Reinforcement learning (CS6700)
Written assignment #4

Aravind S
EE14B013

14th Apr. 2017

1 Problem 1

1.1 Part (a)

We are using linear function approximator for TD(0). The features extracted for a particular state s is denoted by $\phi(s)$ and say we are learning the value function $V(s)$ for every state s. Since we are using a linear function approximator,

$$V(s;\theta) = \theta^T \phi(s)$$

where $\phi(s)$ is a $t\,X\,1$ vector and θ represents the parameter and is also a $t\,X\,1$ vector.

Using eligibility traces we can modify the update rules by using past-gradient information also.

Formulation

Let e_t denote the eligibility vector for all the parameters at the time instant t. This has the same dimension as θ.

$$e_t = \gamma\lambda e_{t-1} + \nabla_{\theta_t} V(s_t;\theta_t)$$

$$e_t = \gamma\lambda e_{t-1} + \phi(s_t)$$

Using e_t the update rule for θ becomes,

$$\theta_{t+1} \leftarrow \theta_t + \alpha\delta_t e_t$$

where $\delta_t = R_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ which is the TD-error at the time step t.

Algorithm

```
Input: policy π - value function is estimated for this policy
Initialise value-function weights θ arbitrarily
Repeat ( for each episode ):

    Initialise state
    Obtain initialise feature vector φ
    e ← 0 # t X 1 vector
```

$V_{old} \leftarrow 0$
Repeat (for each step of the episode):
    Choose $A \sim \pi$
    Take action A, observe R, ϕ' (feature vector of the next state)
    $V \leftarrow \theta^T \phi$
    $V' \leftarrow \theta^T \phi'$
    $e \leftarrow \gamma\lambda e + (1 - \alpha\gamma\lambda e^T \phi)\phi$
    $\delta \leftarrow R + \gamma V' - V$
    $\theta \leftarrow \theta + \alpha(\delta + V - V_{old})e - \alpha(V - V_{old})\phi$
    $V_{old} \leftarrow V'$
    $\phi \leftarrow \phi'$
until $\phi' = 0$ (i.e you reach a terminal state)

1.2 Part (b)

Replacing traces for parameter updates are defined when the elements of $\phi(s)$ are binary i.e they take a value of 0 or 1. In such cases, the trace is defined as follows:

$$e_t^i(s) = \begin{cases} \gamma\lambda e_{t-1}^i(s) & \phi^i(s) = 0 \\ 1 & \phi^i(s) = 1 \end{cases}$$

where v^i denotes the i^{th} component of a vector v.

Accumulating traces on the other hand are far more generic and can be used however arbitrary $\phi(s)$ is.

Here, $\phi(s)$ is not binary and hence accumulating traces should be used. The definition of replacing trace is shown above and hence isn't a valid method to be applied here.

2 Problem 2

2.1 Part (a)

From the information given about $\phi(s) = \begin{matrix} \phi_1(s) \\ \phi_2(s) \\ \phi_3(s) \end{matrix}$ for states $s = s_1, s_2$ we can write the value function as,

$$V = \phi\theta \tag{1}$$

where $V = \begin{matrix} V(s_1) \\ V(s_2) \end{matrix}$, $\theta = \begin{matrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{matrix}$ and $\phi = \begin{matrix} 1 & -1 & -1 \\ -1 & -1 & 1 \end{matrix}$. Now V can be written as,

$$V = \begin{matrix} \theta_1 - \theta_2 - \theta_3 \\ -\theta_1 - \theta_2 + \theta_3 \end{matrix}$$

V is a point in the 2-dimensional space. Any point in the 2 dimensional space can be represented by suitable values of $\theta_i's$. For instance, $V^*(s_1) = x$ and $V^*(s_2) = y$. Then $\theta = \begin{matrix} \frac{x-y}{2} \\ \frac{-x-y}{2} \\ 0 \end{matrix}$. This can also be inferred from the ϕ matrix. The rank of the matrix is 2 as it has 2 independent rows and the dimensionality of V, $dim(V) = 2$. This means any point in the 2 dimensional space can be represented and hence any value function can be learnt. In other words, this is similar to a look-up table, where instead of using one-hot encodings for states we use a different encoding.

2.2 Part (b)

Linear gradient descent TD(0) update rule is given below.

$$\theta_{t+1} \leftarrow \theta_t + \alpha[R_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)][\nabla_{\theta_t} V_t(s_t)]$$

So given the experience $s_2 - a_2 - (-5) - s_1 - a_1$ we have,

$$\theta_{t+1} \leftarrow \theta_t + \alpha[-5 + \gamma V(s_1) - V(s_2)] \begin{matrix} -1 \\ -1 \\ 1 \end{matrix}$$

i.e

$$\begin{matrix} \theta_{t+1}^{(1)} \\ \theta_{t+1}^{(2)} \\ \theta_{t+1}^{(3)} \end{matrix} = \begin{matrix} \theta_t^{(1)} - x \\ \theta_t^{(2)} - x \\ \theta_t^{(3)} + x \end{matrix}$$

where $x = \alpha(-5 + \gamma V(s_1) - V(s_2))$ and $\theta_t^{(i)}$ denotes the i^{th} component of the vector θ_t.

3 Problem 3

3.1 Part (a)

Here we have a different version of gradient-descent and hence a different update rule. The update rule is given below.

$$\theta_{t+1} \leftarrow \theta_t + \alpha[R_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)][\nabla_{\theta_t} V_t(s_t) - \gamma \nabla_{\theta_t} V_t(s_{t+1})]$$

The experience given is $s_2 - a_2 - (-5) - s_1 - a_1$. The update for each parameter for state s_2 of the MDP is given below.

$$\theta_{t+1} \leftarrow \theta_t + \alpha[-5 + \gamma V(s_1) - V(s_2)]\{ \begin{matrix} -1 & & 1 \\ -1 & -\gamma & -1 \\ 1 & & -1 \end{matrix} \}$$

i.e

$$\theta_{t+1} \leftarrow \theta_t + \alpha[-5 + \gamma V(s_1) - V(s_2)]\{ \begin{matrix} -1-\gamma \\ -1+\gamma \\ 1+\gamma \end{matrix} \} \tag{2}$$

$$\begin{matrix} \theta_{t+1}^{(1)} \\ \theta_{t+1}^{(2)} \\ \theta_{t+1}^{(3)} \end{matrix} = \begin{matrix} \theta_t^{(1)} - (1+\gamma)x \\ \theta_t^{(2)} - (1-\gamma)x \\ \theta_t^{(3)} + (1+\gamma)x \end{matrix}$$

where $x = \alpha(-5 + \gamma V(s_1) - V(s_2))$ and $\theta_t^{(i)}$ denotes the i^{th} component of the vector θ_t.

3.2 Part (b)

From equation 1, we know the following.

$$V = \phi\theta$$

where $V = \begin{matrix} V(s_1) \\ V(s_2) \end{matrix}$, $\theta = \begin{matrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{matrix}$ and $\phi = \begin{matrix} 1 & -1 & -1 \\ -1 & -1 & 1 \end{matrix}$.

Therefore, $V_t = \phi\theta_t$. Premultiplying equation 2 by ϕ we get,

$$V_{t+1} \leftarrow V_t + \alpha[R_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)]y$$

where $y = \begin{matrix} 1 & -1 & -1 \\ -1 & -1 & 1 \end{matrix} \begin{matrix} -1-\gamma \\ -1+\gamma \\ 1+\gamma \end{matrix} = \begin{matrix} -1-\gamma+1-\gamma-1-\gamma \\ 1+\gamma+1-\gamma+1+\gamma \end{matrix} = \begin{matrix} -1-3\gamma \\ 3+\gamma \end{matrix}$. Here $V_t = \begin{matrix} V_t(s_1) \\ V_t(s_2) \end{matrix}$.

4 Problem 4

Linear PSRs are as representative as POMDPs and are very good alternatives to POMDPs. But they do have some advantages over POMDPs. They are listed below.

- POMDP learning algorithms may not give optimal solutions as they may converge to a local minimum or a saddle point. This is because all their states are equipotential. PSRs on the other hand are more grounded in data and hence don't have this symmetry problem.
- PSRs are more dependent on the data and can learn complex representations. You just need more tests if we constrain the function f_t to be linear. As a result, learning complex representations is easier and hence PSRs generalise better.
- POMDPs depend on accurate prior models which may not be available in all problems. PSRs on the other hand are less dependent on this as they just take the actions and observations for constructing tests.
- Learning POMDPs without these prior models is difficult and PSR is a very good alternative if prior models aren't available.

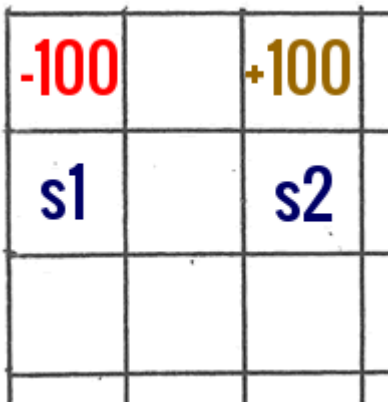
5 Problem 5

POMDPs have partially observed states and hence the agent doesn't have the complete information about the state. So, it is tougher to solve when compared to solving an MDP.

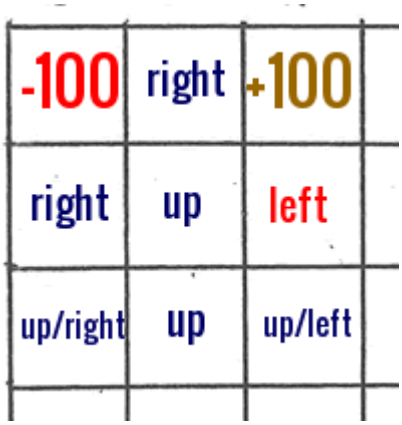
Q-MDP is one way to solve a POMDP. Here you assume that you have complete information about the MDP i.e you assume the minimal information about the state as the complete information and learn action-value or value functions. When you are executing the policy, you use some heuristic to choose the action based on the belief state for that particular state.

Consider 2 states s_1 and s_2 which have the same representation as they are partially observed. Lets call this representation f. Now, $Q(s_1, a)$ and $Q(s_2, a)$ will be learnt from the samples where state is observed as f. This means, if the agent is s_1 and makes a transition after taking action a to some other state s_x that is used to update $Q(s_1, a)$. But since we have partial information about the state space, that transition updates $Q(f, a)$. Similarly, the transitions from state s_2 after taking action a will update $Q(f, a)$. In other words, a Q function is shared between 2 different states which have identical representations when observed partially.

Lets assume the case that, s_1 is close to a state which gives a high positive reward and s_2 is close to a state which gives a high negative reward. Now, when trained $Q(f, a)$ will have updates which partially get cancelled due to the positive and negative rewards. The gridworld is shown below.



If the agent reaches the top-right most square it gets a positive reward of +100 and if it lands up in the top-left most square it gets a negative reward of −100. States s_1 and s_2 have the same representation f. As a result $Q(f, a)$ won't be learned properly. The reason being, $Q(f, UP)$ gets cancelling updates as both the transitions $s_1 - UP$ and $s_2 - UP$ update $Q(f, UP)$. So, for state s_1, s_2 UP action won't be optimal. In fact if the other states have unique representations (even though they are partially observed), the policy learnt won't be optimal. The policy learnt in this case is shown below.



Now, the policy for the state s_2 is not optimal. This is because we have solved the POMDP in a Q-MDP fashion which doesn't result in optimal policies in all cases.

The policies computed using Q-MDP method will be optimal if the states which have similar representations in the partially observed space are similarly rewarding. In that case, the updates made to $Q(f, a)$ are more related and hence the policy learnt will be optimal.

6 Problem 6

6.1 Part (a)

When we use the 'ostrich' approach for handling partial observability, the resulting MDP has the following components:

- State-space $S = \{f_1, f_2, f_3, ..\}$ where $f_i's$ denote the partially observed state.
- Action set $A(f)$ denoting the set of valid actions which can be taken from the PO (partially observed) state f for all states $f \epsilon S$.
- The transitions are modelled using $p(s_{t+1} = f' | s_t = f, a_t = a)$.

6.2 Part (b)

We are operating in a modified state-space where same observed states don't necessarily correspond to same environmental states, which adds up the stochasticity factor of the environment. So,

- Q-learning in highly stochastic environments doesn't perform well as there is increased maximisation bias in this case. This results in divergence or slower learning. Since we have a max operation in Q-learning, if we see where the backups come from they tend to backup from some particular states into the future, and if estimates of those states aren't good enough, errors tend to get accumulated over time. So, Q-learning is not preferred.

- Due to partial observability, states with dissimilar payoffs (not similarly rewarding) tend to have the value functions. As a result, using MC method doesn't make sense as it gives the maximum likelihood estimate for the given data points. Consider the example given in problem 5. In that case $Q(f, UP)$ gets contradicting updates as both s_1 and s_2 are observed as f. So, the estimate of values/Q-values will have a very high variance and hence aren't reliable. So, MC is not a preferred learning technique.

- SARSA! Compared to the other two, SARSA is better. This is because SARSA updates consider the actual transition i.e for the experience $s_1 - a_1 - r_1 - s_2 - a_2$, $Q(s_1, a_1)$ is updated using r_1 and $Q(s_2, a_2)$. Inspite of partial observability, updating from states which are actually seen in the episodes leads to lesser error in the estimate. This is because of the following reason. Consider 2 states s_1, s_2 which have the same partial observation f. When these are encountered in the trajectory, the next-states of these states s_{1next}, s_{2next} can have a different partial observation (or their next-states further into the future will have a different partial observation) and hence their estimates are more reliable. Thus estimates for $Q(s_1, a_1)$ are better here. So, SARSA results in better policies than Q-learning/MC.

- According to *Loch and Singh* 1998 (ref. 8), it is empirically shown that SARSA(λ) results in finding the best memoryless policy (states don't contain history of how you landed up in that position) in POMDPs. Even though, using eligibility traces in SARSA(λ) also contributes to the performance, even with $\lambda = 0$ (no eligibility trace), SARSA seems to be a better alternative when compared to others.

6.3 Part (c)

Policy search methods can be applied to POMDPs and they have the following advantages over value-function based methods.

- If the POMDP is not Markov, then Bellman updates aren't correct and hence we won't be able to find a stable Q.

- It is better to learn stochastic policies in the case of POMDPs as we have only partial observability of the environment. So it makes sense to use stochastic policies to overcome this "partial observability problem" as at times we might take the right action. Deriving stochastic policies from value-function based methods may not be optimal ones and value-function methods may be unstable.

7 Problem 7

It depends. According to the original algorithm mentioned in the paper, it results in recursively optimal policies. The HAM-Q learning algorithm is described below.

HAM Q learning

The policy learnt in a HAM is what output transition should be chosen at every choice point. So, say the current environment state is t, the current machine state is n and the environment and machine state at the previous choice point is s_c and m_c respectively. Let a be the action taken at the previous choice point. Now, we can do a Q-learning update for $Q([s_c, m_c], a)$ where the "SMDP state" is a concatenation of the current machine state and the environment state (core MDP state). Now, we need to accumulate discounted rewards which is done as follows. For every transition from state s to state t with reward r

and discount β, the following update is done.

$$r_c \leftarrow r_c + \beta_c r$$

$$\beta_c \leftarrow \beta_c \beta$$

Whenever there is a transition to a choice point,

$$Q([s_c, m_c], a) \leftarrow Q([s_c, m_c], a) + \alpha[r_c + \beta_c V([t, n]) - Q([s_c, m_c], a)]$$

and then $r_c \leftarrow 0$ and $\beta_c \leftarrow 1$.

Now, if we consider the crucial steps in the above algorithm we can conclude what type of optimality it acheives.

- The "SMDP state" is assumed to be a concatenation of the "current machine state" and the environment state. This means that this particular task doesn't have any idea about what are the parent tasks, the history of how the agent landed up in this particular machine state. Therefore the agent tries to improve performance based on the local goals setup for this particular task and hence the policy obtained need not be hierarchically optimal.
- Also, it depends on how the reward function r is defined. If we try to optimise using a global reward function, then the local goals set up will have rewards based on the history. So, if and only if the $[s_c, m_c]$ is representative enough about history somehow the agent can learn hierarchical optimal policies.

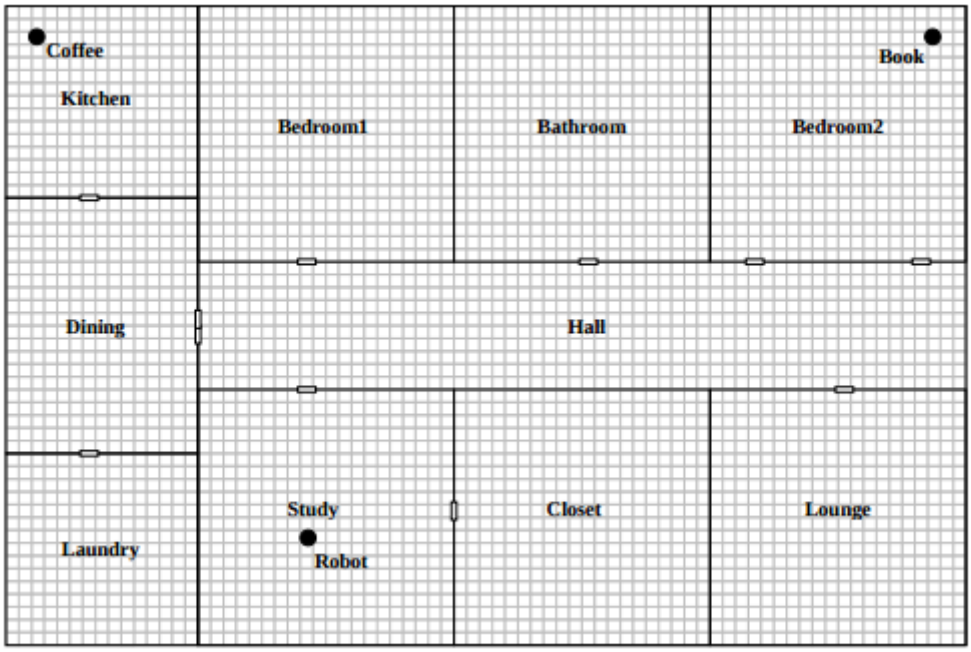
In the paper, the "SMDP state" doesn't include history and rewards are taken from the MDP M. So, in general it can't learn hierarchical optimal policies. Since it tries to optimise performance locally and hence acheive local goals, the policies learnt are recursively optimal. Every subtask is optimal in itself. This results in more state-abstracted policies at the cost of optimality.

We can make the algorithm learn hierarchically optimal policies by making some modifications.

- We should include the call-stack or the history of machine states visited so that the agent has an idea of the global task.
- We should modify the reward functions suitably so that the agent chooses actions to maximise global rewards.

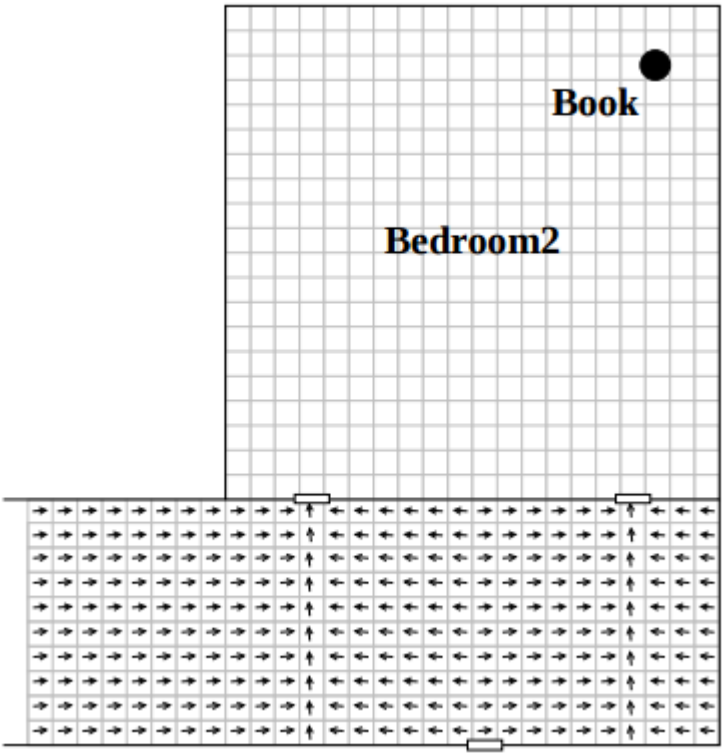
The following example will make this idea clear.

Suppose we have the following environment. And the agent is initially in the hall. The agent has to fetch the book (gets a high positive reward of $+100$) and it is trying to learn this in a Hierarchical RL setup. Say, there is a penalty of -1 for every timestep. Assume that a suitable discount factor is used.



Say the task is split as following: $move(hall, bedroom2 - door)$ and $move(bedroom2 - door, book)$. Lets consider 2 cases.

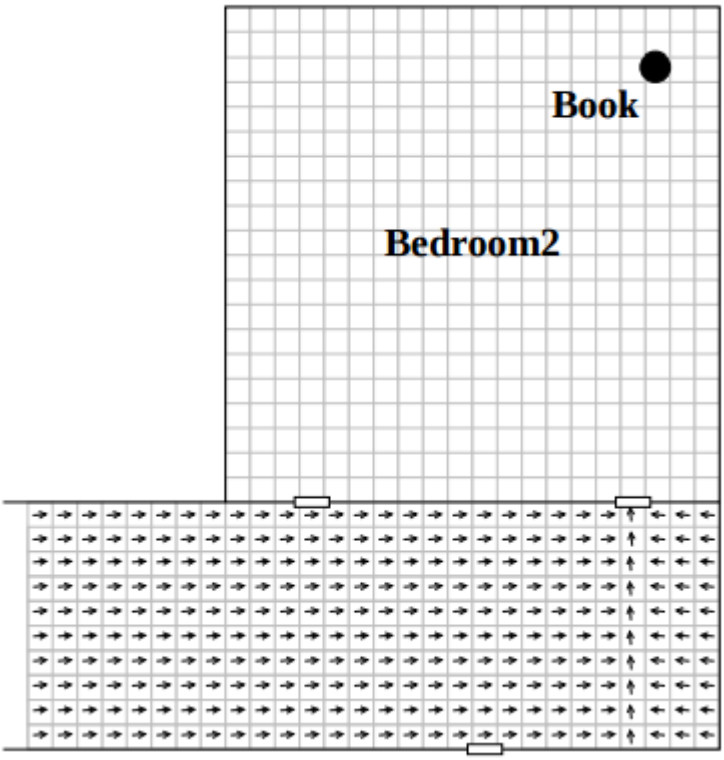
In the first case, say we are using 2 HAMs for the 2 tasks and the $1st$ HAM independently solves the first problem without the idea of the end goal. We will get a recursively optimal policy then as it best exploits the hierarchy of tasks. The policy obtained is shown below.



This policy optimises the number of steps to enter the room as the $1st$ HAM has no idea of fetching the book.

Suppose, we modify the rewards for the first task so that the agent knows about the end goal and hence

tries to maximise the total reward to be obtained. This is what we do in case 2. The policy learnt here is hierarchically optimal as it not only confirms to the hierarchy, but also chooses each individual behaviours so that they obtain the overall best. The policy learnt in this case is shown below.



This policy optimises the total number of steps required to fetch the book. As a result actions for the locations between the 2 doors in hall change.

8 Problem 8

Dietterich has specified 5 conditions for safe-state abstractions for the $MAXQ$ framework. But the $MAXQ$ framework provides both the hierarchy i.e the $MAXQ$ graph and the decomposition of the value function. We can use the hierarchy but not use the decomposition. Instead of that, we can use some other learning method on top of this hierarchy.

In such cases, a subset of those conditions need to be satisfied to ensure safe-state abstractions. They are listed below.

- **Subtask irrelevance**
  A set of variables are irrelevant to a subtask i if their exists a partition (into X and Y) of the state variables of the original MDP such that the following property holds.

  $$P^{\pi}(s', N|s, j) = P^{\pi}(x', N|s, j)P^{\pi}(y'|x, y, j)$$

  where x and x' give values for variables in the set X and y and y' give values for variables in the set Y.

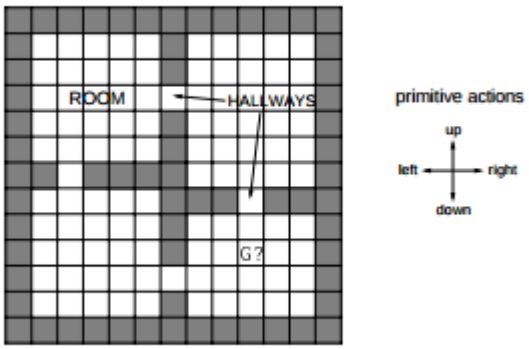
- **Leaf irrelevance**
  Consider a primitive action a. If for any 2 states s_1 and s_2 which differ only in their values for some set of variables and the following condition holds true, then that set of variables is irrelevant.

  $$\sum_{s_1'} P(s_1'|s_1, a)R(s_1'|s_1, a) = \sum_{s_2'} P(s_2'|s_2, a)R(s_2'|s_2, a)$$

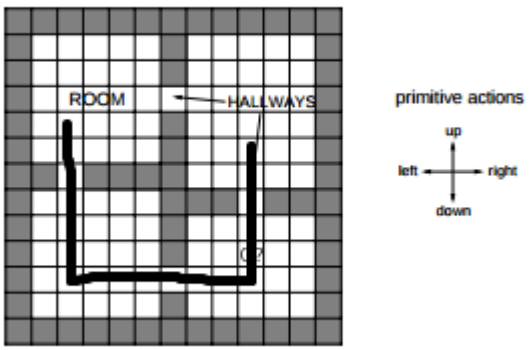
In other words if the expected pay-off from the state after taking a primitive action is the same for any 2 states, then the state variables where these 2 states differ are irrelevant.

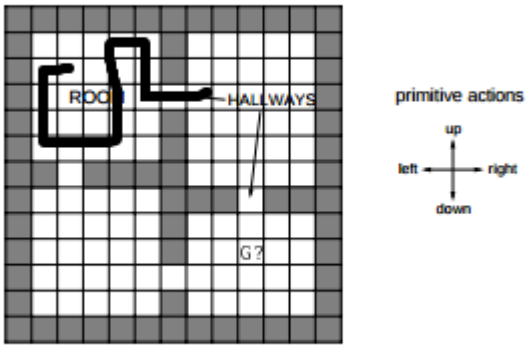
9 Problem 9

Options are temporally extended actions and learning Q-values over options has its own advantages. Introducing options reduces unnecessary exploration and as a result the agent learns faster. But all these hold true only when the option is defined properly. If we define bad options - i.e those when incorporated leads to sub-optimal behaviour, learning won't be faster. In fact, after a point of time the agent will begin to ignore the options defined and start executing primitive actions in all states. This happens because the option-value for those options goes much below than the expected reward obtained when primitive actions are taken. Consider the case of the gridworld given in the question. The gridworld is shown below.



The agent is provided with options to exit from one room to the other. Say the option is defined by choosing a longer path to go from one room to the other (shown in the figure below; $room(top-left)\,to\,room(top-right)$).





In such cases, if the agent chooses to use the option (which is sub-optimal), it will accumulate huge negative rewards (as more steps are taken). Assuming the agent uses an explorative policy, the agent will learn that using these options causes more harm than good. Instead it can learn to exit from the room using primitive actions themselves. If we design bad options, this is what happens. After some point during training, option-values tend to be very low and choosing primitive actions in those states result in higher pay-offs. As a result, agent tends to discard options and hence the problem now reduces to Q-learning without options (the usual case). This means the advantages of options - improved exploration and faster learning will not be observed. This might be the case with the gridworld in the question. We have to make sure options are designed in a proper way so as to exploit its advantages.

References

1. Stolle, Martin, and Doina Precup. "Learning options in reinforcement learning." International Symposium on Abstraction, Reformulation, and Approximation. Springer Berlin Heidelberg, 2002.

2. Parr, Ronald, and Stuart Russell. "Reinforcement learning with hierarchies of machines." Advances in neural information processing systems (1998): 1043-1049.

3. Dietterich, Thomas G. "The MAXQ Method for Hierarchical Reinforcement Learning." ICML. 1998.

4. Dietterich, Thomas G. "State Abstraction in MAXQ Hierarchical Reinforcement Learning." NIPS. 1999.

5. Ryan, M. R. K. "1 Hierarchical decision making." Handbook of Learning and Approximate Dynamic Programming (2004): 203-232.

6. Littman, Michael L., Richard S. Sutton, and Satinder Singh. "Predictive representations of state." Advances in neural information processing systems 2 (2002): 1555-1562.

7. Perkins, Theodore J. "Reinforcement learning for POMDPs based on action values and stochastic optimization." AAAI/IAAI. 2002.

8. Loch, John, and Satinder P. Singh. "Using Eligibility Traces to Find the Best Memoryless Policy in Partially Observable Markov Decision Processes." ICML. 1998.

9. Murphy, Kevin P. "A survey of POMDP solution techniques." environment 2 (2000): X3.