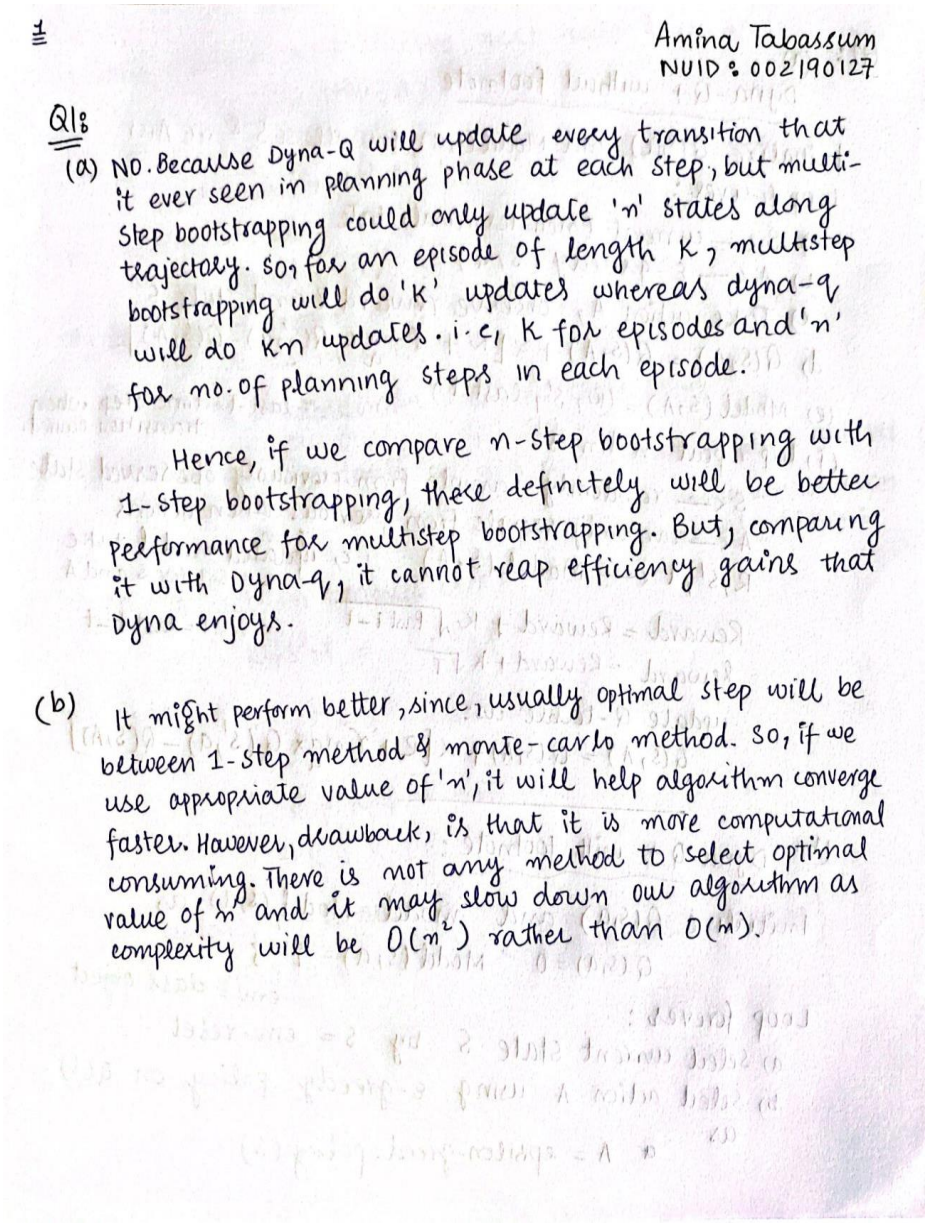
Amina Tabassum

NUID: 002190127

Question 01:



Amina Tabassum
NUID : 002190127

Q1:

(a) NO. Because Dyna-Q will update every transition that it ever seen in planning phase at each step, but multi-step bootstrapping could only update 'n' states along trajectory. So, for an episode of length K, multistep bootstrapping will do 'K' updates whereas dyna-q will do Kn updates. i.e, K for episodes and 'n' for no. of planning steps in each episode.

Hence, if we compare n-step bootstrapping with 1-step bootstrapping, these definitely will be better performance for multistep bootstrapping. But, comparing it with Dyna-q, it cannot reap efficiency gains that Dyna enjoys.

(b) It might perform better, since, usually optimal step will be between 1-step method & monte-carlo method. So, if we use appropriate value of 'n', it will help algorithm converge faster. However, drawback, is that it is more computational consuming. There is not any method to select optimal value of n and it may slow down our algorithm as complexity will be $O(n^2)$ rather than $O(n)$.

c)

I implemented n-step SARSA on blocking maze environment and results are shown below. It can be seen from plots that nstep SARSA does not perform as well as Dyna-Q. N-step bootstrapping is better than 1-step bootstrapping I.e, n-step SARSA is better than 1-step SARSA but when it comes to planning algorithms DynaQ, multistep bootstrapping is not better than Dyna-Q because Dyna-q performs number of episodes * number of planning steps updates whereas multistep bootstrapping just performs number of episodes times steps updates.

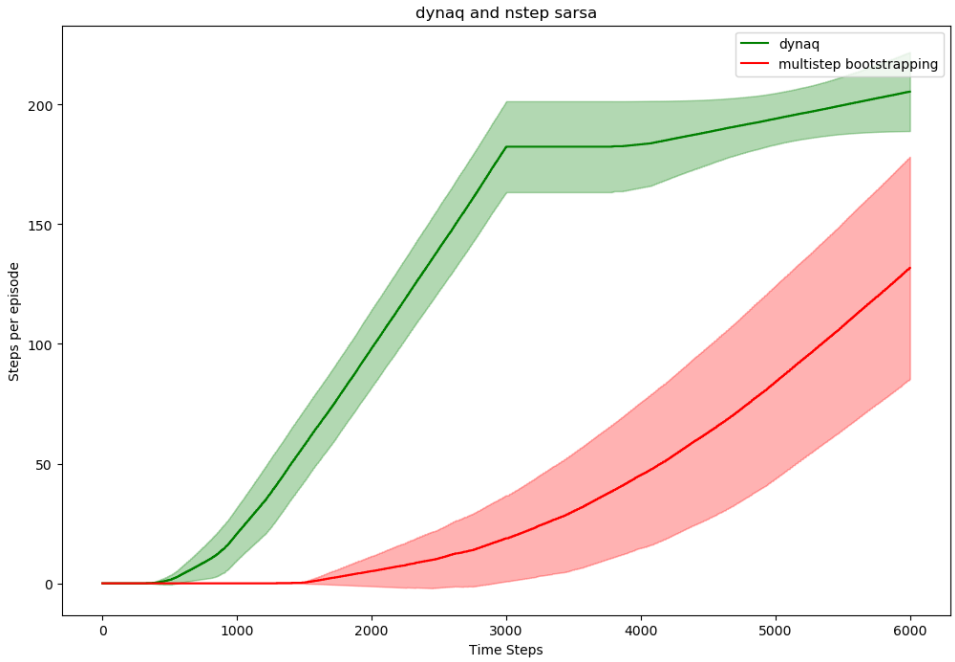


Figure1: Dyna-Q and Multistep Bootstrapping

Coming to experimental verification of part b) when n-step returns are implemented in planning step, the running complexity of algorithm increases by O(n2) times number of episodes. In this case, algorithms perform both exploration and exploitation. It takes a lot of time in the planning phase. In this case, multistep bootstrapping updates are performed in every planning step. So, initially, the cumulative reward is slow. In other words, random sampling and multistep bootstrapping of already visited state-action pairs wastes a lot of computation on trivial updates because most of updates did not yet receive the backpropagated reward from goal state. However, when more states receive an update, the amount of wasted compute is reduced during subsequent episodes and increased timesteps. Hence, dynaq with nstep sarsa in planning phase will outperform Dynaq in long run but it is computationally expensive and time consuming.

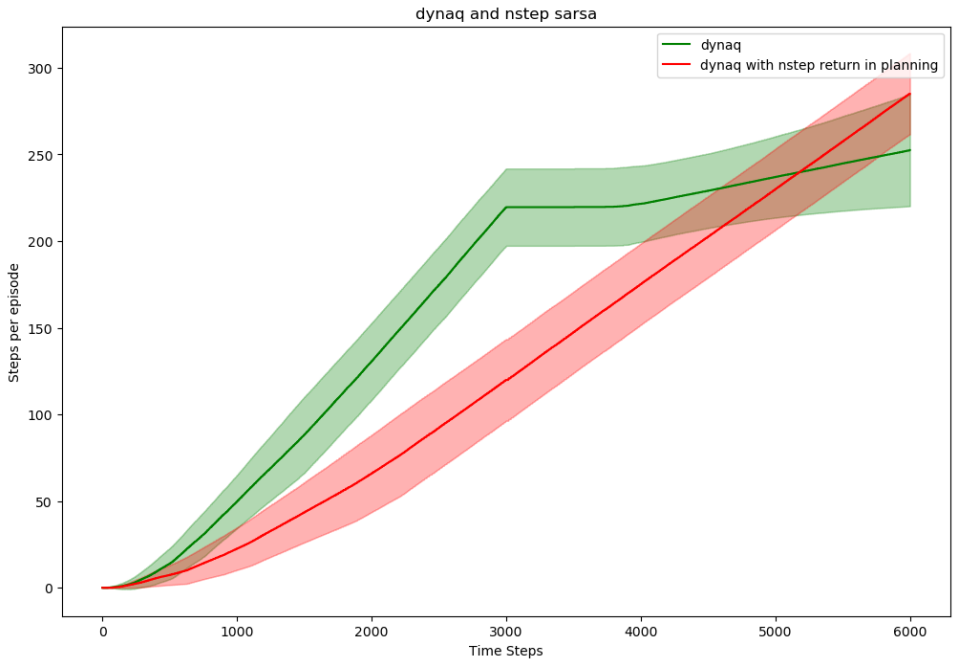
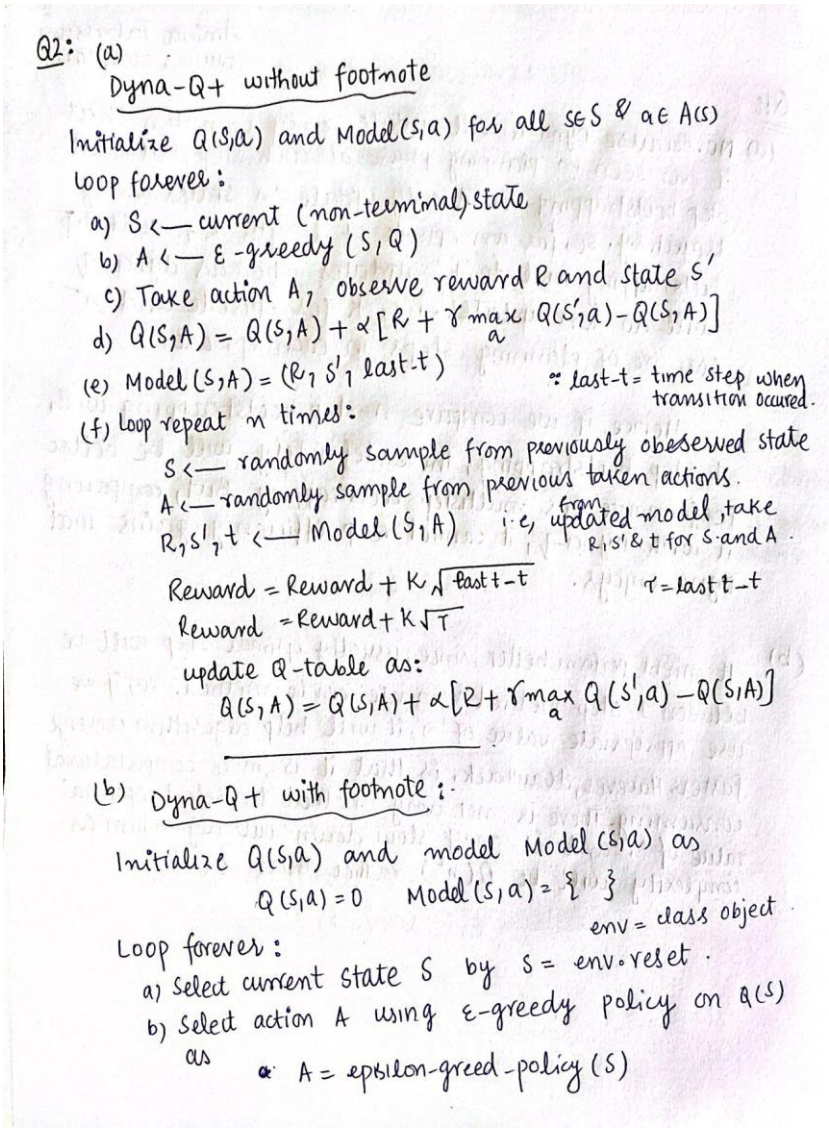


Figure 2: Dyna_Q and Dyna-Q with n-step return in planning phase

Question 02:

a)



Q2: (a)

Dyna-Q+ without footnote

Initialize $Q(S,a)$ and $Model(S,a)$ for all $S \in S$ & $a \in A(S)$

Loop forever:

a) $S \leftarrow$ current (non-terminal) state

b) $A \leftarrow \varepsilon$-greedy (S,Q)

c) Take action A, observe reward R and state S'

d) $Q(S,A) = Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$

(e) $Model(S,A) = (R, S', last\text{-}t)$ ∵ last-t = time step when transition occured.

(f) Loop repeat n times:

$S \leftarrow$ randomly sample from previously obeserved state

$A \leftarrow$ randomly sample from previous taken actions.

$R, S', t \leftarrow Model(S,A)$ i.e, from updated model, take R, S' & t for S and A.

$Reward = Reward + K\sqrt{last\,t - t}$ $\tau = last\,t - t$

$Reward = Reward + K\sqrt{\tau}$

update Q-table as:

$Q(S,A) = Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$

(b) Dyna-Q+ with footnote:

Initialize $Q(S,a)$ and model $Model(S,a)$ as

$Q(S,A) = 0$ $Model(S,a) = \{\}$

Loop forever: env = class object

a) Select current state S by S = env.reset.

b) Select action A using ε-greedy policy on $Q(S)$ as A = epsilon-greed-policy(S)

c) Take action A and observe next state & reward:

$R, S' = \text{step}(A)$

d) update Q-table as:

$$Q(S,A) = Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$$

e) update model:

$\text{Model}(S,A) = (R, S', t')$

f) loop for n times as:

```
for _ in range(n):
    S = randomly selected state from
        previously visited states.
    A = randomly selected action from previous actions.
```

→ if action has not been previously taken:

```
if A not in Model(S,A):
    R, S', t = 0, S, 0
else:
    R, S', t = Model(S,A)
```

$R = R + K\sqrt{t' - t}$

$= R + K\sqrt{T}$

g) Update Q-table as:

$$Q(S,A) = Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$$

b)

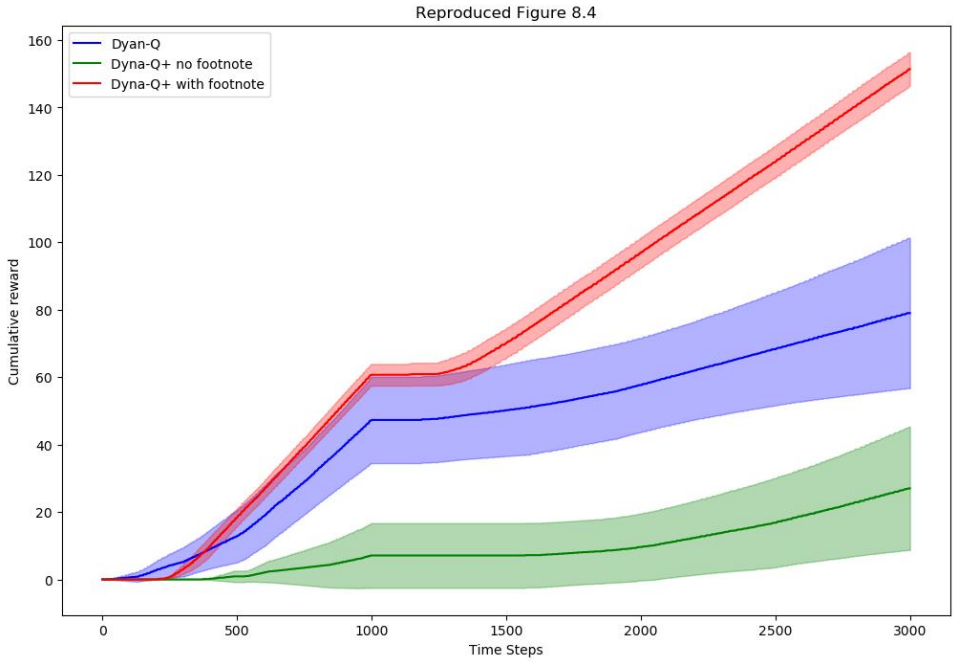


Figure 3: Dyna-Q and Dyna-Q + for Blocking Maze

In this case, both Dyna-Q and Dyna-Q+ are implemented for Blocking maze enivronment. It can be seen from plots that Dyna-Q+ with footernote performs better than both Dyna-Q and Dyna-Q+ without footnote. It has a high learning rate, higher cumulative reward and hence low variance.

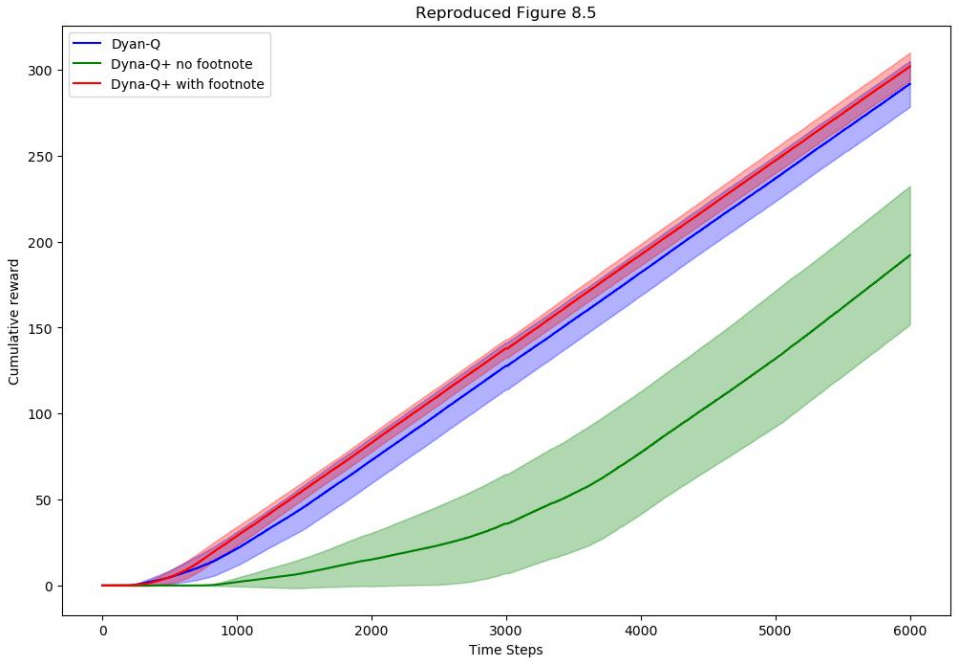


Figure 4: Dyna-Q and Dyna-Q + for Shortcut Maze

c) The footnote does matter. It can be seen from the above results that learning curve with footnote rises rapidly and variance is small. In other words, DynaQ+ without footnote has high variance and learning curve rises very slowly. Without footnote, the actions that have not been taken will not be updates during planning phasea and will appear in planning phase only when agent actually takes those actions in real environment and it depends on exploration facor epsilon which makes learning rate slow in new environments.

Question No: 03

Question NO: 03

In both methods, extra bonus either applied during action selection or to bonus reward will encourage the agent to chose those actions that have not been chosen for long time but UCB is used for K-arm bandit problem and it does not have transitions between different states, and it only need to pick action once to know how good is action now.

In Dyna-Q+, the bonus has actually been used in action value computation. If in the planning phase, the update time is large enough, the action value will be infinity, given learning target $R + \text{Bonus} + \gamma \max_a Q(s', a)$.

Dyna-Q+:

Pros: Bootstrap update based on following state could handle state transition better than UCB.

Cons: Extra bonus reward makes all action values keep increasing, which makes action value deviate from true value & not realistic.

UCB:

Advantages:

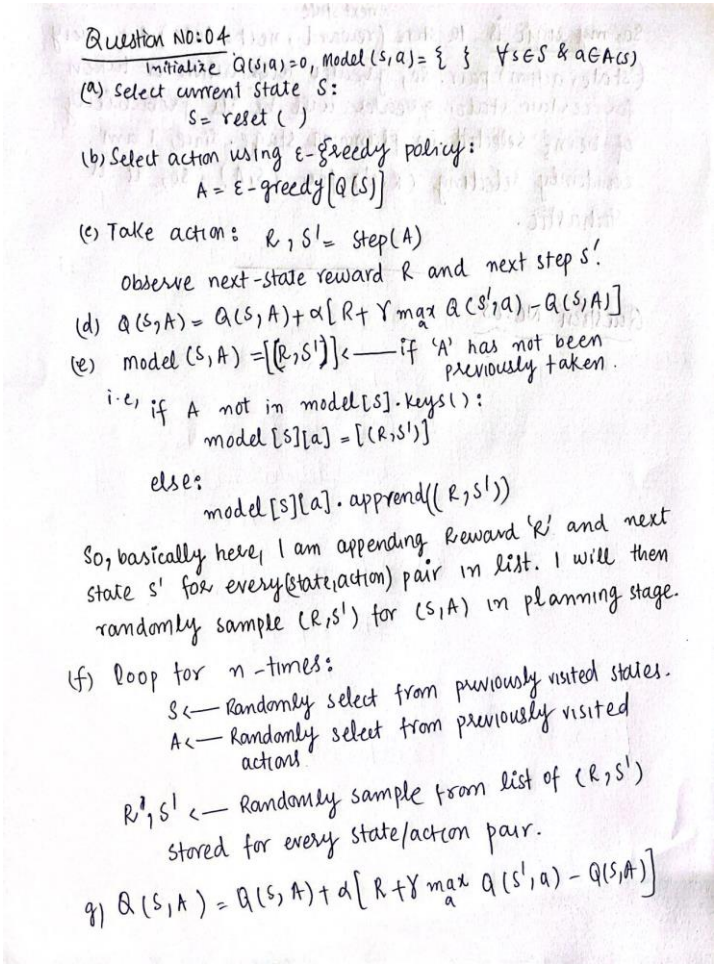
Since, we only need to record time that every state-action pair last appears.

Disadvantages:

Since, it does not update action value based on next state it reaches to, it cannot handle state transition well.

Question No: 04

a)



Question NO:04

Initialize $Q(s,a)=0$, $Model(s,a)=\{\}$ $\forall s \in S$ & $a \in A(s)$

(a) Select current state S:

S = reset ( )

(b) Select action using ε-greedy policy:

A = ε-greedy [Q(S)]

(c) Take action: R, S' = step(A)

observe next-state reward R and next step S'.

(d) $Q(S,A) = Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$

(e) model (S, A) = [(R, S')] <— if 'A' has not been previously taken.

i.e, if A not in model[S].keys():

model[S][a] = [(R, S')]

else:

model[S][a].append((R, S'))

So, basically here, I am appending Reward 'R' and next state S' for every (state, action) pair in list. I will then randomly sample (R, S') for (S, A) in planning stage.

(f) loop for n-times:

S <— Randomly select from previously visited states.

A <— Randomly select from previously visited actions

R', S' <— Randomly sample from list of (R, S') stored for every state/action pair.

g) $Q(S,A) = Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$

So, my logic is to store (reward, next-state) for every (state, action) pair. So, greater this action is taken for certain state, greater will be its probability of being selected in planning stage. Since, I am randomly selecting (R, S') for (S, A), so, it is stochastic.

b)

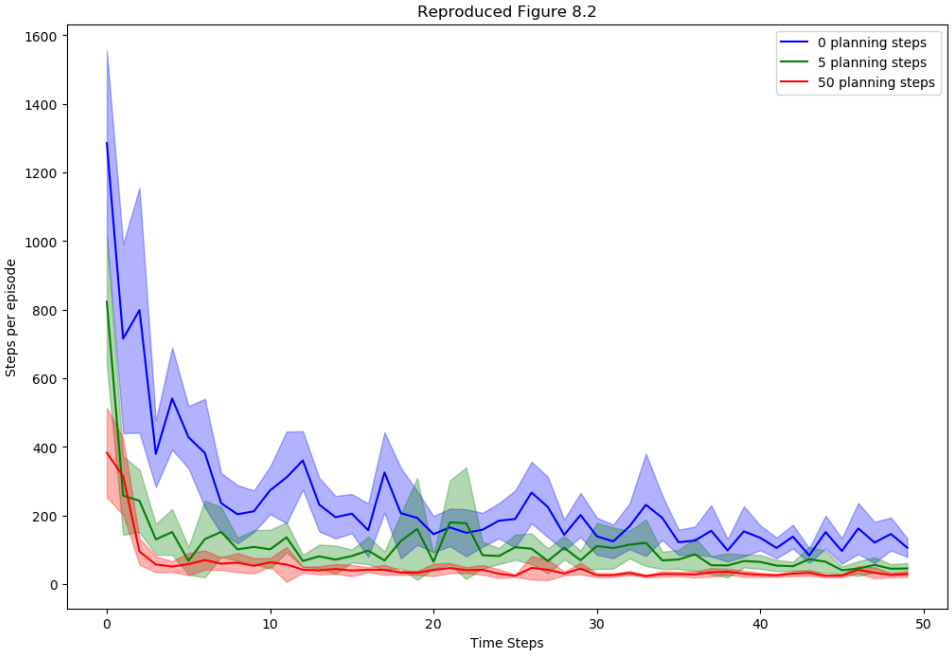
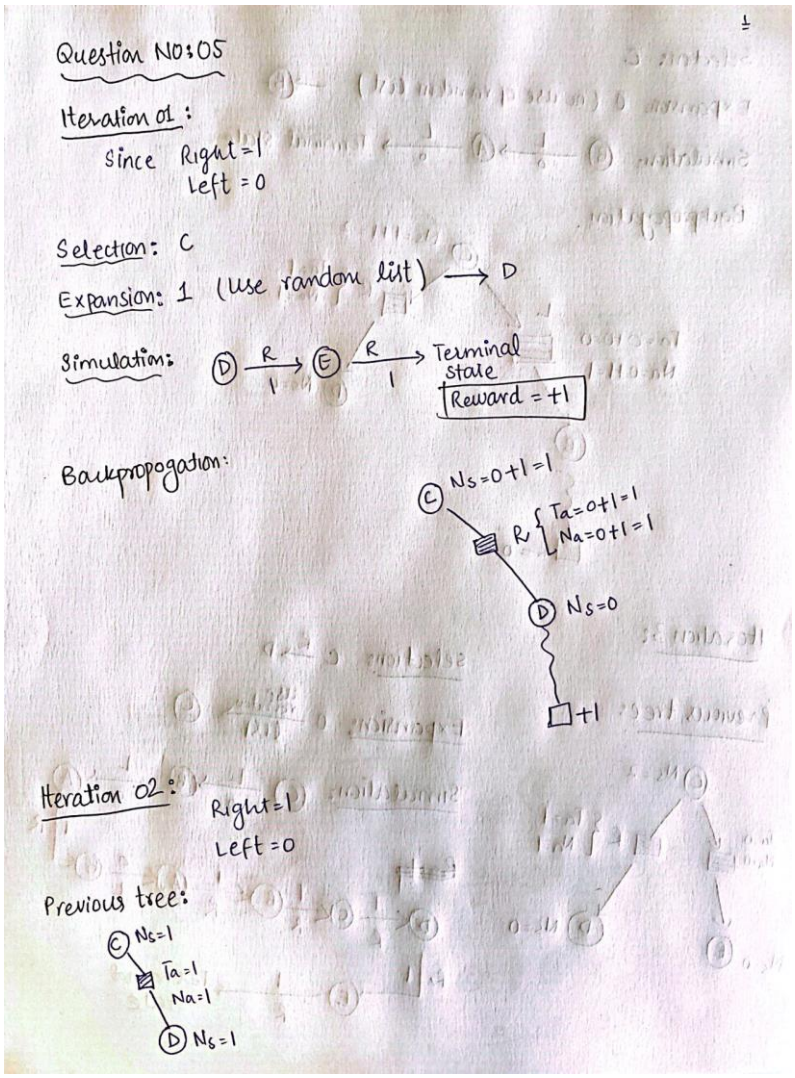


Figure 5: Dyna-Q and Dyna-Q + for Stochastic Windy Gridworld

Question No: 05

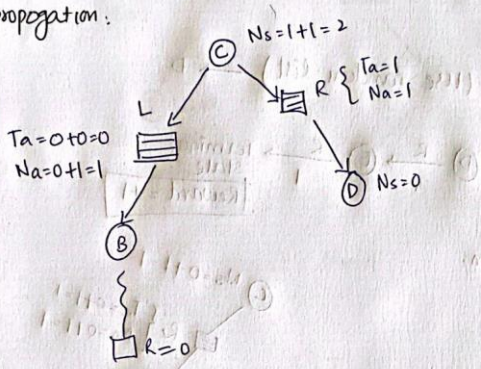


Selection: C

Expansion: 0 (no use of random list) → (B)

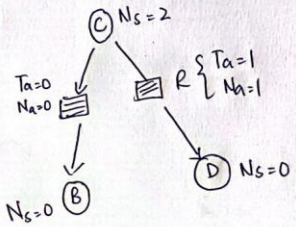
Simulation: (B) —L/0→ (A) —L/0→ Terminal state.

Backpropogation:



Iteration 3:

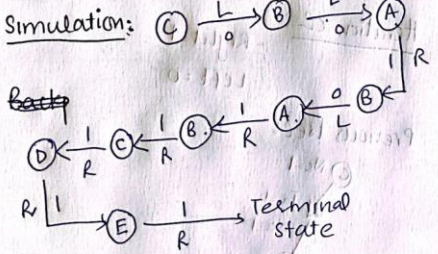
Previous tree:



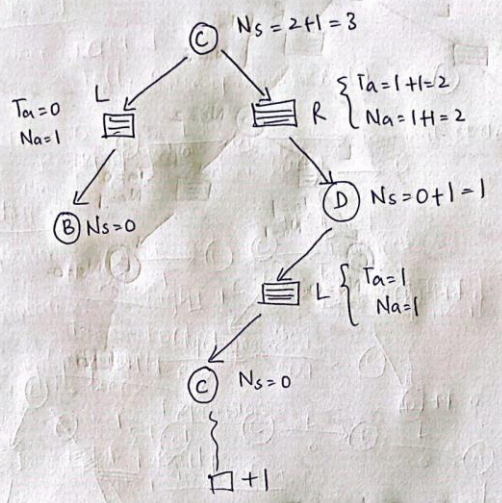
selection: C —R→ D

Expansion: 0 —use random list→ (C)

Simulation:



Final tree



Iteration 04:

Selection: C → (R) D

Expansion: 1 → (no use of random dist) E

Simulation: E →(0,L) D →(0,L) C →(1,R) D →(0,L) C →(1,R) D →(1,R) E →(0,L) D

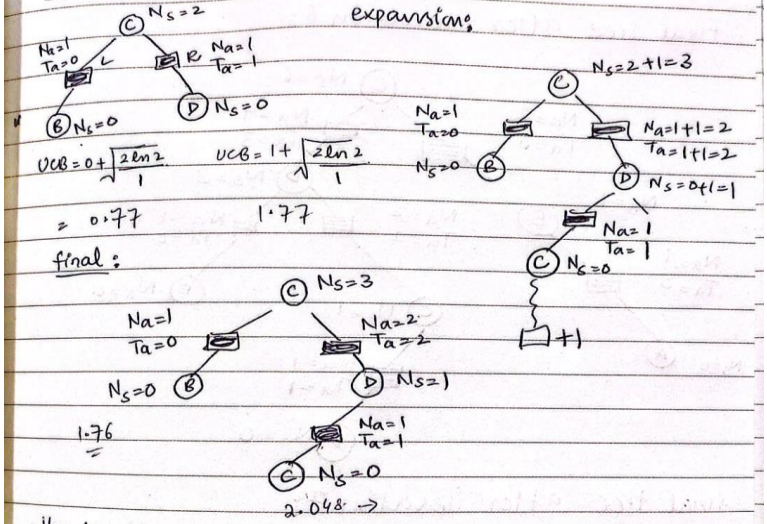
D ←(0,L) C ←(1,R) B ←(0,L) C ←(0,L) D ←(1,R) C ←(1,R) B ←(0,L) C ←(0,L) D

D →(1,R) D →(0,L) C →(0,L) B →(0,L) A →(0,L) Terminal state

Iteration #03

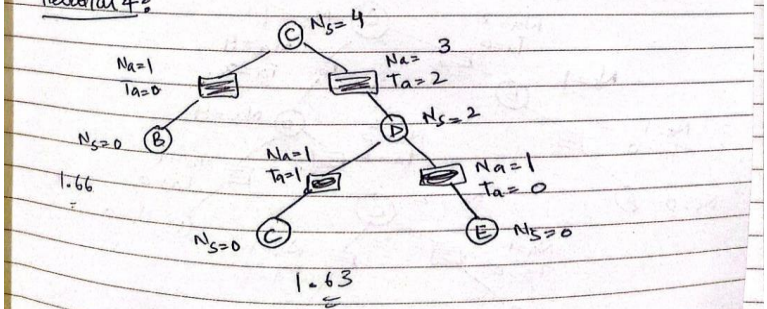
selection: C → R → D

expansion:



final:

Iteration 4:

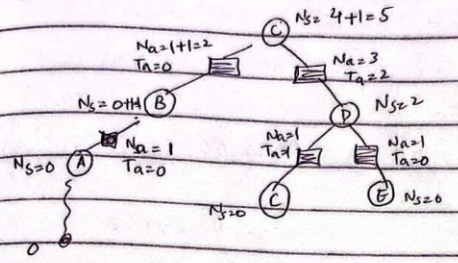


selection: C → B A-B-C-D-E

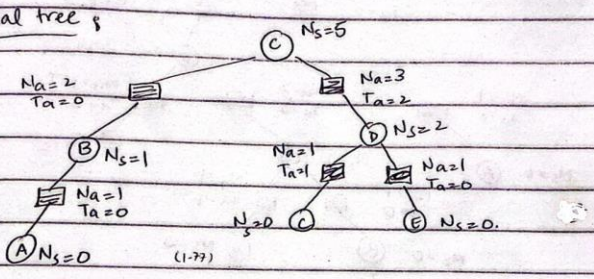
expansion: B → A (0 use).

Iteration 5:

simulations: A → (+1)



final tree:

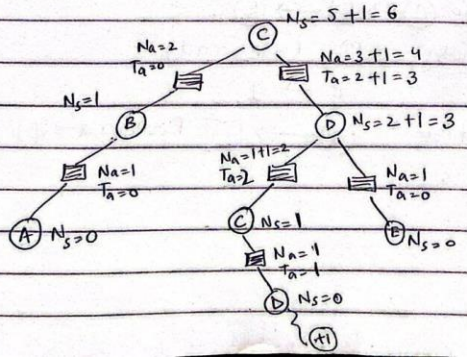


(1-77)

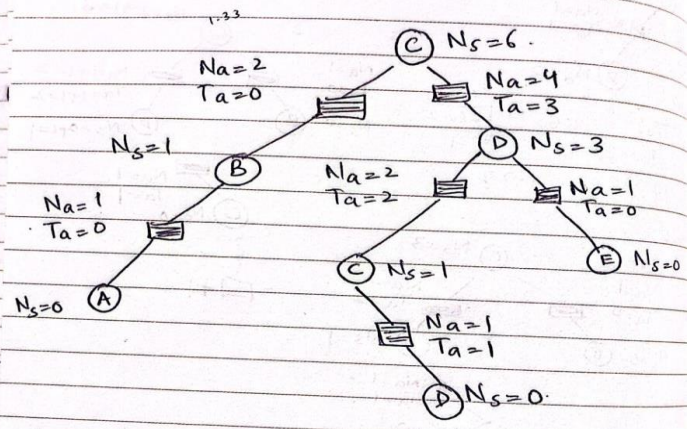
Iteration 6: selection: C → D → C.

expansion: (1) C → D

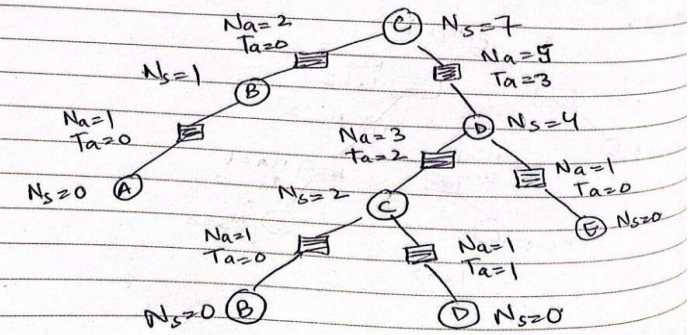
simulations: (D) → 1 → (E) → 0 → (D) → 1 → (E) → 1 → (0) → +1



final tree after iteration 6 :



final tree after iteration 7 :



A – B – C – D – E

Iteration 07:

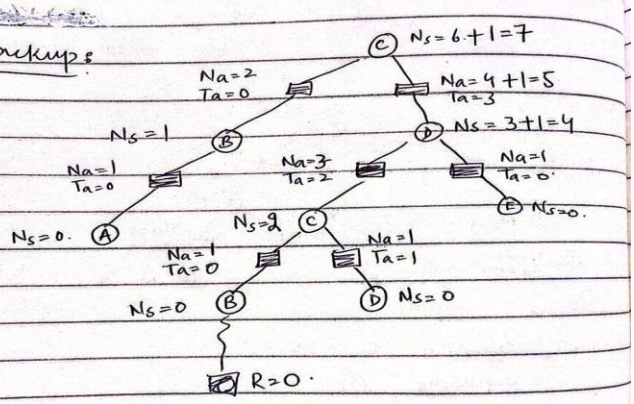
Selection: (C) → (D) → (C)

expansion: (C) → (B) (0) no use of rnd list

Simulation:

(B) $\xrightarrow{0}$ (A) $\xrightarrow{1}$ (B) $\xrightarrow{1}$ (C) $\xrightarrow{0}$ (B) $\xrightarrow{1}$ (C) $\xrightarrow{0}$ (B) $\xrightarrow{0}$ (A) $\xrightarrow{0}$ (R=0)

Backup:



Iteration 08:

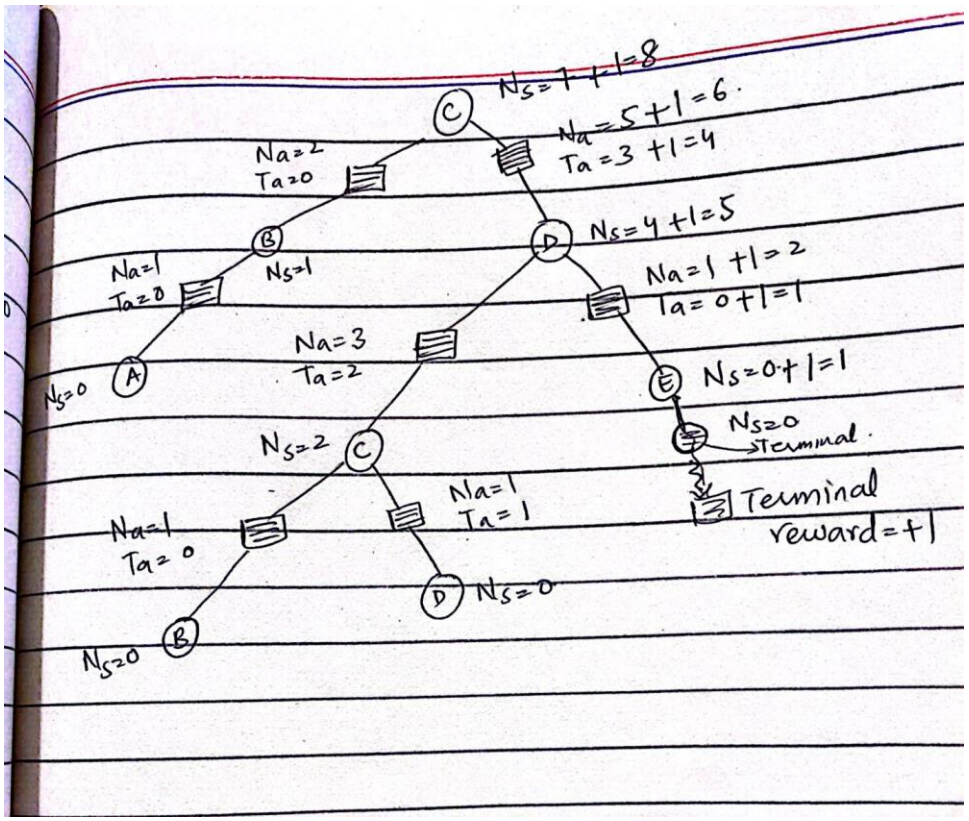
Selection: (C) → (D) → (E)

expansion: (1) (use rnd)

E → 1

Simulation: (1) → ☐ Reward = +1

Backup:


C
Ns=7+1=8
Na=5+1=6
Ta=3+1=4
Na=2
Ta=0
B
Ns=1
Na=1
Ta=0
A
Ns=0
D
Ns=4+1=5
Na=1+1=2
Ta=0+1=1
Na=3
Ta=2
E
Ns=0+1=1
Ns=0
Terminal
Terminal
reward=+1
Ns=2
C
Na=1
Ta=1
Na=1
Ta=0
D
Ns=0
B
Ns=0

Final tree after iteration 8: