15	14	13	12	11

Fig. 4. Value function for the case where the passenger is at (0,0) (location Y) and wishes to get to (0,4) (location R).

MAXQ Alg. (cont'd)

function MAXQQ(state s, subtask p) **returns** float
    Let $TotalReward = 0$
    while p is not terminated **do**
        Choose action $a = \pi_x(s)$ according to exploration policy π_x
        Execute a.
            if a is primitive, Observe one-step reward r
            else $r := MAXQQ(s, a)$, which invokes subroutine a and
                returns the total reward received while a executed.
        $TotalReward := TotalReward + r$
        Observe resulting state s'
        if a is a primitive
            $V(a, s) := (1 - \alpha)V(a, s) + \alpha r$
        else a is a subroutine
            $C(p, a, s) := (1 - \alpha)C(p, s, a) + \alpha \max_{a'} [V(a', s') + C(p, s', a')]$
    end // **while**
    return $TotalReward$
end

State Abstraction

Three fundamental forms

- Irrelevant variables

  e.g. passenger location is irrelevant for the **navigate** and **put** subtasks and it thus could be ignored.

- Funnel abstraction

  A funnel action is an action that causes a larger number of initial states to be mapped into a small number of resulting states. E.g., the ***navigate(t)*** action maps any state into a state where the taxi is at location *t*. This means the completion cost is independent of the location of the taxi—it is the same for all initial locations of the taxi.

State Abstraction (cont'd)

- Structure constraints

- E.g. if a task is terminated in a state s, then there is no need to represent its completion cost in that state

- Also, in some states, the termination predicate of the child task implies the termination predicate of the parent task

Effect

- reduce the amount memory to represent the Q-function.

  14,000 q values required for flat Q-learning

  3,000 for HSMQ (with the irrelevant-variable abstraction

  632 for C() and V() in MAXQ

- learning faster

State Abstraction (cont'd)

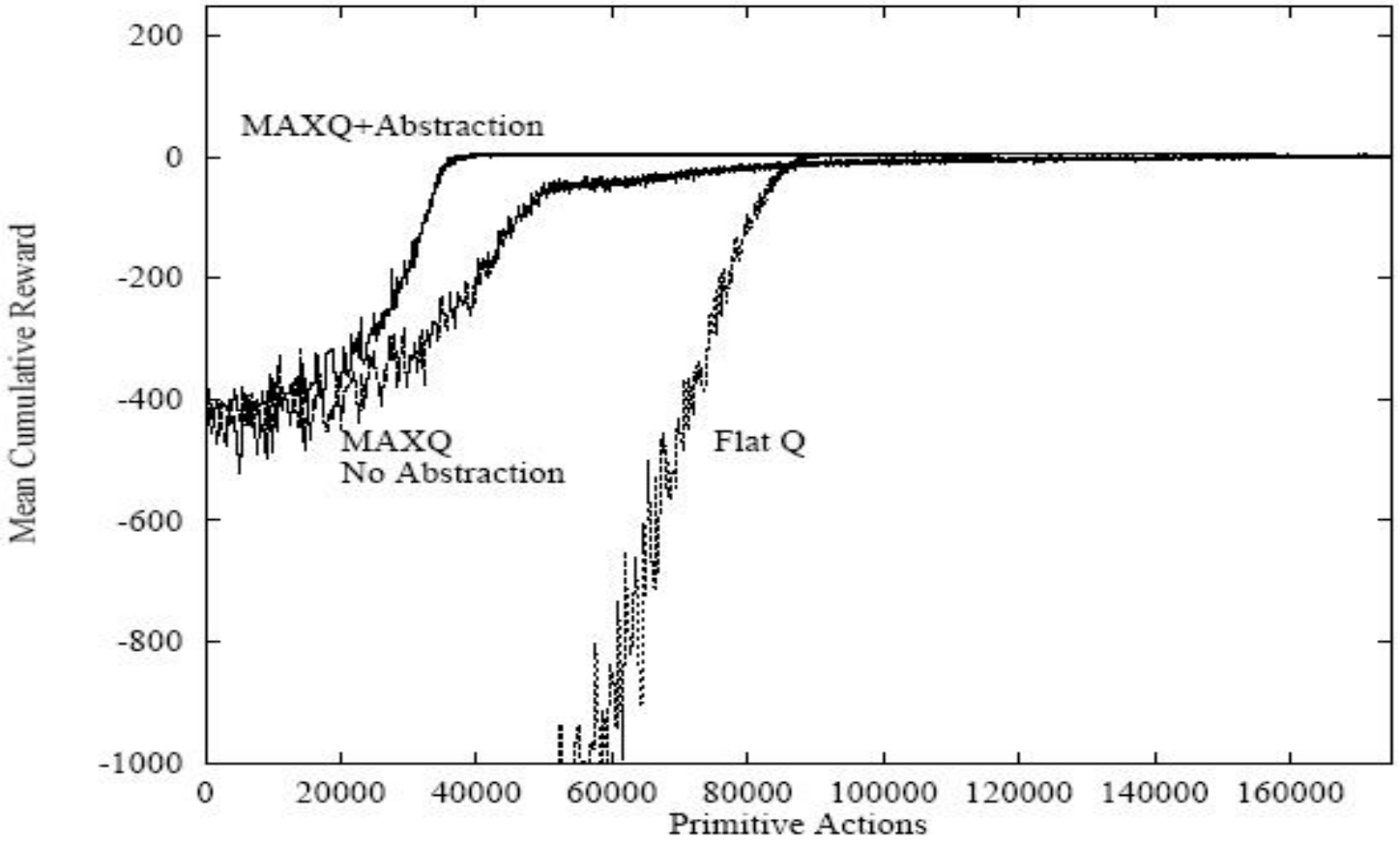
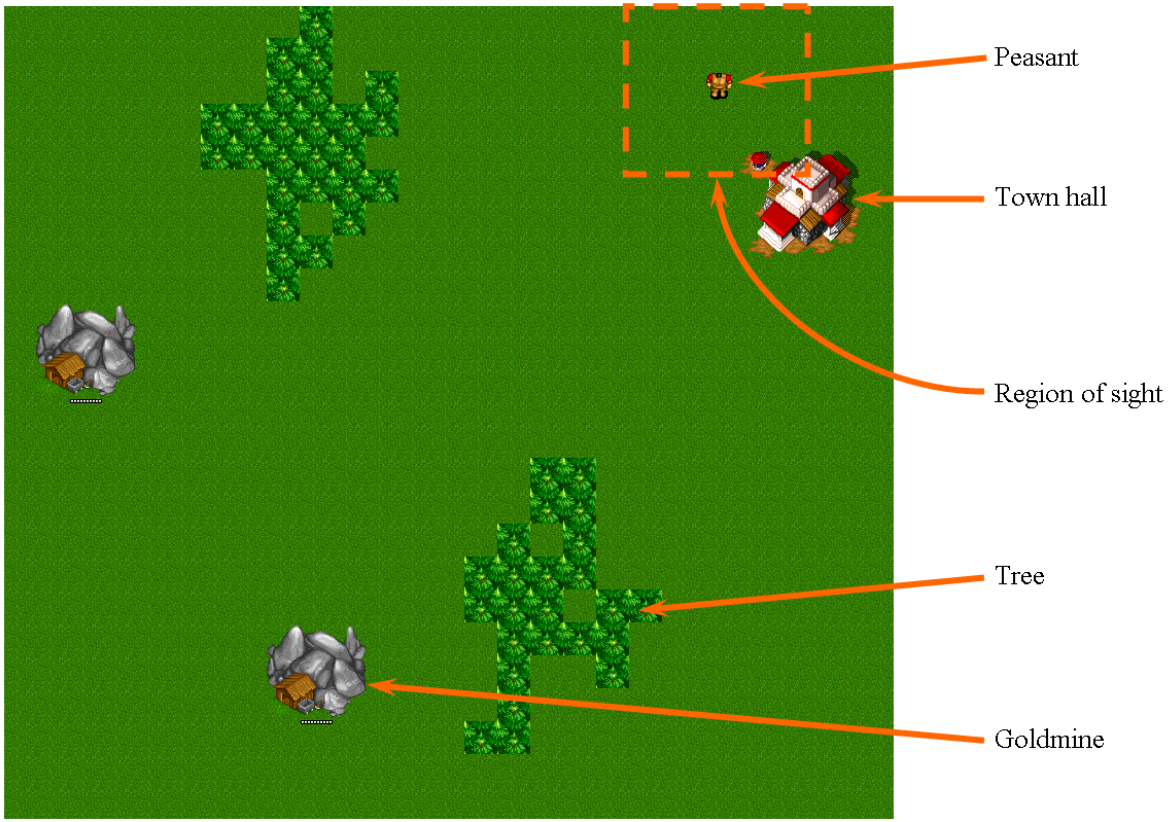


Fig. 7. Comparison of Flat Q learning, MAXQ Q learning with no state abstraction, and MAXQ Q learning with state abstraction on a noisy version of the taxi task.

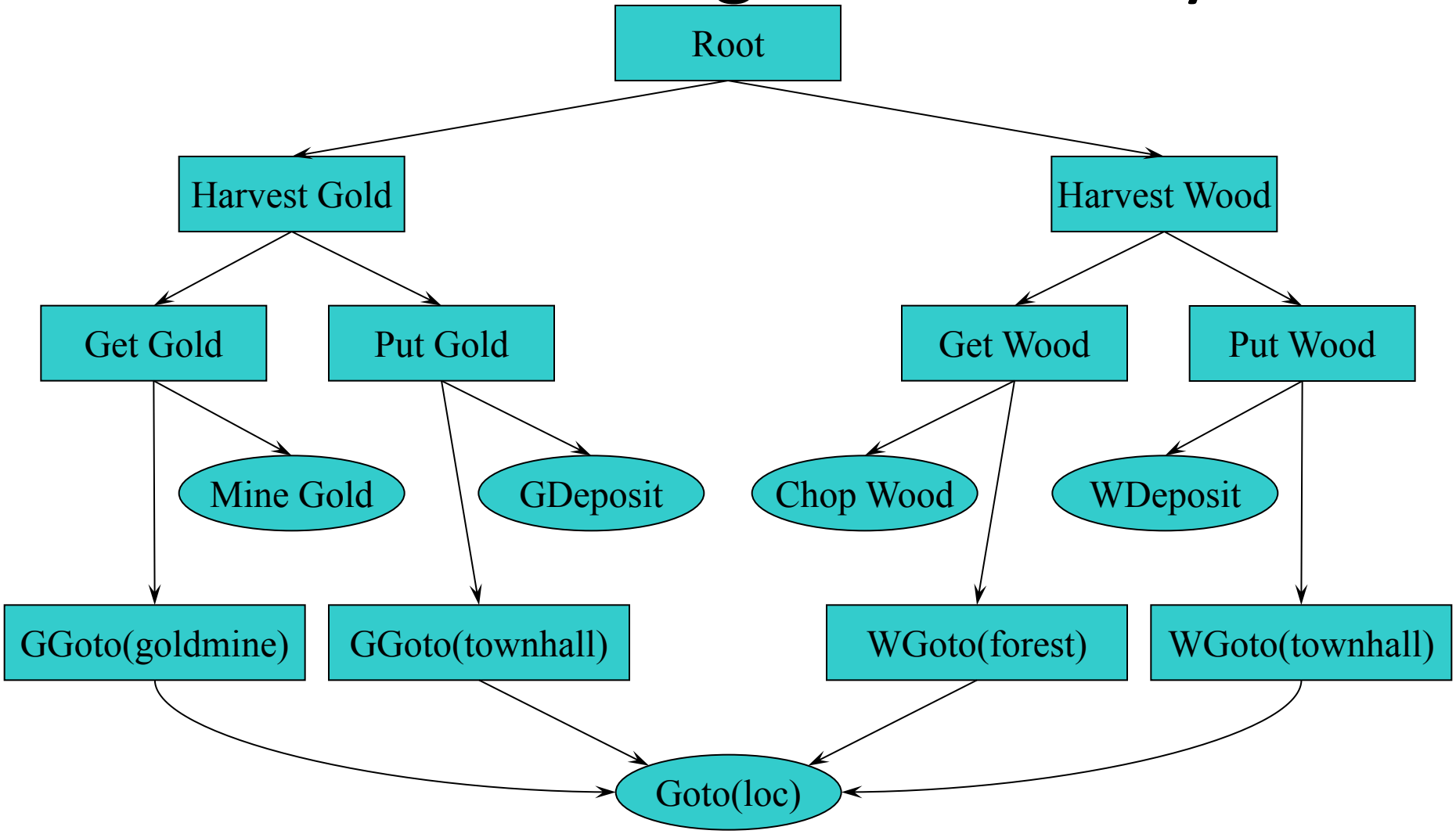
Wargus Resource-Gathering Domain



State variables
Peasant location: `a.l`
Peasant resource: `a.r`
Gold mine within sight radius: `reg.gold`
Trees within sight radius: `reg.wood`
Town hall within sight radius: `reg.townhall`
Required gold quota: `req.gold`
Required wood quota: `req.wood`

Primitive actions
Mine gold: `MG`
Chop wood: `CW`
Deposit: `Dep`
Navigate: `Goto(loc)`

Induced Wargus Hierarchy



Induced Abstraction & Termination

Task Name	State Abstraction	Termination Condition
Root	req.gold, req.wood	req.gold = 1 && req.wood = 1
Harvest Gold	req.gold, agent.resource, region.townhall	req.gold = 1
Get Gold	agent.resource, region.goldmine	agent.resource = gold
Put Gold	req.gold, agent.resource, region.townhall	agent.resource = 0
GGoto(goldmine)	agent.x, agent.y	agent.resource = 0 && region.goldmine = 1
GGoto(townhall)	agent.x, agent.y	req.gold = 0 && agent.resource = gold && region.townhall = 1
Harvest Wood	req.wood, agent.resource, region.townhall	req.wood = 1
Get Wood	agent.resource, region.forest	agent.resource = wood
Put Wood	req.wood, agent.resource, region.townhall	agent.resource = 0
WGoto(forest)	agent.x, agent.y	agent.resource = 0 && region.forest = 1
WGoto(townhall)	agent.x, agent.y	req.wood = 0 && agent.resource = wood && region.townhall = 1
Mine Gold	agent.resource, region.goldmine	NA
Chop Wood	agent.resource, region.forest	NA
GDeposit	req.gold, agent.resource, region.townhall	NA
WDeposit	req.wood, agent.resource, region.townhall	NA
Goto(loc)	agent.x, agent.y	NA

Note that because each subtask has a unique terminal state,
Result Distribution Irrelevance applies

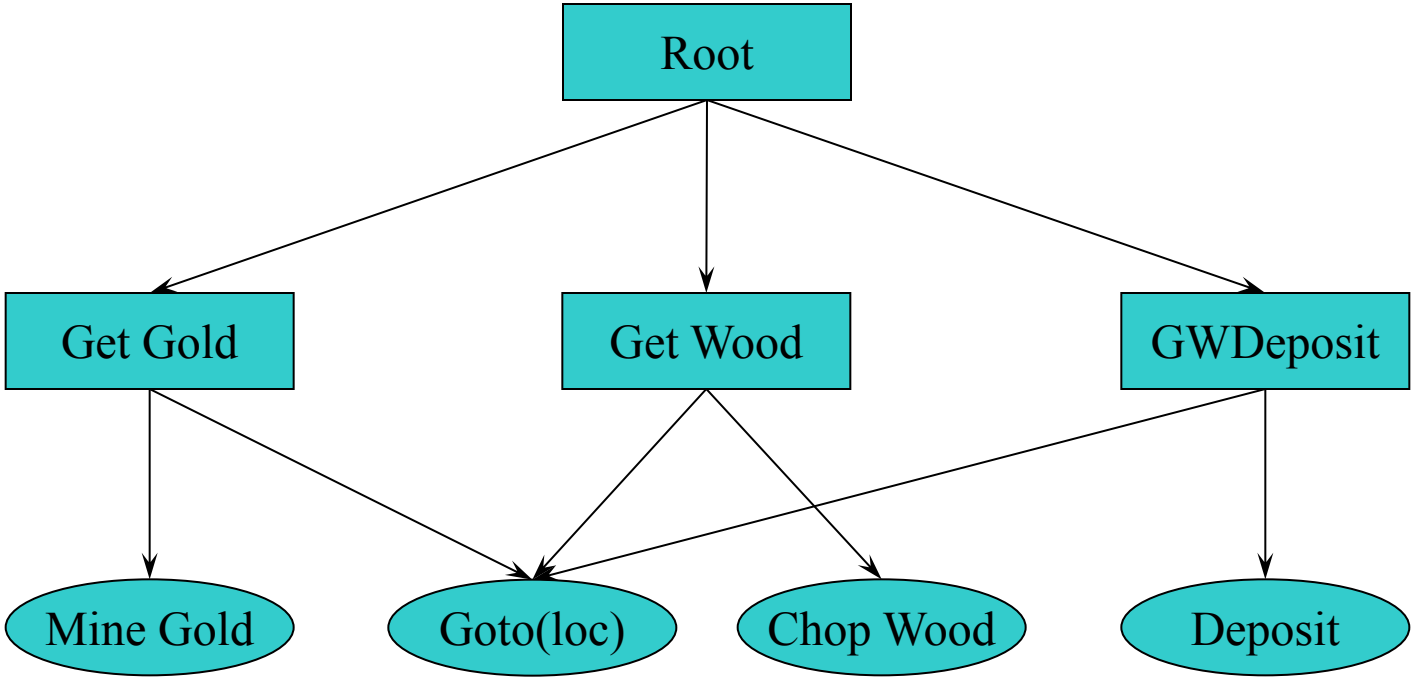
Claims

- The resulting hierarchy is unique
  - Does not depend on the order in which goals and trajectory sequences are analyzed
- All state abstractions are safe
  - There exists a hierarchical policy within the induced hierarchy that will reproduce the observed trajectory
  - Extend MaxQ Node Irrelevance to the induced structure
- Learned hierarchical structure is “locally optimal”
  - No local change in the trajectory segmentation can improve the state abstractions (very weak)

Experimental Setup

- Randomly generate pairs of source-target resource-gathering maps in Wargus
- Learn the optimal policy in source
- Induce task hierarchy from a single (near) optimal trajectory
- Transfer this hierarchical structure to the MaxQ value-function learner for target
- Compare to direct Q learning, and MaxQ learning on a manually engineered hierarchy within target

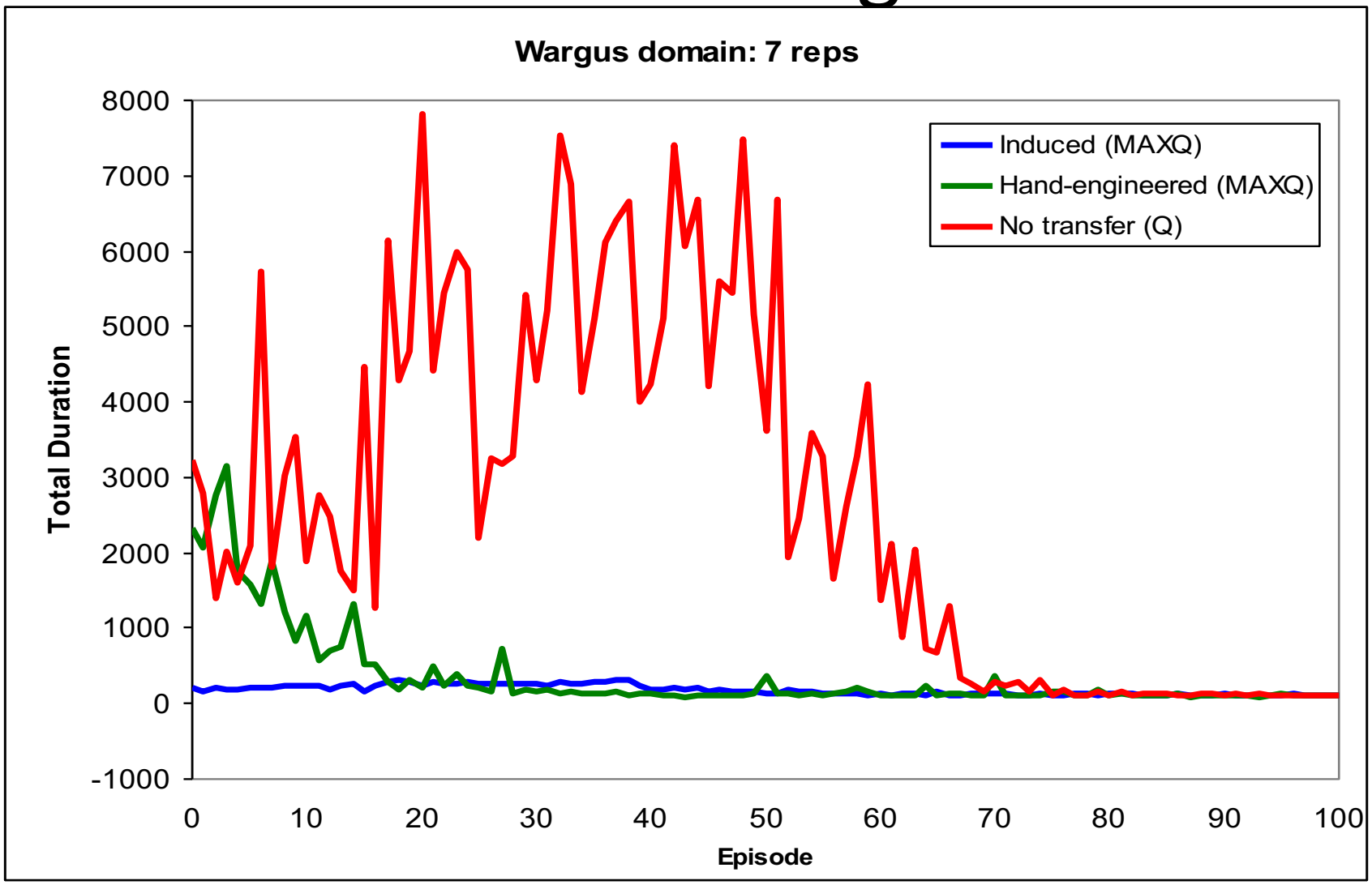
Hand-Built Wargus Hierarchy



Hand-Built Abstractions & Terminations

Task Name	State Abstraction	Termination Condition
Root	req.gold, req.wood, agent.resource	req.gold = 1 && req.wood = 1
Harvest Gold	agent.resource, region.goldmine	agent.resource ≠ 0
Harvest Wood	agent.resource, region.forest	agent.resource ≠ 0
GWDeposit	req.gold, req.wood, agent.resource, region.townhall	agent.resource = 0
Mine Gold	region.goldmine	NA
Chop Wood	region.forest	NA
Deposit	req.gold, req.wood, agent.resource, region.townhall	NA
Goto(loc)	agent.x, agent.y	NA

Results: Wargus



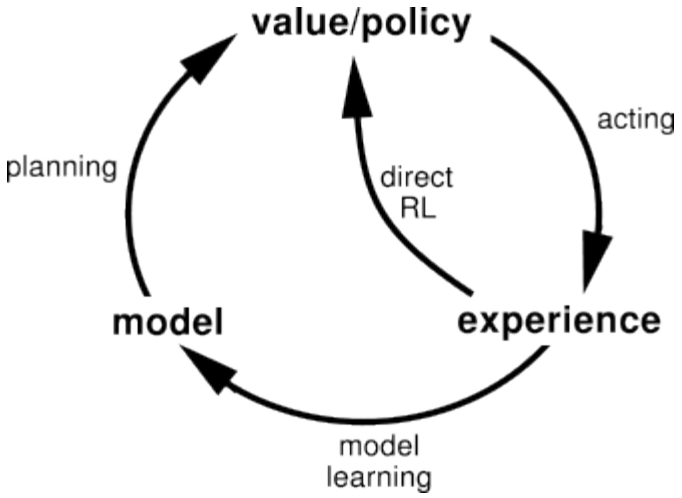
Limitations

- Recursively optimal not necessarily optimal
- Model-free Q-learning

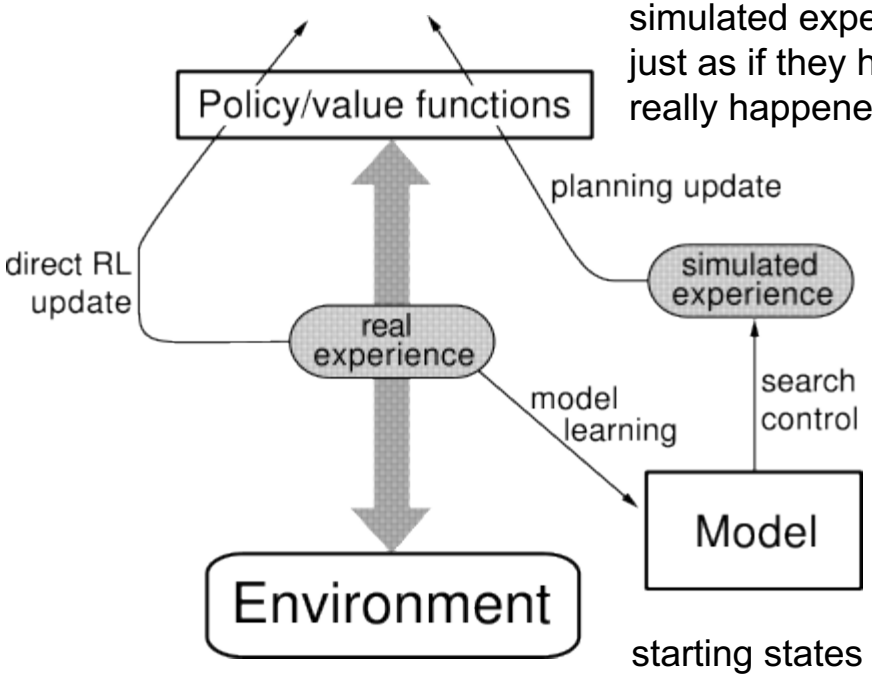
Model-based algorithms (that is, algorithms that try to learn $P(s'|s,a)$ and $R(s'|s,a))$ are generally much more efficient because they remember past experience rather than having to re-experience it.

Planning, Acting, Learning

- On-line planning
- RL Learning
- Dyna-Q



RL methods to the simulated experiences just as if they had really happened



starting states and actions for the simulated experiences generated by the model

The reinforcement learning method is thus the "final common path" for both learning and planning

Planning, Acting, Learning

- Dyna-Q alg.

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:

(a) $s \leftarrow$ current (nonterminal) state

(b) $a \leftarrow \varepsilon$-greedy(s,Q)

(c) Execute action a; observe resultant state, s', and reward, r

(d) $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$

(e) $Model(s,a) \leftarrow s', r$ (assuming deterministic environment)

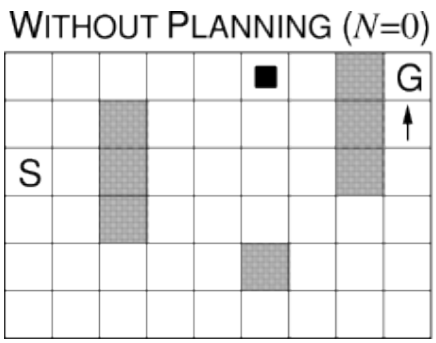
(f) Repeat N times:

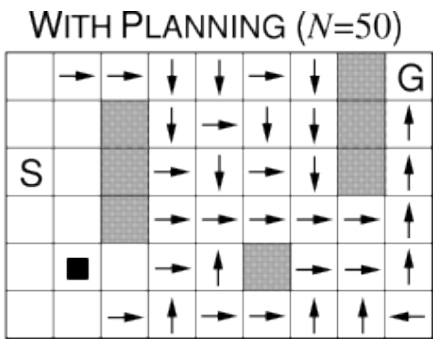
$s \leftarrow$ random previously observed state

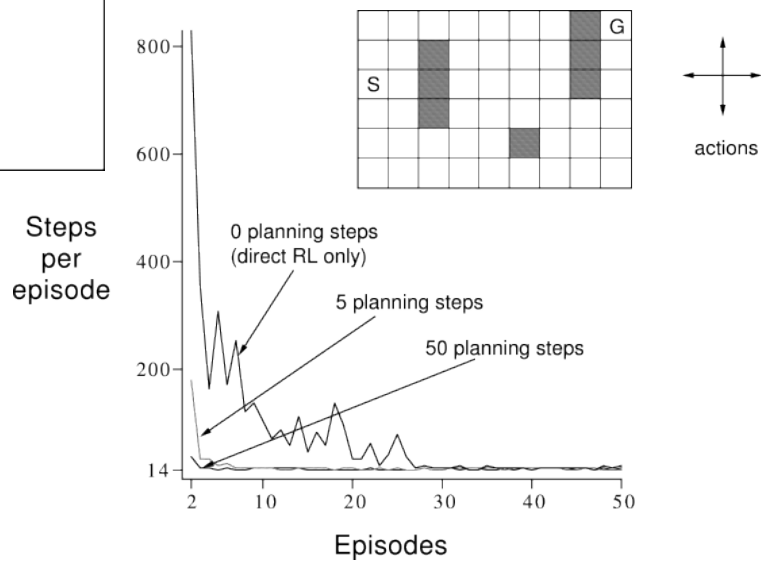
$a \leftarrow$ random action previously taken in s

$s', r \leftarrow Model(s,a)$

$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$







References and Further Reading

- Sutton, R., Barto, A., (2000) *Reinforcement Learning: an Introduction*, The MIT Press
  http://www.cs.ualberta.ca/~sutton/book/the-book.html

- Kaelbling, L., Littman, M., Moore, A., (1996) Reinforcement Learning: a Survey, *Journal of Artificial Intelligence Research*, **4**:237-285

- Barto, A., Mahadevan, S., (2003) Recent Advances in Hierarchical Reinforcement Learning, *Discrete Event Dynamic Systems: Theory and Applications*, **13**(4):41-77