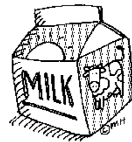
Real-world behavior is hierarchical

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

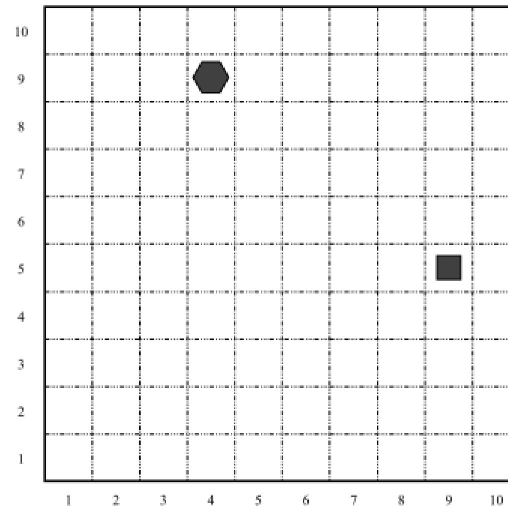
too cold — add hot
too hot — add cold
change — wait 5sec
just right — success

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

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).

HRL: a toy example

100
90
80
70
60
50

S S G
G

S: start G: goal

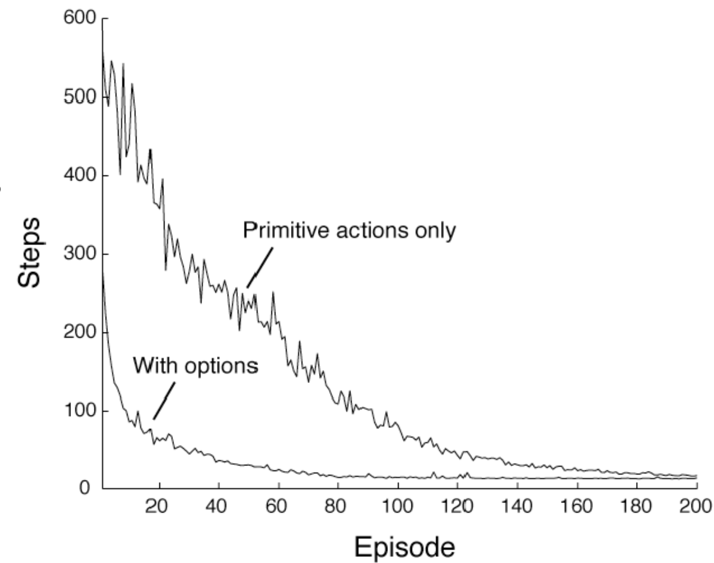
Options: going to doors

Actions: + 2 door options

Advantages of HRL

1. Faster learning
   (mitigates scaling problem)

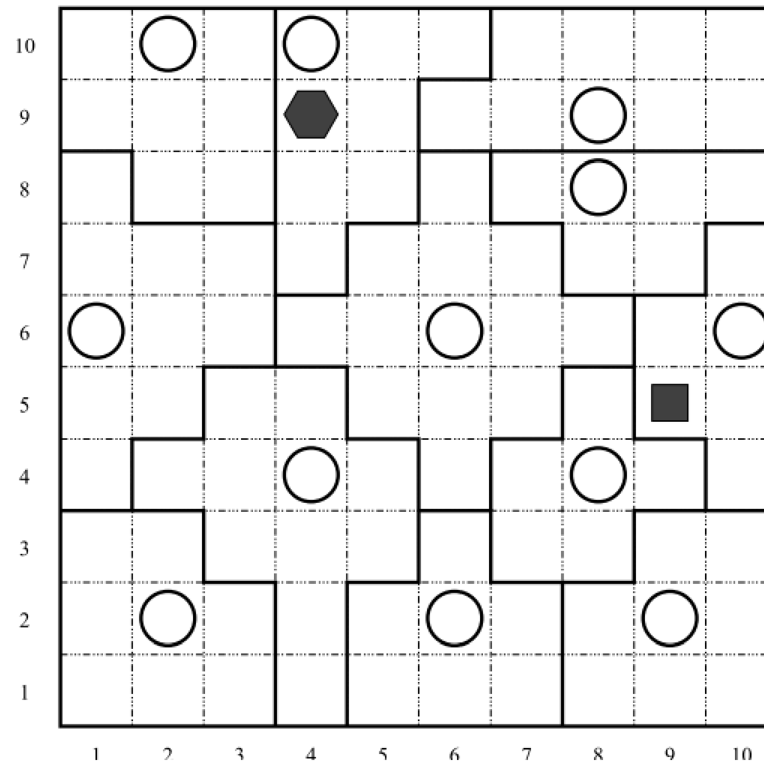
2. *Transfer of knowledge* from previous tasks
   (generalization, shaping)



RL: no longer 'tabula rasa'

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).

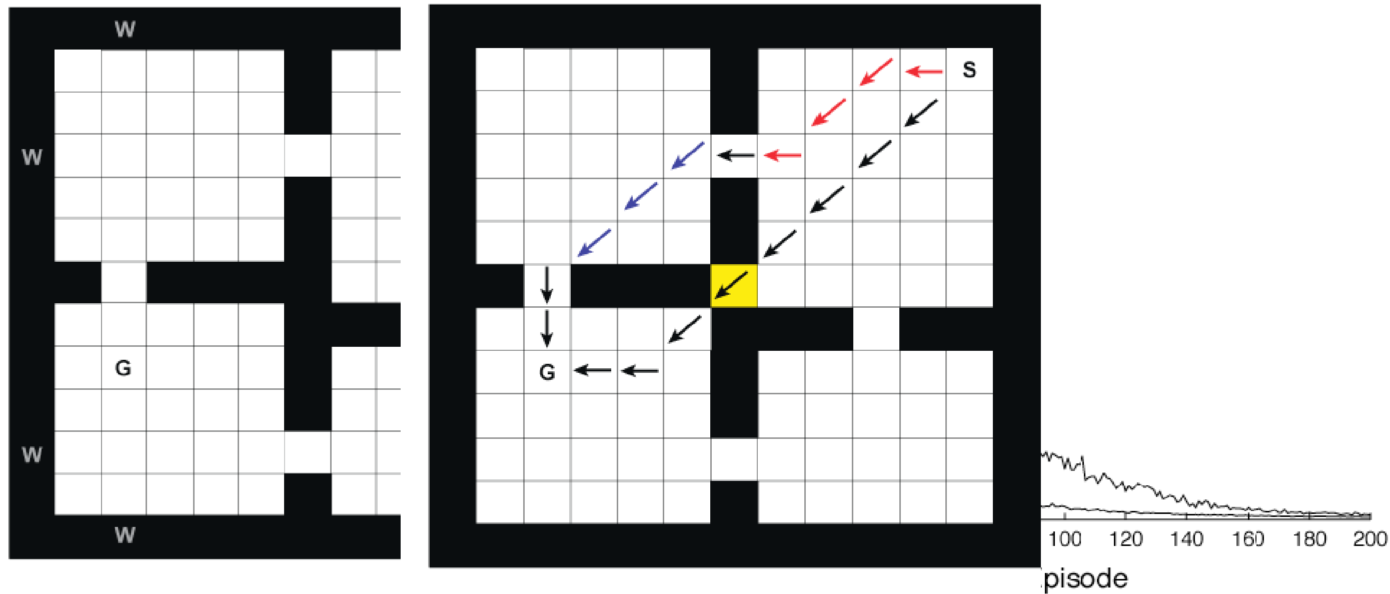
Each of these subtasks can be shared by many combinations of initial states and goal states.

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

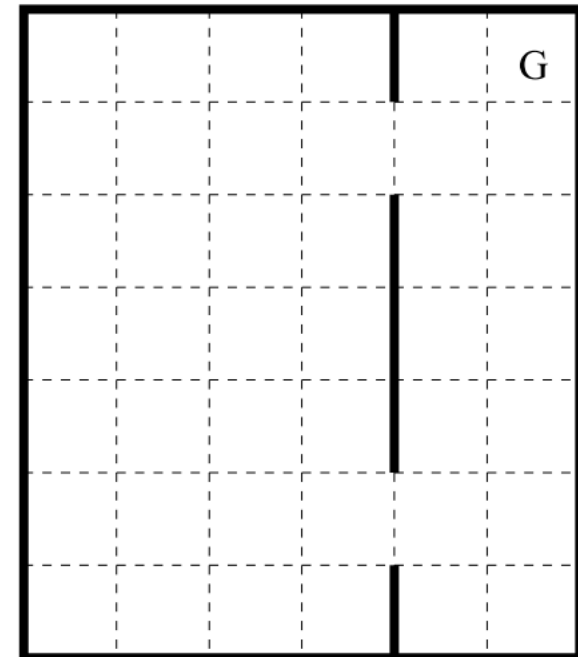
1. Need 'right' options - how to learn them?
2. Suboptimal behavior ("negative transfer"; habits)
3. More complex learning/control structure



no free lunches…

Example problem

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



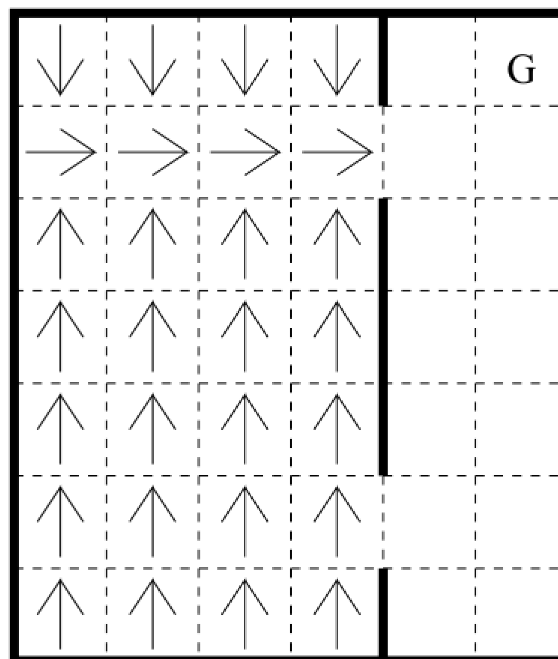
Options (Sutton; Precup; Singh)

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

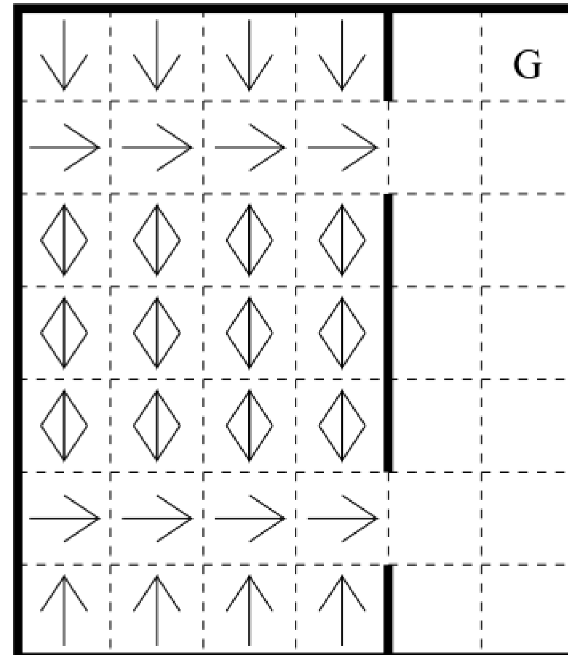
- **Mapping from states to sets of possible actions**.

  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.



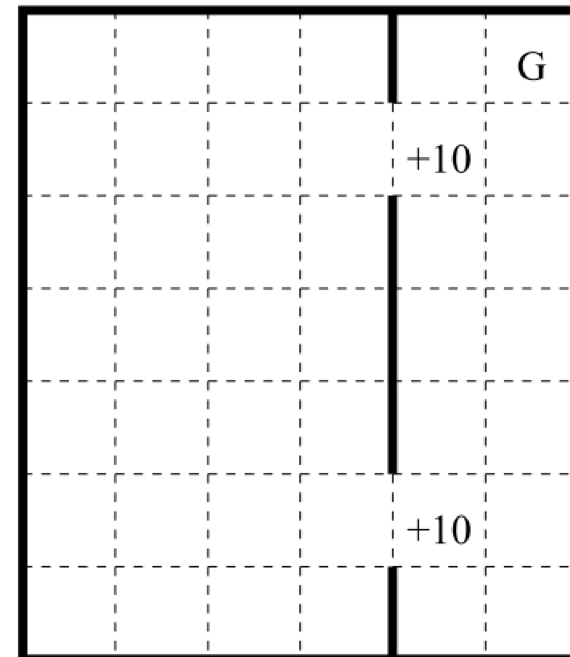
Subtasks

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:

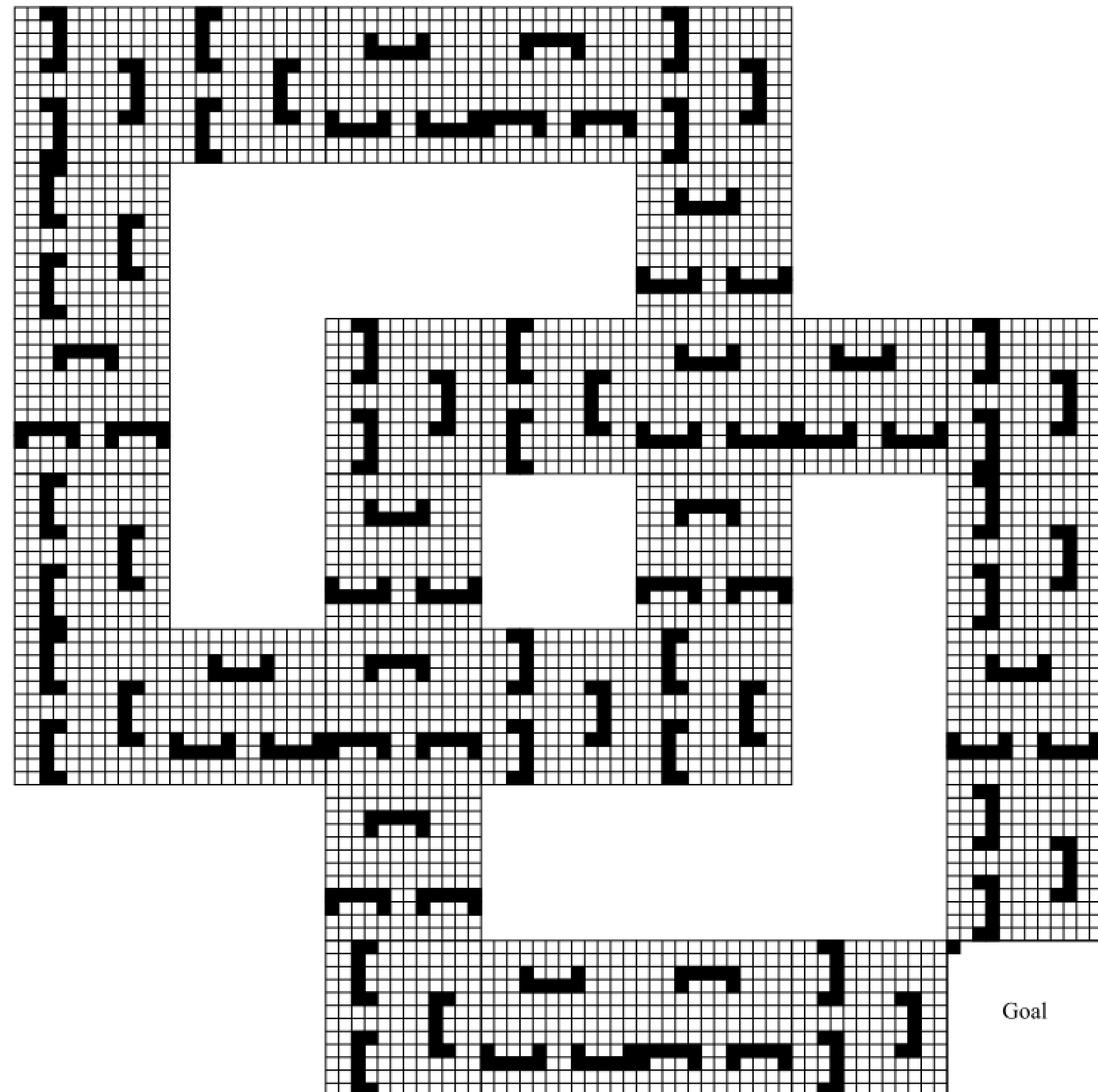
- **States:** pairs of [state, call-stack] pairs
- **Actions:** as dictated by partial policy
- **Reward function:** same as original reward function.

Apply SMDP Q learning

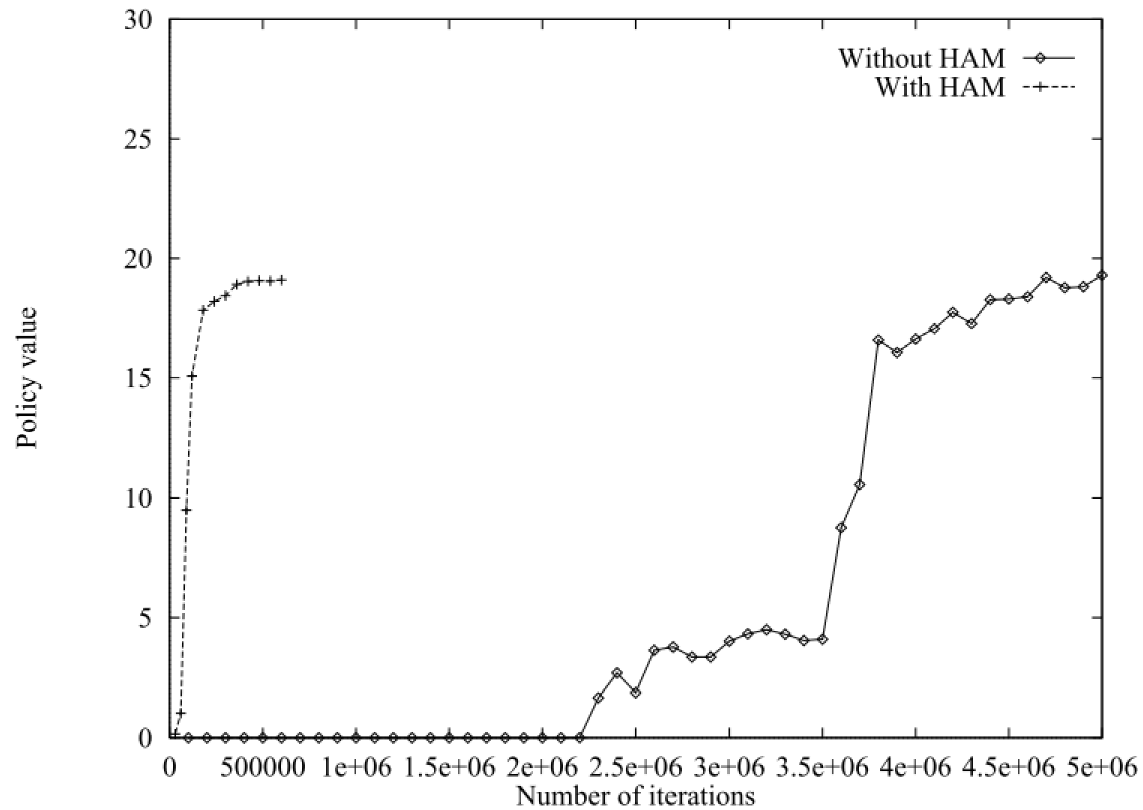
Task 2: Example: Parr's Maze Problem

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).

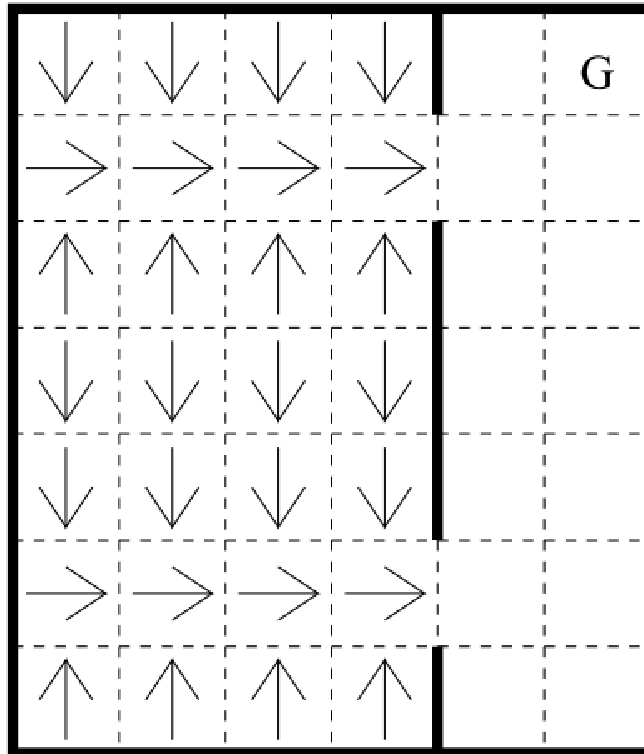
However, it converges to only a *locally optimal* policy.

Hierarchical Optimality versus Recursive Optimality

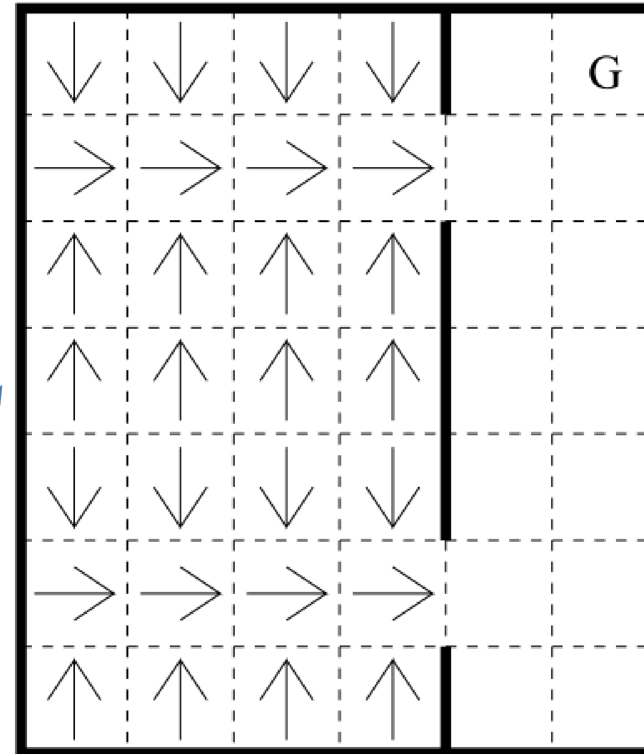
- **Hierarchical Optimality: The overall learned policy is the best policy consistent with the hierarchy**
- **Recursive Optimality: The policy for a task is optimal** *given* **the policies learned by its children**

- **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



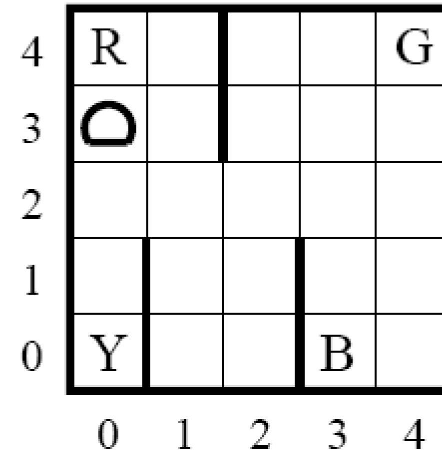
Locally optimal

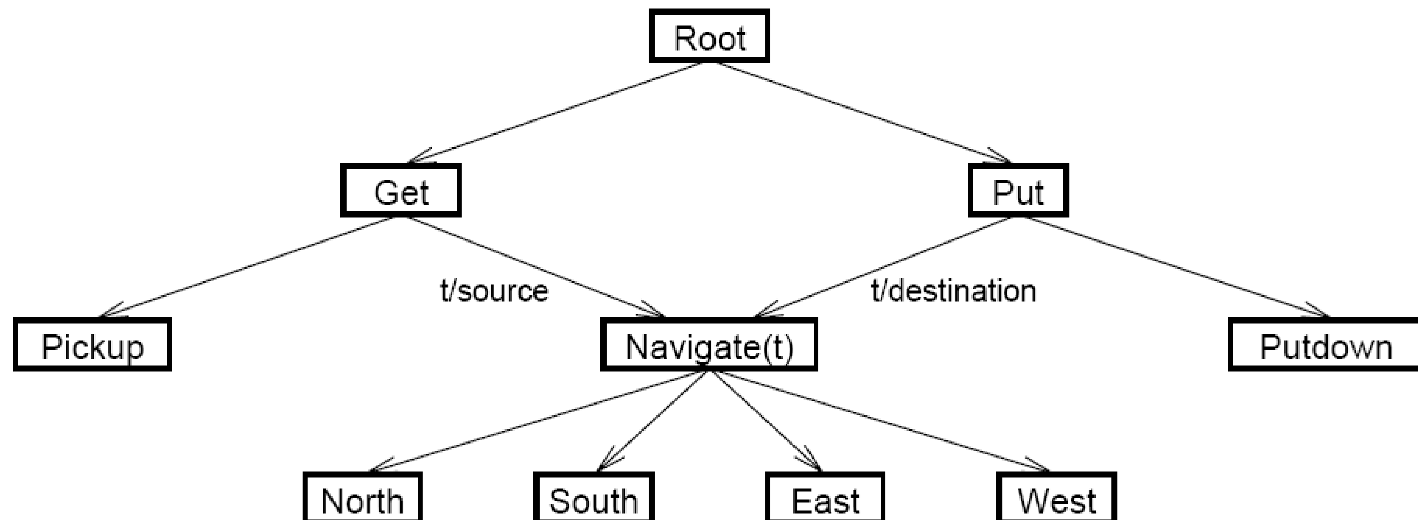


Optimal for the entire task

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

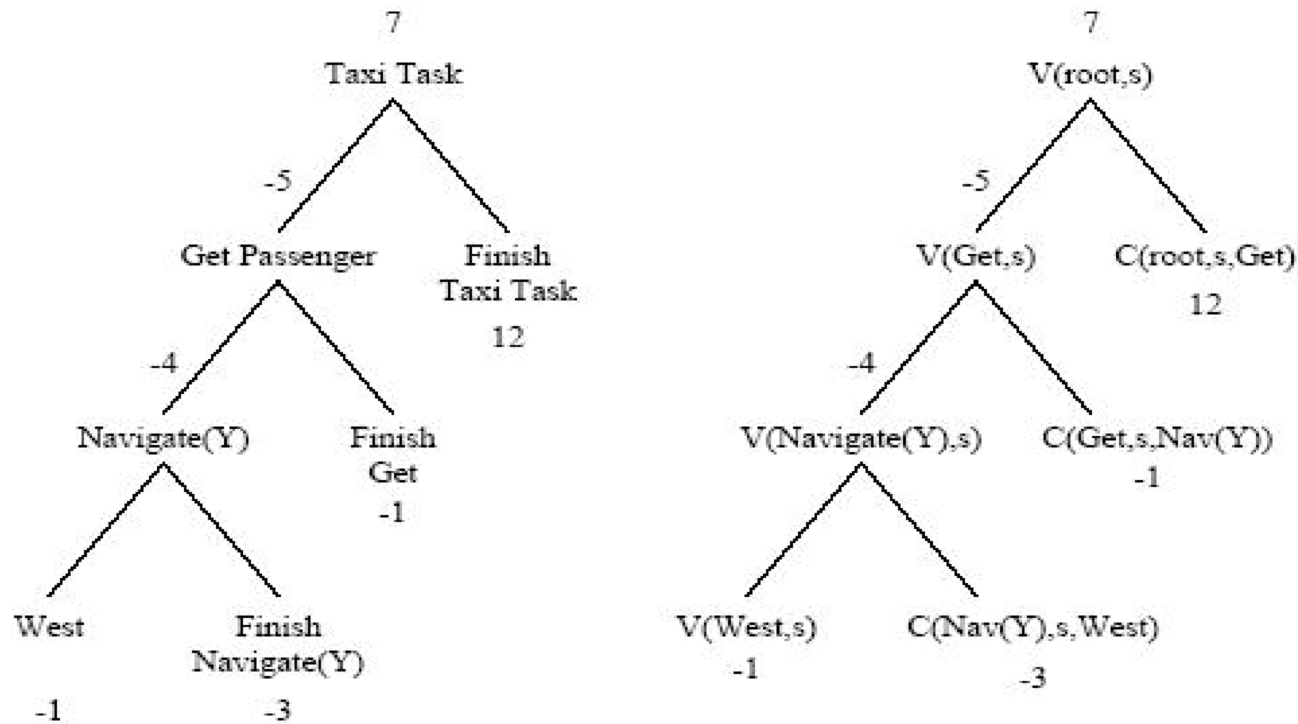


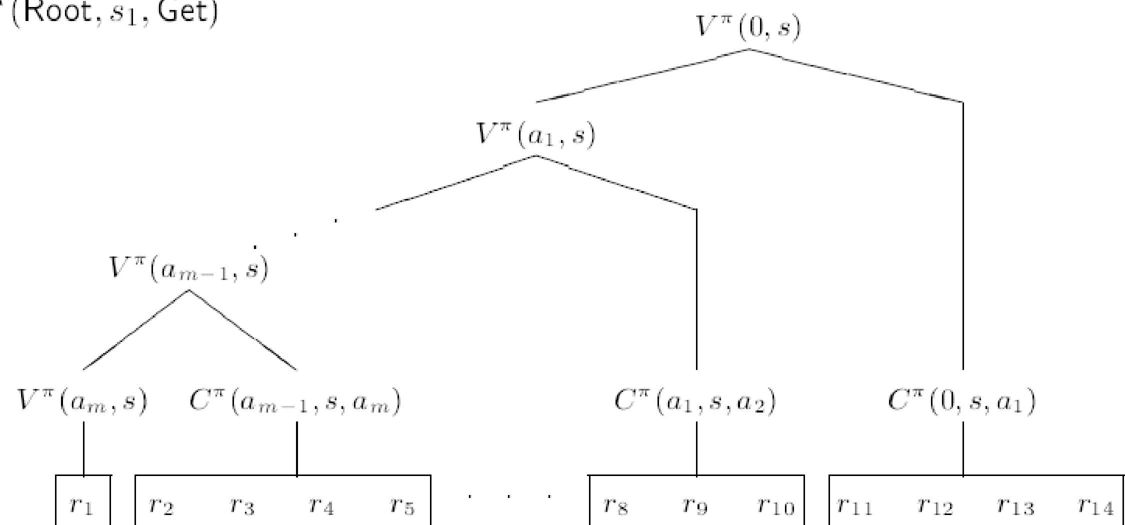
Fig. 5. An example of the MAXQ value function decomposition for the state in which the taxi is at location (2,2), the passenger is at (0,0), and wishes to get to (3,0). The left tree gives English descriptions, and the right tree uses formal notation.

Value Decomposition

- The value function can be decomposed as follows

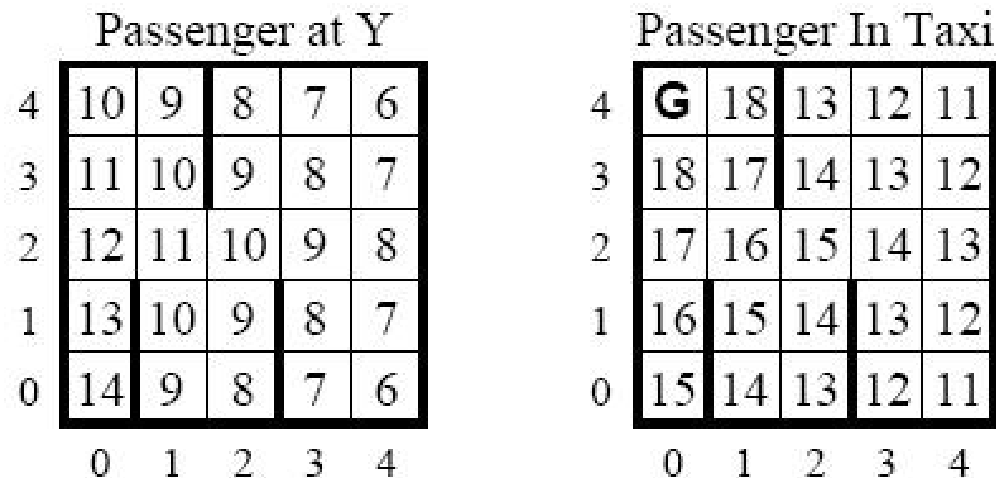
$$V^{\pi}(0,s) = V^{\pi}(a_m,s) + C^{\pi}(a_{m-1},s,a_m) + \ldots + C^{\pi}(a_1,s,a_2) + C^{\pi}(0,s,a_1)$$

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



MAXQ Alg. (cont'd)

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



Passenger at Y

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
	0	1	2	3	4

Passenger In Taxi

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
	0	1	2	3	4

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)

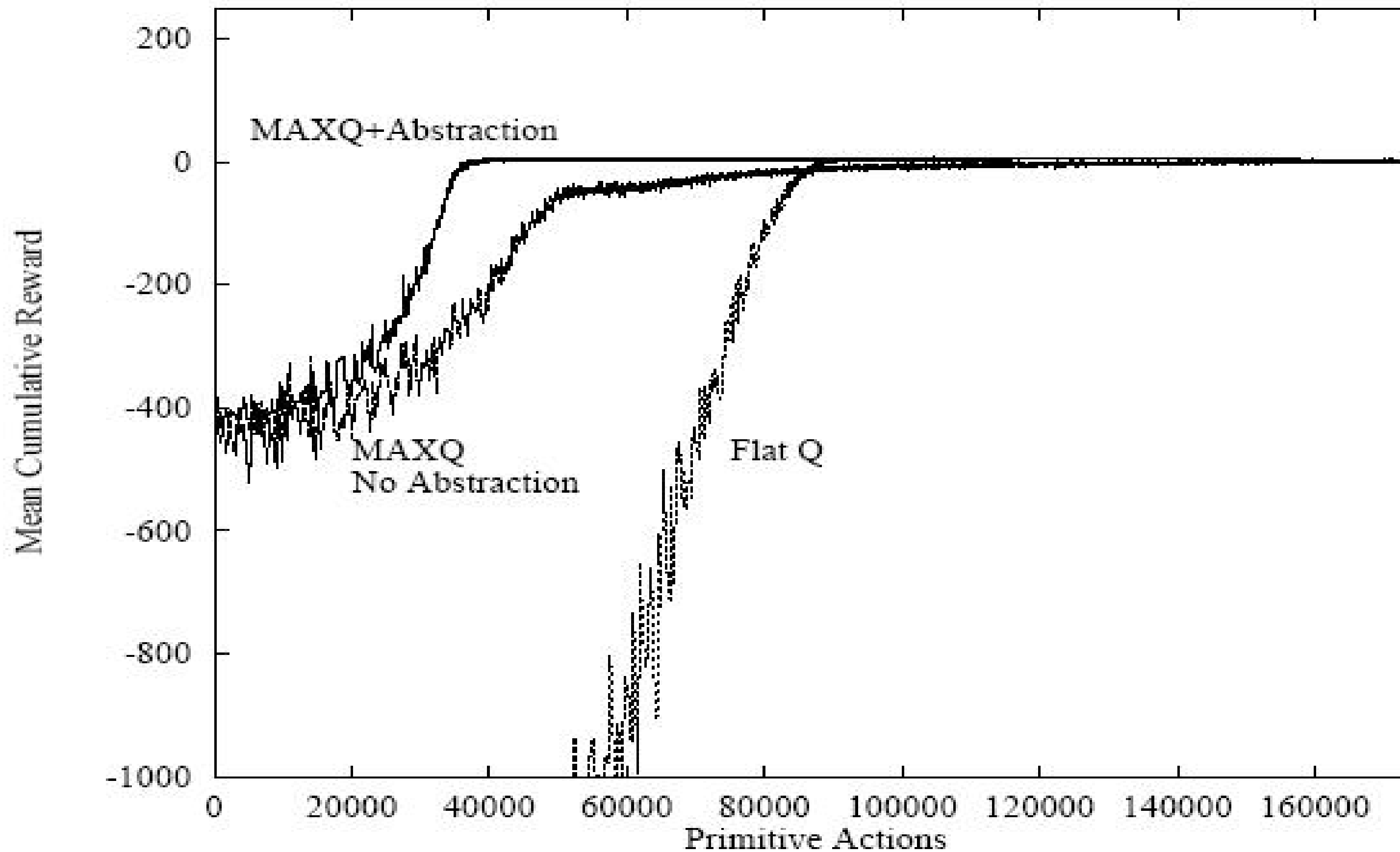
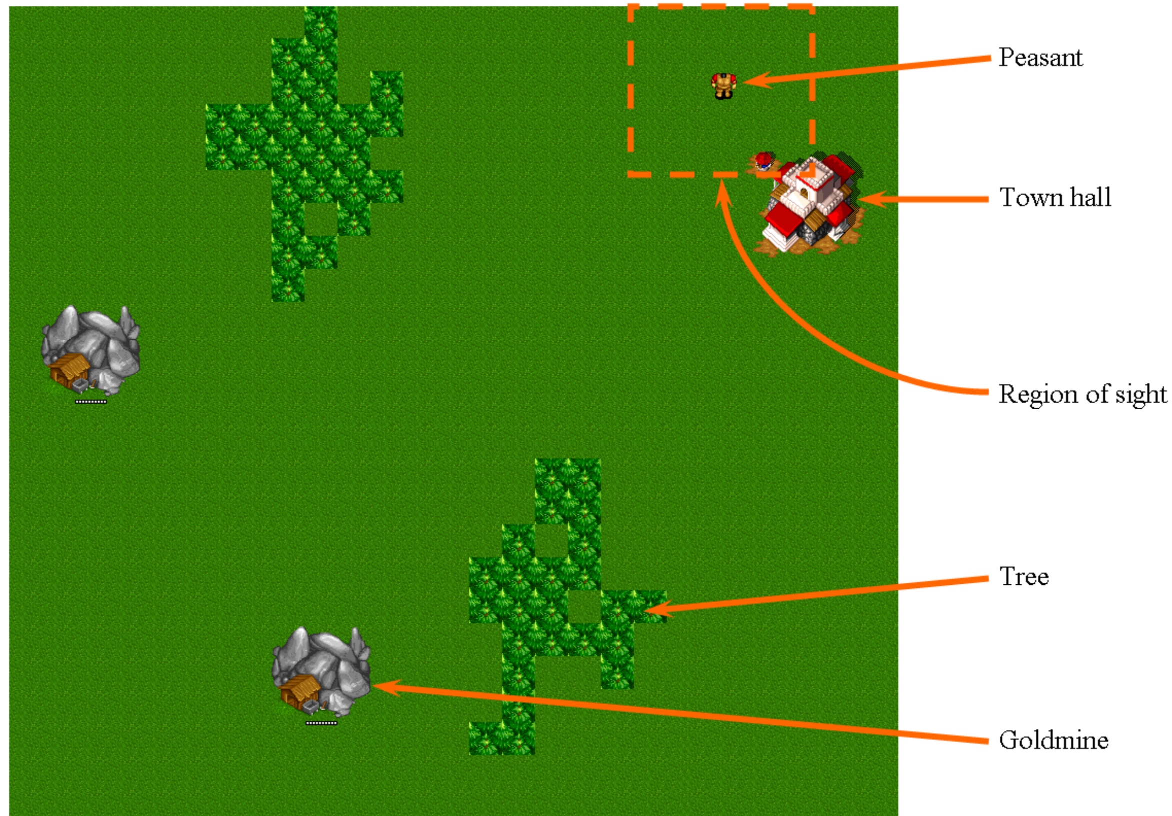


Fig. 7. Comparison of Flat Q learning, MAXQ Q learning with no state abstraction, and MAXQ Q learning with state abstraction on a noisy version of the taxi task.

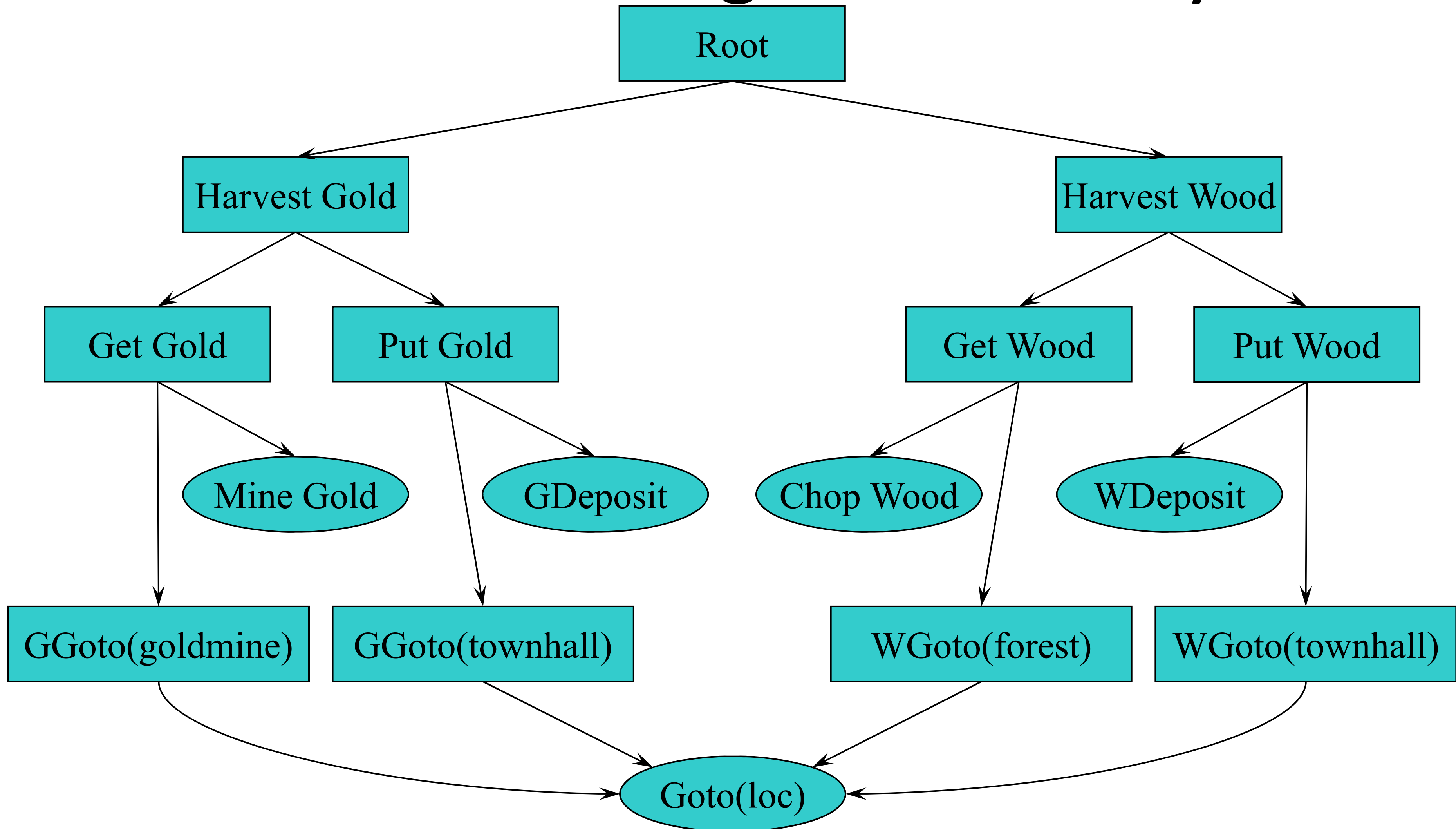
Wargus Resource-Gathering Domain



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

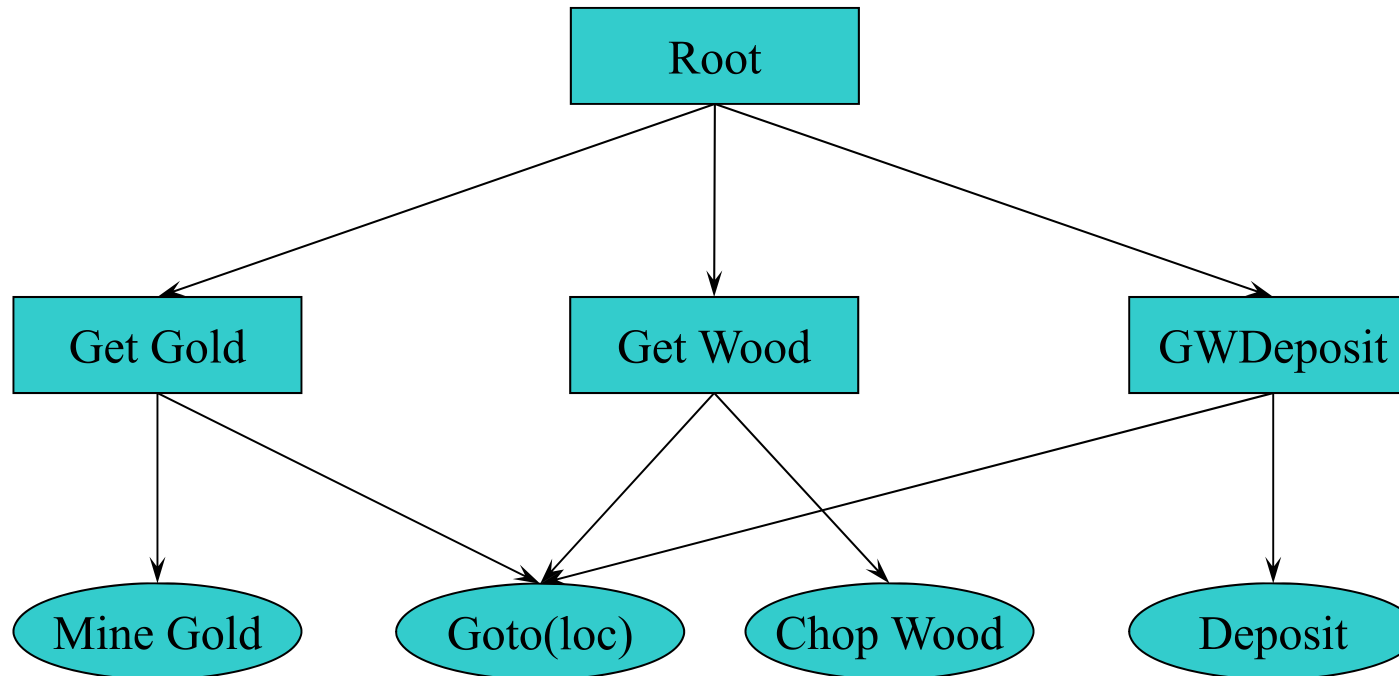
Claims

- 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

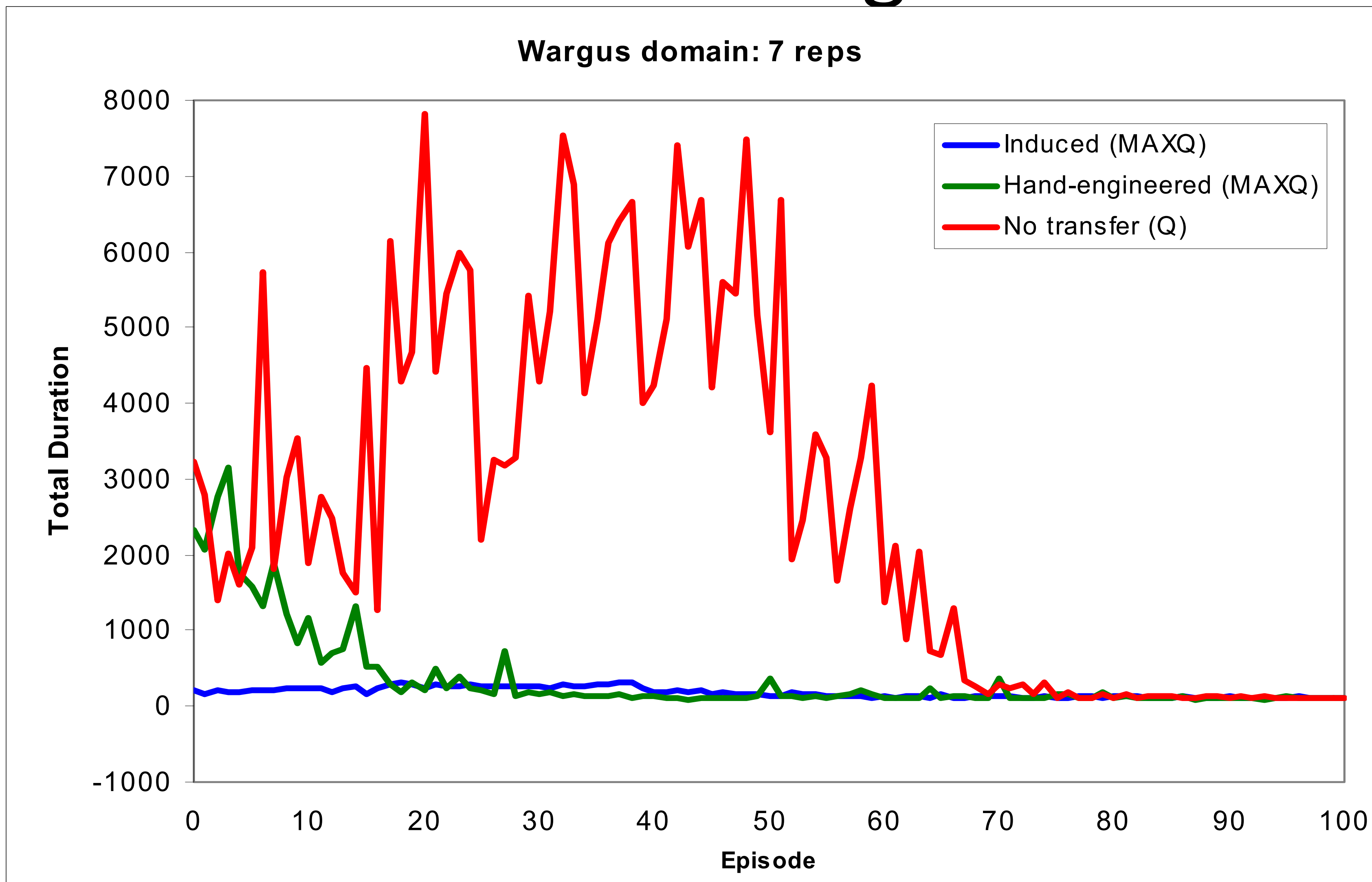
Hand-Built Wargus Hierarchy



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



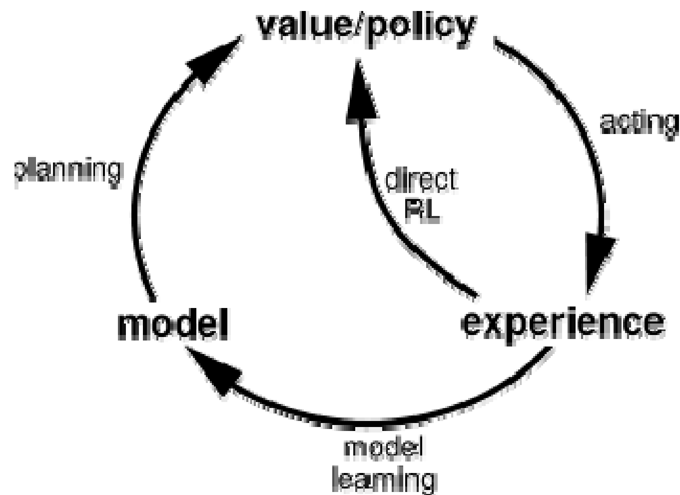
Limitations

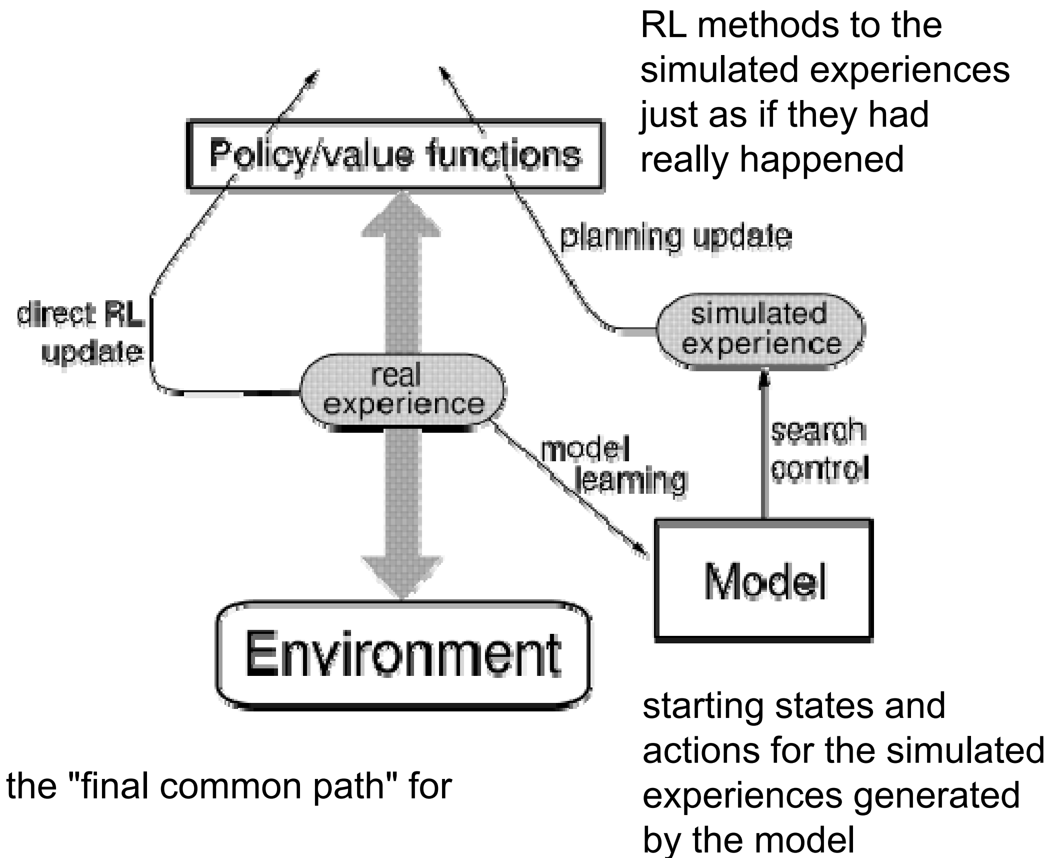
- Recursively optimal not necessarily optimal
- Model-free Q-learning

  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





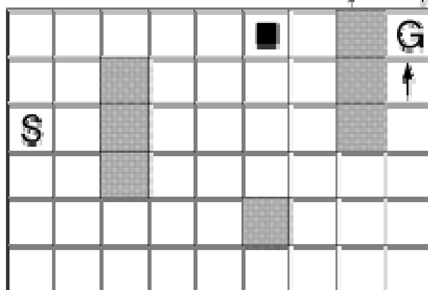
The reinforcement learning method is thus the "final common path" for both learning and planning

Planning, Acting, Learning

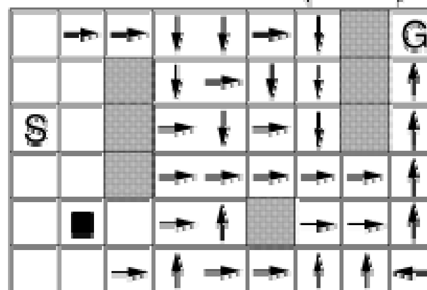
- Dyna-Q alg.

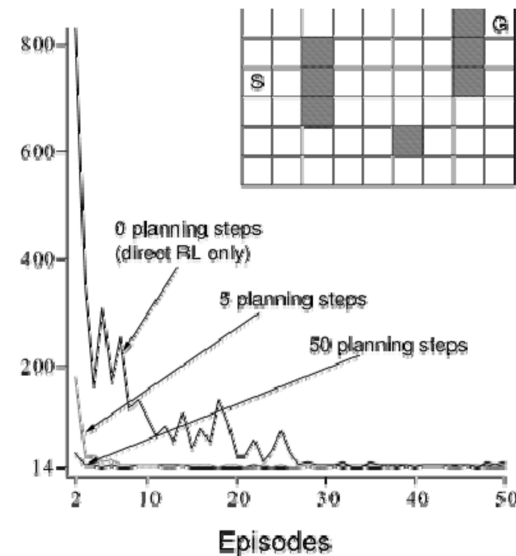
Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
  (a) $s \leftarrow$ current (nonterminal) state
  (b) $a \leftarrow \varepsilon\text{-greedy}(s, Q)$
  (c) Execute action a; observe resultant state, s', and reward, r
  (d) $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
  (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
    $s', r \leftarrow Model(s,a)$
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$

WITHOUT PLANNING (N=0)



WITH PLANNING (N=50)





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