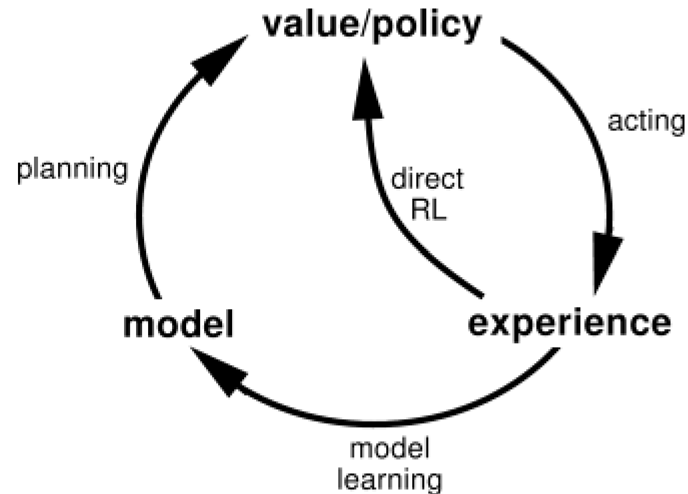
Applications of RL

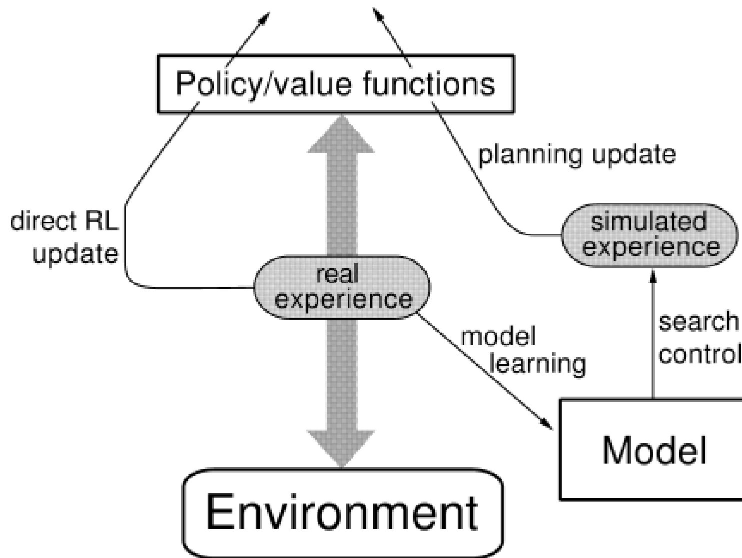
- Checker's [Samuel 59]
- TD-Gammon [Tesauro 92]
- World's best downpeak elevator dispatcher [Crites at al ~95]
- Inventory management [Bertsekas et al ~95]
  - 10-15% better than industry standard
- Dynamic channel assignment [Singh & Bertsekas, Nie&Haykin ~95]
  - Outperforms best heuristics in the 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]

Planning and Learning

- Dyna-Q algorithm



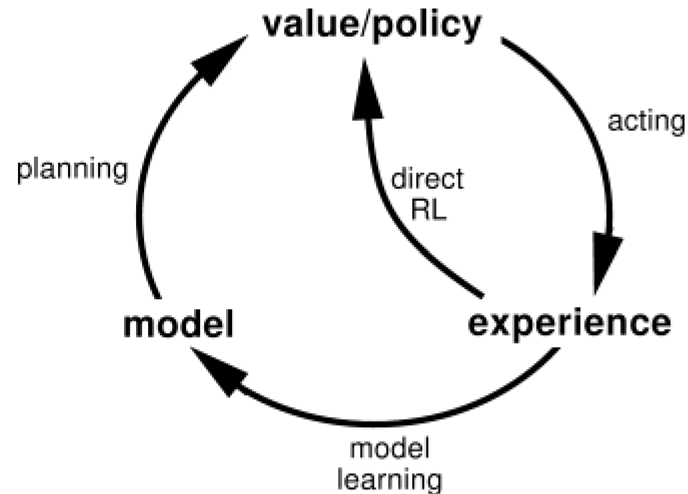
Experience can improve value and policy functions either directly or indirectly via the model. It is the latter, which is sometimes called *indirect reinforcement learning*, that is involved in planning.



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

Planning and Learning

- Dyna-Q algorithm



Experience can improve value and policy functions either directly or indirectly via the model. It is the latter, which is sometimes called *indirect reinforcement learning*, that is involved in planning.

The agent is always reactive and always deliberative, responding instantly to the latest sensory information and yet always planning in the background.

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

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

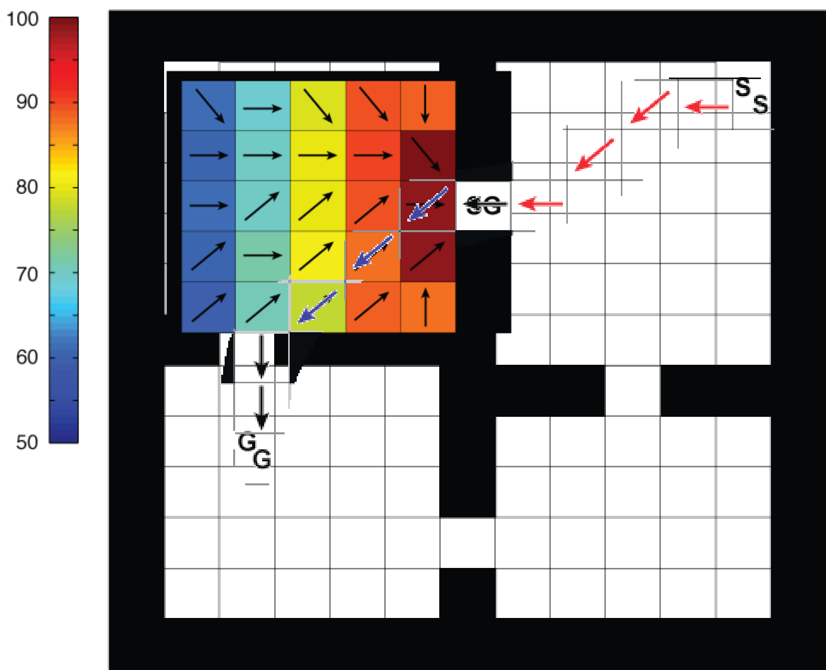
too cold — add hot
too hot — add cold
change — wait 5sec
just right — success

simplified control, disambiguation, encapsulation

Hierarchical Reinforcement Learning

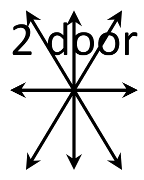
- Exploits domain structure to facilitate learning
  - Policy constraints
  - State abstraction
- Paradigms: Options, HAMs, MaxQ
- MaxQ task hierarchy
  - Directed acyclic graph of subtasks
  - Leaves are the primitive MDP actions
- Traditionally, task structure is provided as prior knowledge to the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

HRL: a toy example



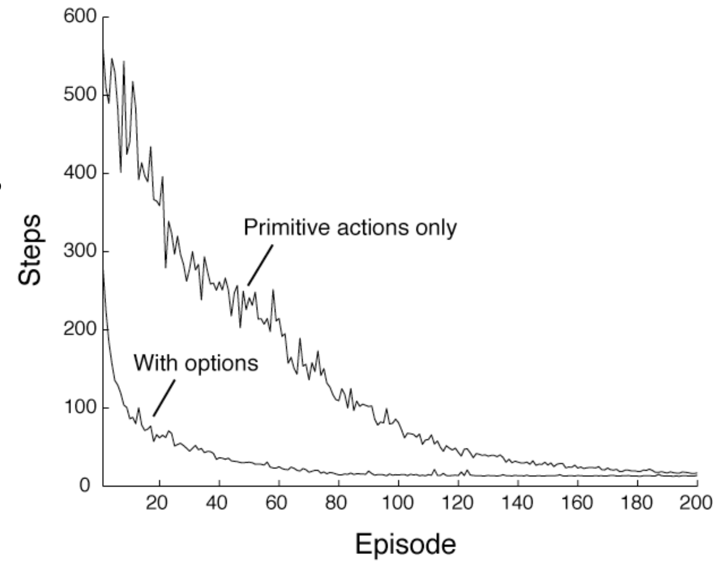
S: start G: goal

Options: going to doors

Actions:  + 2 door options

Advantages of HRL

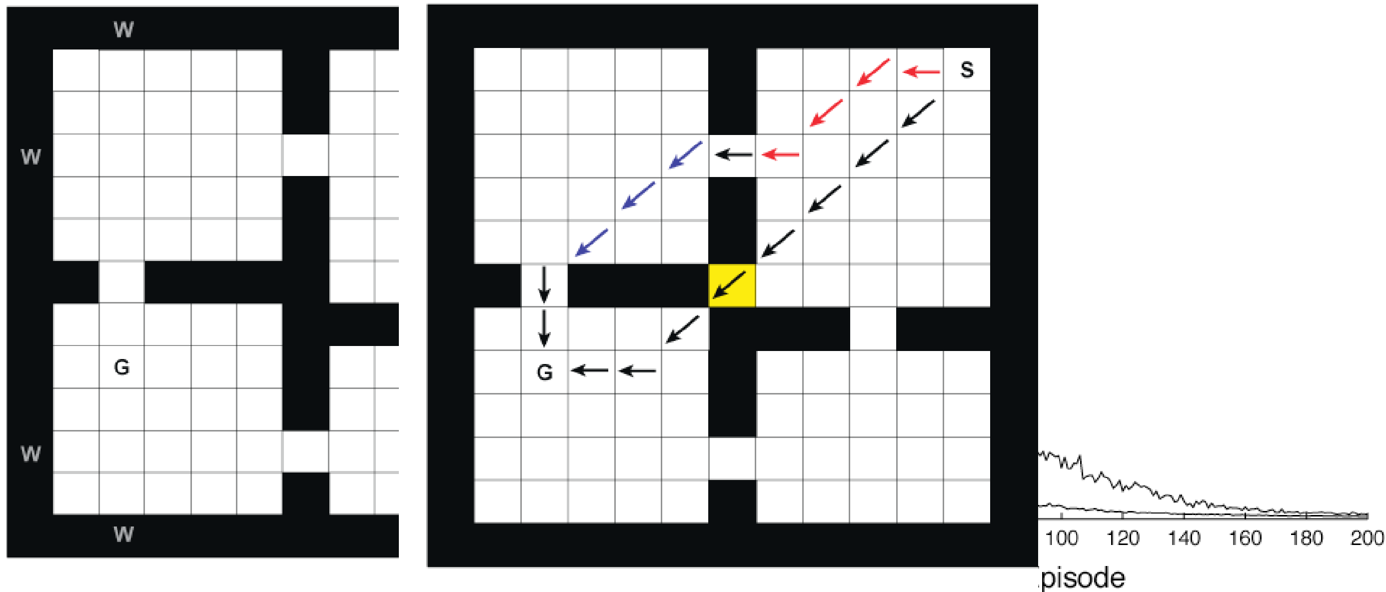
1. Faster learning
   (mitigates scaling problem)
2. *Transfer of knowledge* from previous tasks
   (generalization, shaping)



RL: no longer 'tabula rasa'

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…

Semi-Markov Decision Process

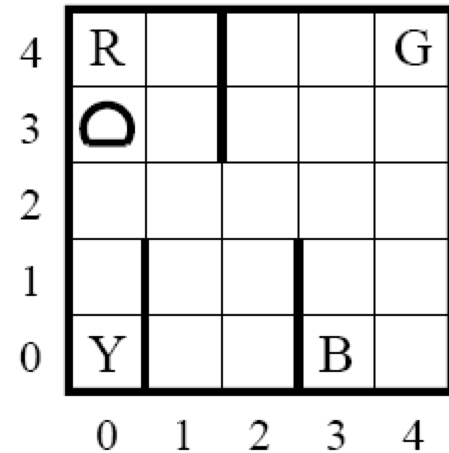
- Generalizes MDPs
- Action **a** takes **N** steps to complete in **s**
- P(**s'**,**n** | **a**, **s**), R(**s'**, **N** | **a**, **s**)
- Bellman equation:

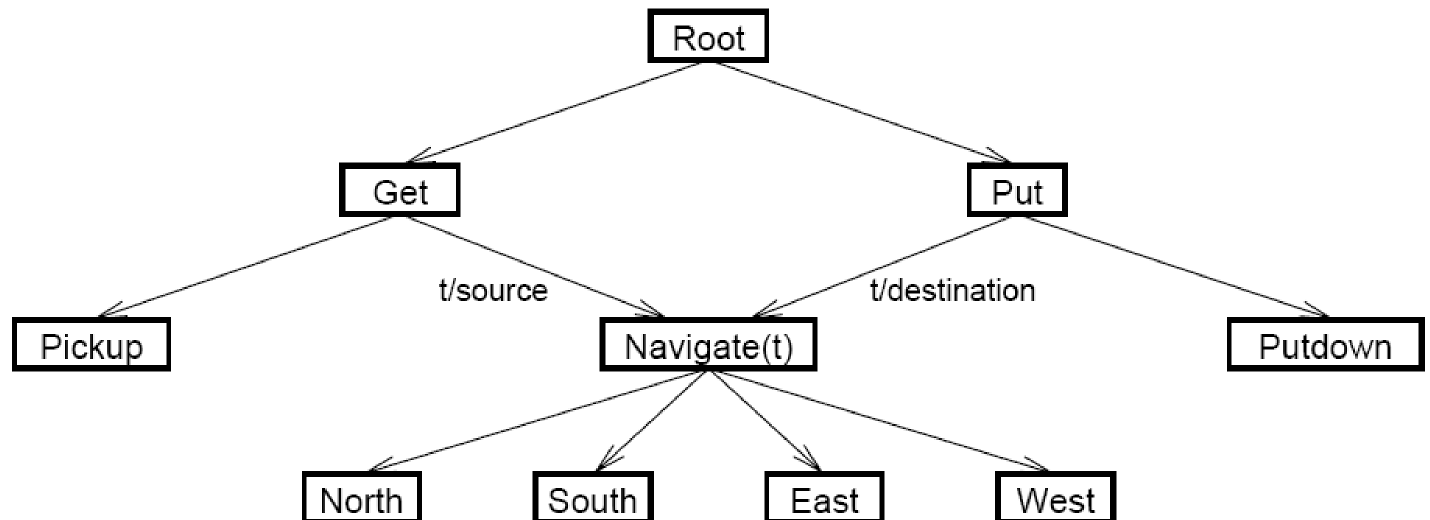
$$V^{\pi}(s) = \sum_{s',N} P(s', N | s, \pi(s)) \left[R(s', N | s, \pi(s)) + \gamma^N V^{\pi}(s') \right].$$

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

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 = π_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 − α)Q(p, s, a) + α [r + max_a' Q(p, s', a')]
    end // while
    return TotalReward
end
```

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 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

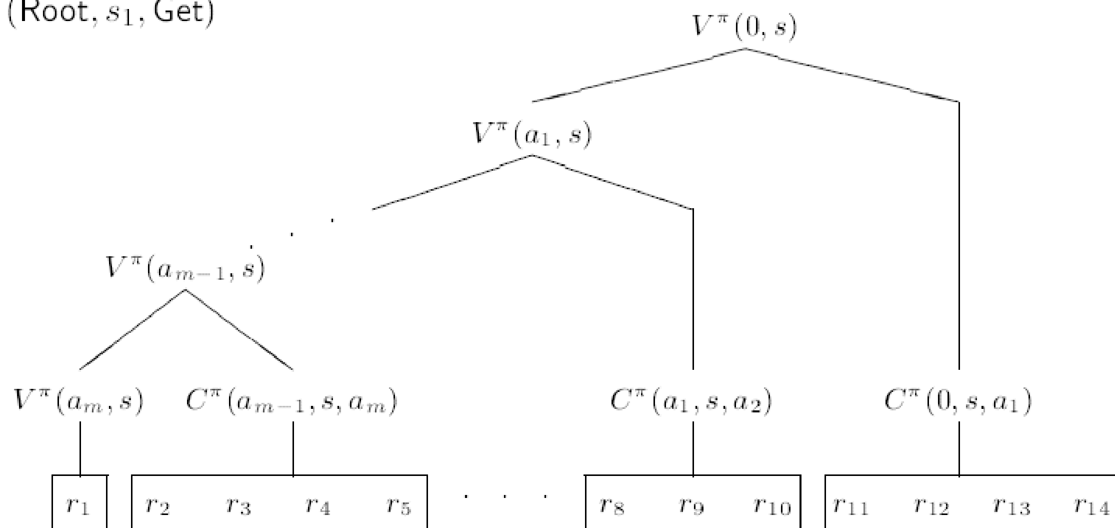
$$V^\pi(i,s) = \begin{cases} Q^\pi(i,s,\pi_i(s)) & \text{if } i \text{ is composite} \\ \sum_{s'} P(s'|s,i)R(s'|s,i) & \text{if } i \text{ is primitive} \end{cases} \tag{11}$$

Value Decomposition

- The value function can be decomposed as follows

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

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



MAXQ Alg. (cont'd)

- An example

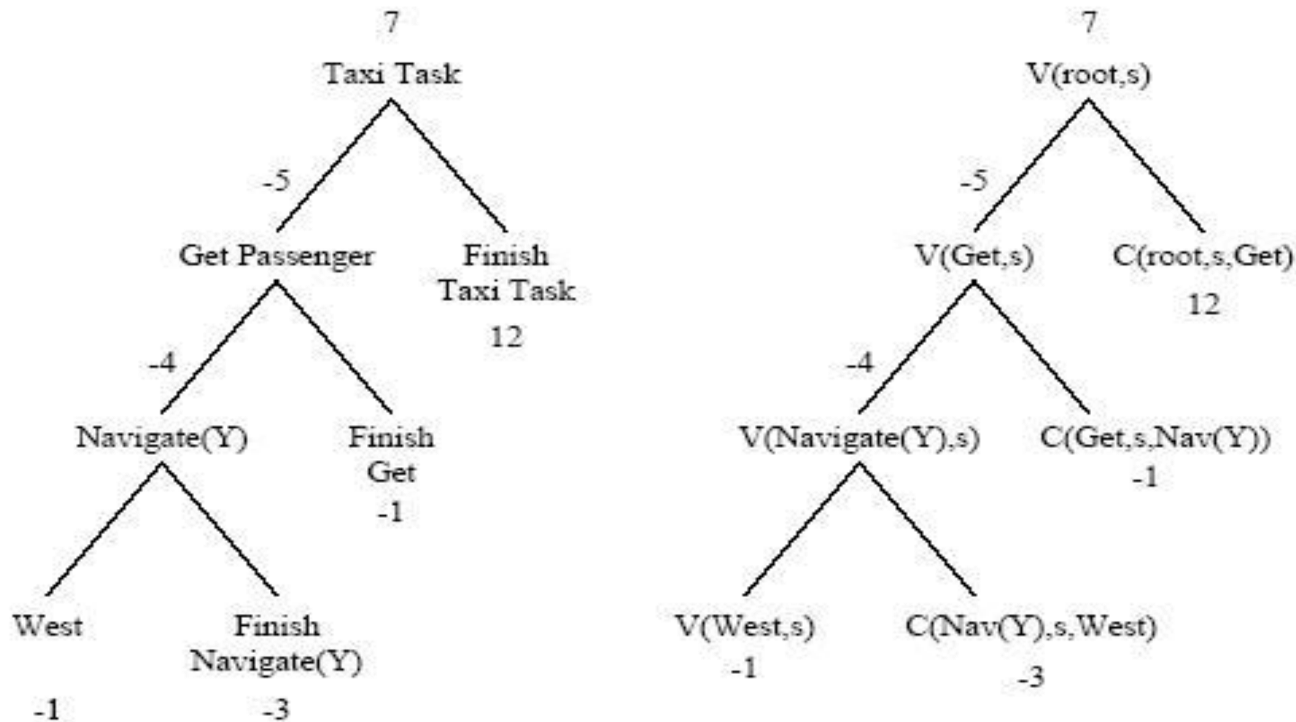


Fig. 5. An example of the MAXQ value function decomposition for the state in which the taxi is at location (2,2), the passenger is at (0,0), and wishes to get to (3,0). The left tree gives English descriptions, and the right tree uses formal notation.

MAXQ Alg. (cont'd)

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

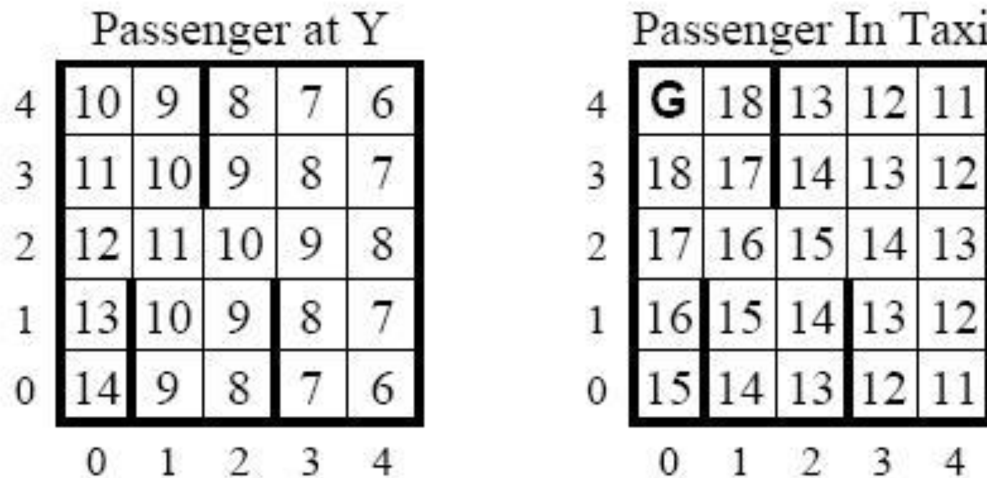


Fig. 4. Value function for the case where the passenger is at (0,0) (location Y) and wishes to get to (0,4) (location R).

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)

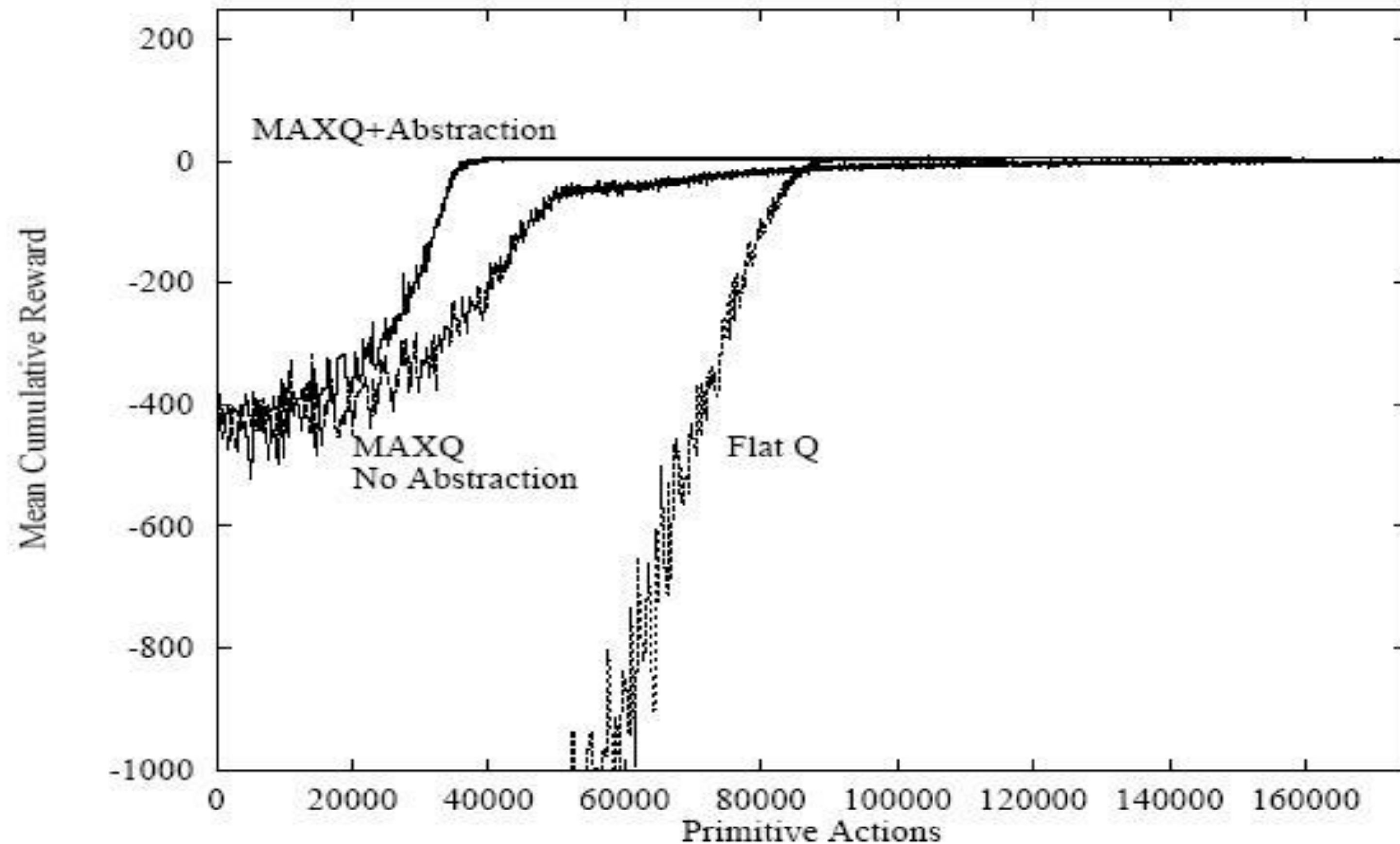
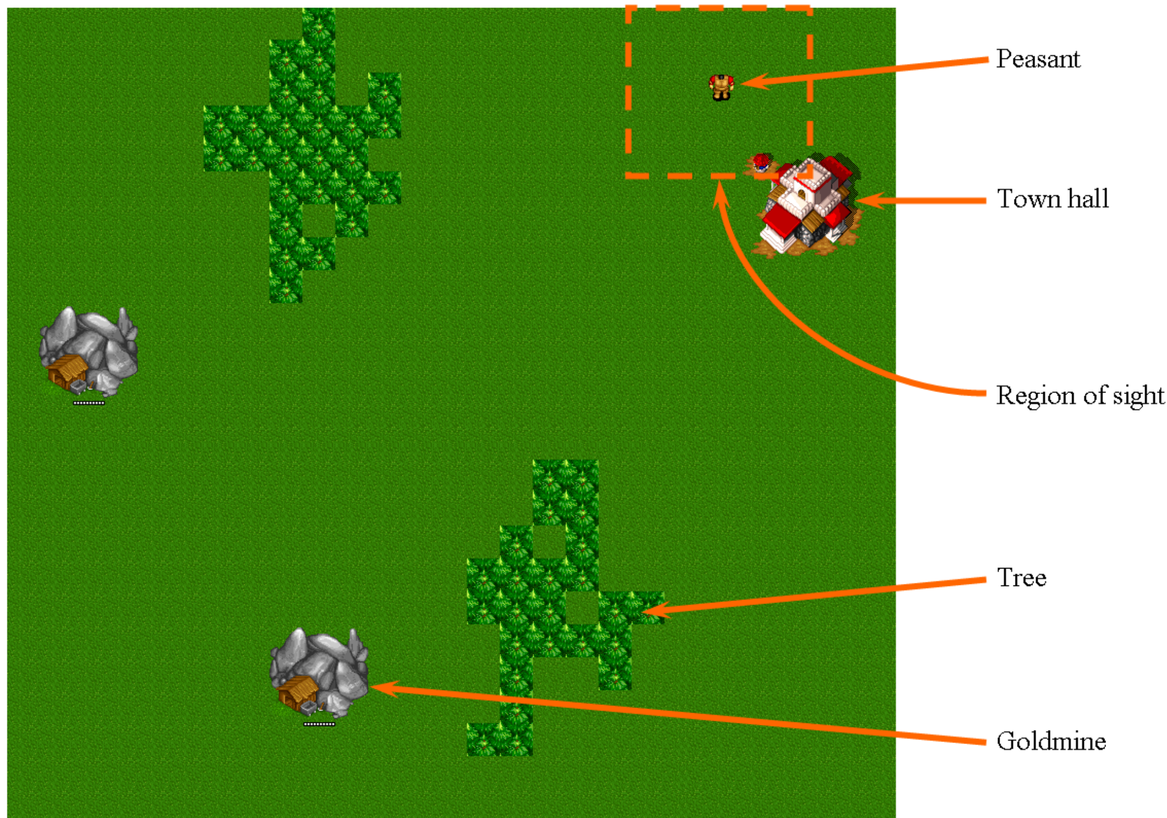


Fig. 7. Comparison of Flat Q learning, MAXQ Q learning with no state abstraction, and MAXQ Q learning with state abstraction on a noisy version of the taxi task.

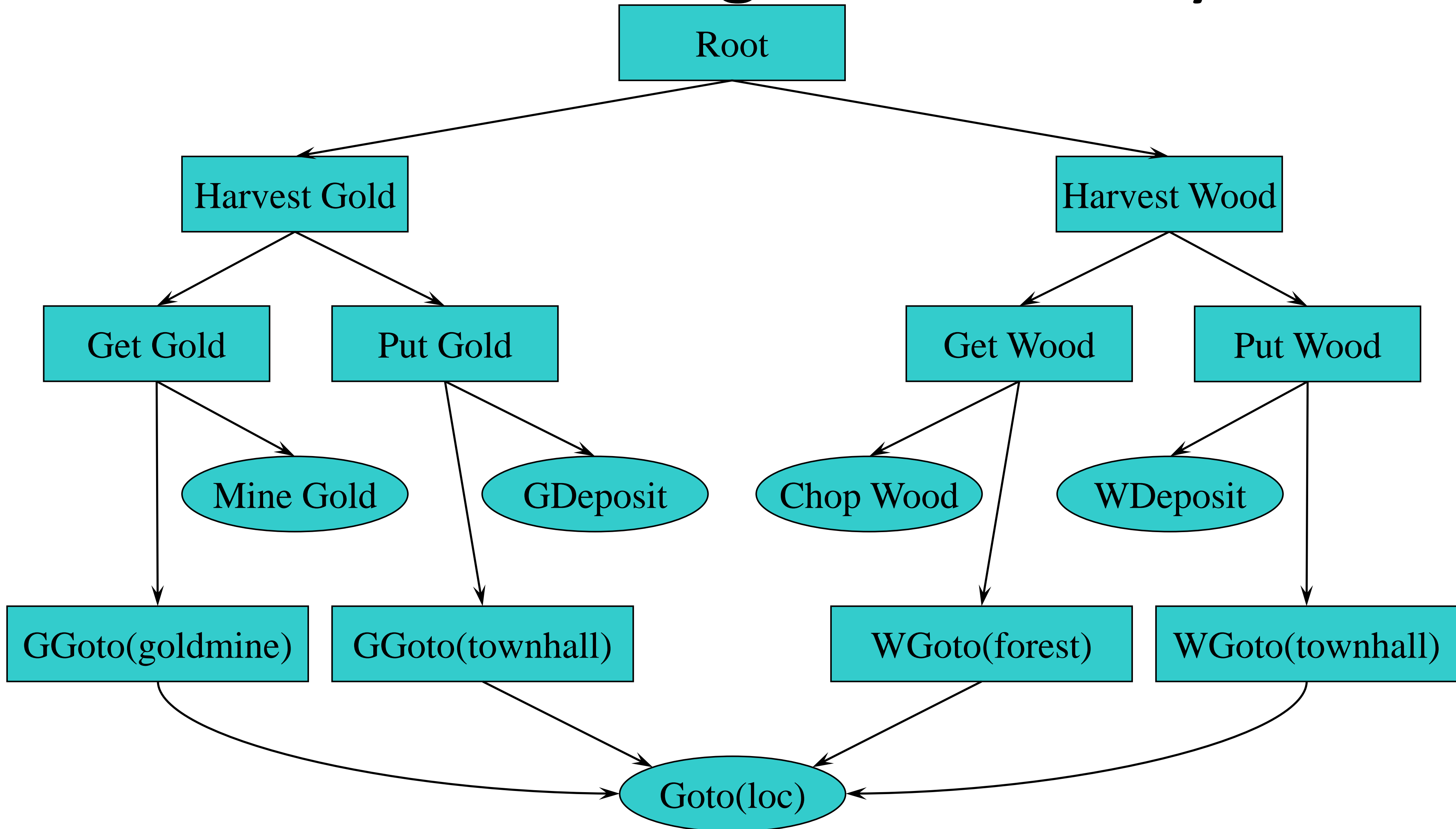
Wargus Resource-Gathering Domain



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

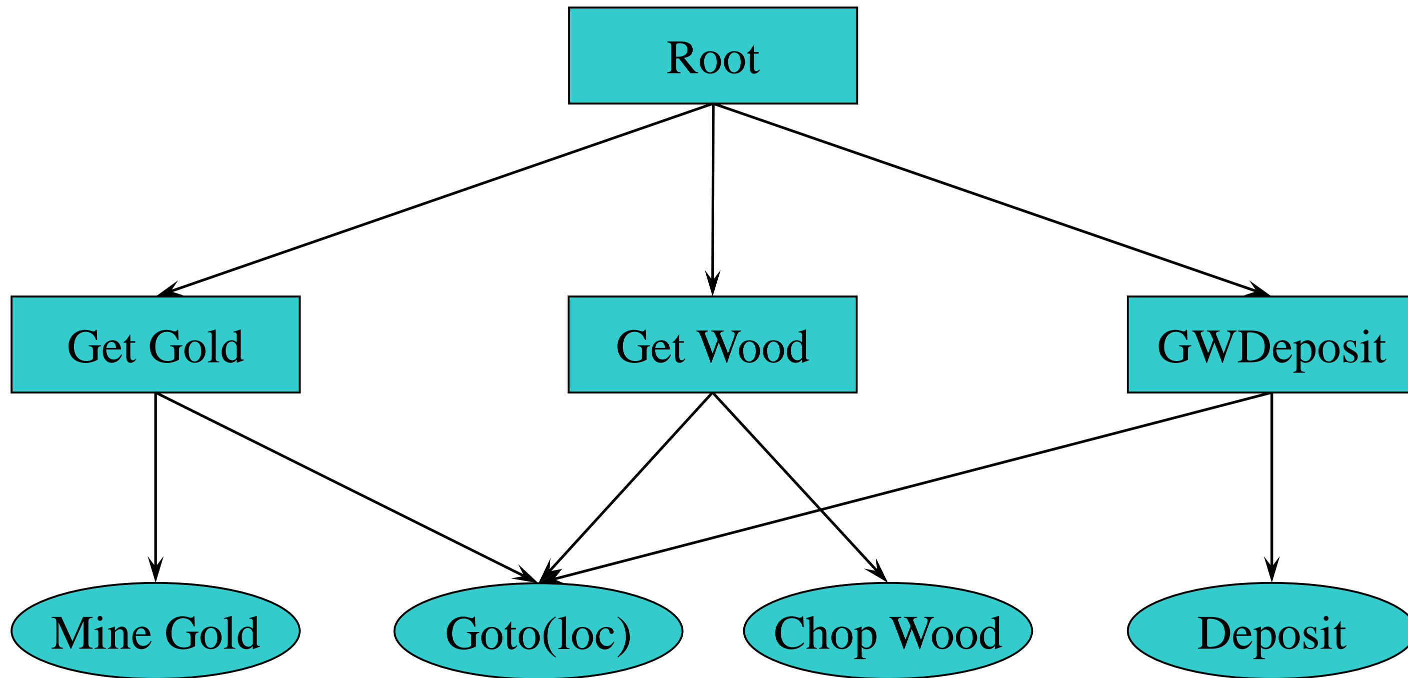
Claims

- 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

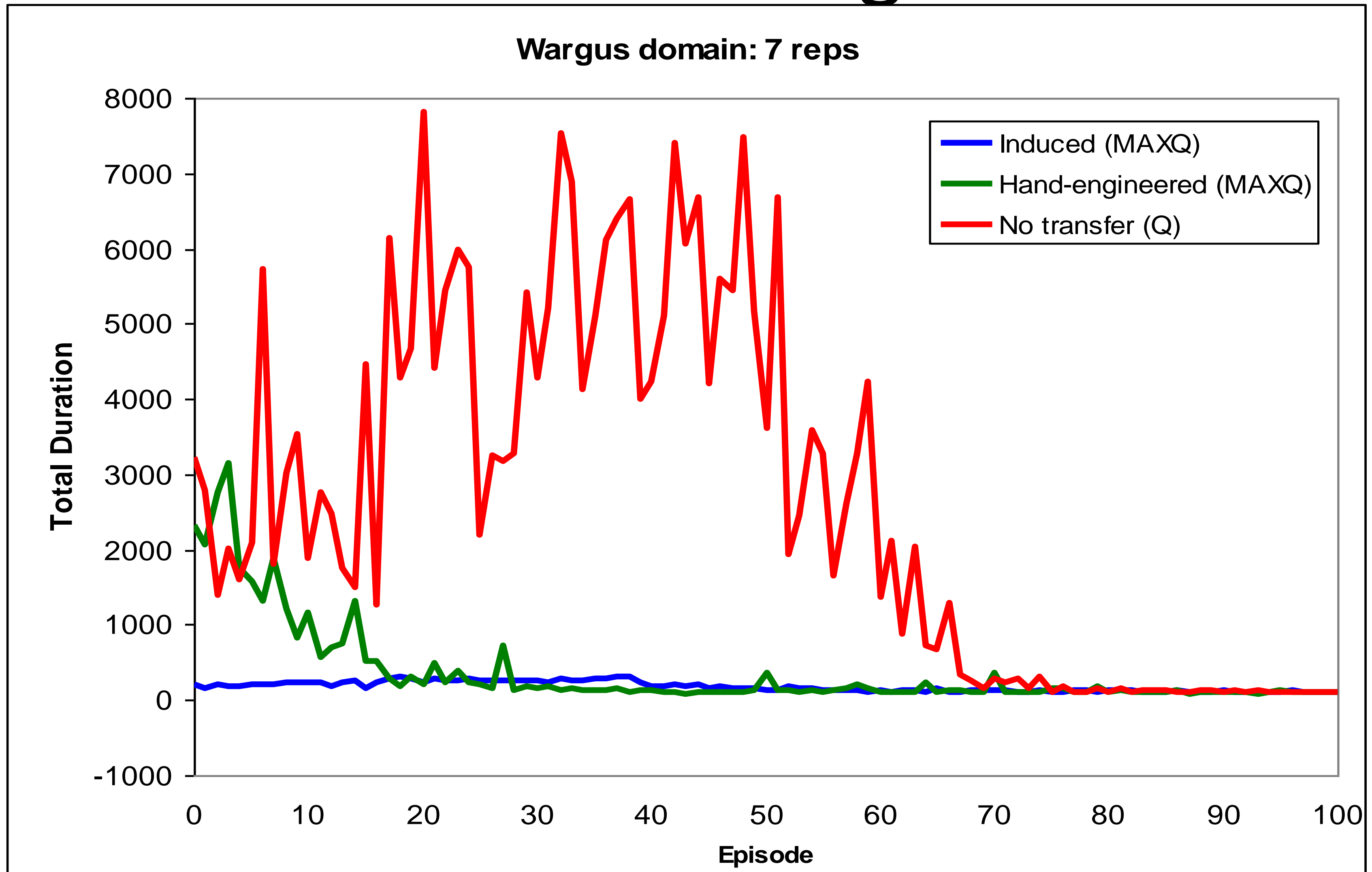
Hand-Built Wargus Hierarchy



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



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