Assignement: ex6

Ian Steenstra

November 2024

1 Problem 1

- **(a):** If the question is referring to Dyna-Q without planning (n=0), then any of the n-step methods in Chapter 7 would perform better because the n=0 agent updates the value of only one state-action pair on each episode—the last one on the path to the goal—whereas n-step methods update n state-action pairs on each episode. With n=0, learning is slow where each episode provides information about the optimal action from only one state. n-step methods, on the other hand, allow learning from n transitions on each episode, allowing much more rapid propagation of reward information through the state-action space. However, Dyna-Q with planning (e.g., n=10), is still likely to be better. However, it could depend on the environment. For instance, if the environment was highly stochastic, learning a perfect model would be difficult or impossible. In such cases, n-step methods might eventually outperform Dyna-Q, as its model might mislead the planning process.
- **(b):** Yes, n-step returns can be used in Dyna-Q's planning phase (f). A significant advantage is the potential for faster planning. N-step returns can propagate value information more quickly than one-step returns, potentially reducing the number of planning updates required to find a near-optimal policy. However, there are disadvantages. Calculating n-step returns requires significantly more computation per update than one-step returns, especially in stochastic environments, potentially offsetting gains in planning efficiency. Furthermore, storing the experience needed for n-step updates requires more memory than for one-step updates, particularly in stochastic environments. Finally, n-step returns cannot be used to update the agent until n time steps have elapsed, which can be problematic if low-latency action selection is necessary.

2 Problem 2

- **(a):**

  Tabular Dyna-Q+

Initialize:

- $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A(s)$
- $\tau(s, a) = 0$ for all $s \in S$ and $a \in A(s)$ *(Initialize time since last visit)*
- n, the number of planning steps
- κ, the exploration bonus scaling factor

Loop forever:

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

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

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

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

(e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)

(f) $\tau(S, A) \leftarrow 0$ *(Reset time since last tried)*

(g) **Loop repeat** n **times:**

- $S \leftarrow$ random previously observed state
- $A \leftarrow$ random action previously taken in S, or if never tried in S, any action in $A(S)$
- $R, S' \leftarrow Model(S, A)$
- $bonus \leftarrow \kappa \cdot \sqrt{\tau(S, A)}$ *(Exploration Bonus)*
- $Q(S, A) \leftarrow Q(S, A) + \alpha[R + bonus + \gamma \max_a Q(S', a) - Q(S, A)]$

(h) **For all** $s \in S, a \in A(s)$:

- $\tau(s, a) \leftarrow \tau(s, a) + 1$ *(Increment time since tried for all state-action pairs)*

- **(b):** See code and Figures 1 & 2.
- **(c):** From the two graphs, there was a mixed result on whether the footnote version of Dyna-Q+ performed better than without it. For the Blocking task, the footnote version did better, but for the shortcut maze task, the non-footnote version performed better. This may be attributed to the shortcut opening up a previously inaccessible area where the initial optimistic initialization of untried actions (as suggested in the footnote) in the newly accessible states may have provided misleading information, momentarily delaying the discovery of the shortcut.

3 Problem 3

UCB and Dyna-Q+ both utilize exploration bonuses but differ significantly in their application. UCB applies the bonus during action selection, influencing which action is chosen without affecting the underlying Q-value estimates.

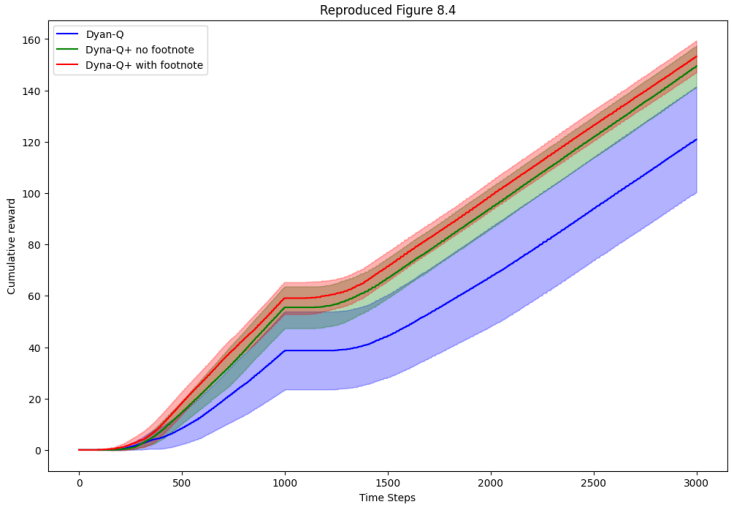


Figure 1: Problem 2(b): Reproduced Figure 8.4

This results in easier implementation and analysis, but exploration can be less persistent and sensitive to bonus scaling. On the other hand, Dyna-Q+ incorporates the bonus directly into simulated rewards, thereby altering Q-values and promoting directed exploration towards less-visited states. This approach excels in dynamic environments, driving persistent exploration and adapting to changes, but introduces complexity in implementation and analysis, potentially overestimating action values. Ultimately, UCB's extrinsic bonus suits stable environments, while Dyna-Q+'s intrinsic bonus, integrated into learning, benefits dynamic environments requiring model accuracy and adaptation.

4 Problem 4

- **(a): Tabular Dyna-Q for Stochastic Environments**

  Initialize:

  - $Q(s, a)$ arbitrarily for all $s \in S$, $a \in A(s)$
  - $Model(s, a)$ as an empty dictionary for all $s \in S$, $a \in A(s)$

  Loop forever:

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

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

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

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

  (e) **if** (S, A) not in $Model$:

- $Model(S, A) = \{\}$ // *Initialize dictionary for* (S, A) *if first visit*

(f) $Model(S, A)[S'] = Model(S, A).get(S', 0) + 1$ // *Increment count of* S' *given* S, A

(g) **Loop repeat** n times:

- $S \leftarrow$ random previously observed state
- $A \leftarrow$ random action previously taken in S
- **if** $Model(S, A)$: // *Ensure there is an entry for the state-action pair*
  - $S' \leftarrow$ random next state given distribution from $Model(S, A)$
  - $R \leftarrow$ average reward previously experienced for S, A, S'
  - $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

This modified tabular Dyna-Q algorithm accommodates stochastic environments by storing distributions of next states and rewards within the model, rather than single values. This is because stochastic environments have probabilistic transitions and rewards. The algorithm uses sample updates during planning, drawing from the stored distributions. This approach more accurately reflects learning in environments where the true distributions are unknown, making the algorithm more robust and broadly applicable to real-world scenarios. The chosen dictionary-based representation and averaging method for rewards offer reasonable efficiency while capturing the stochastic nature of the environment.

- **(b):** See code and Figure 3 & 4.

5 Problem 5

Iteration 4

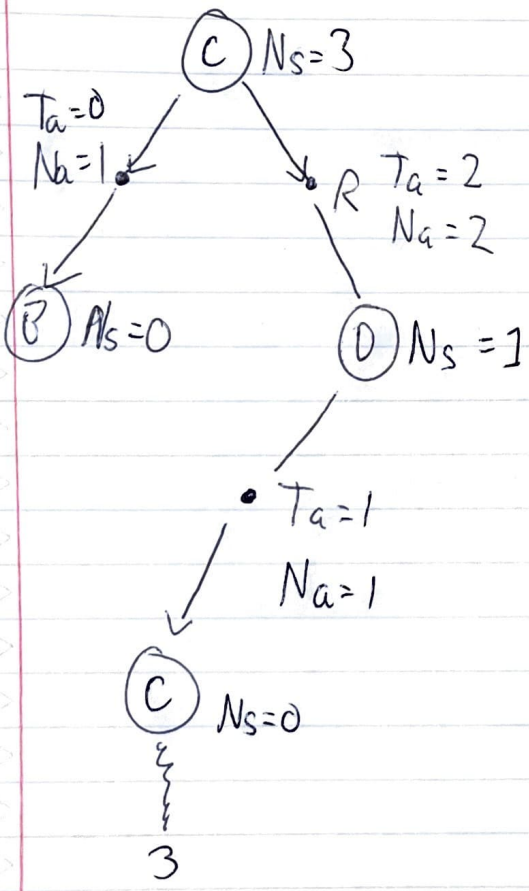
Selection: C → ~~D~~ → C

Expansion: C → B (using road list)

Simulation: B → A → B → C → D → E

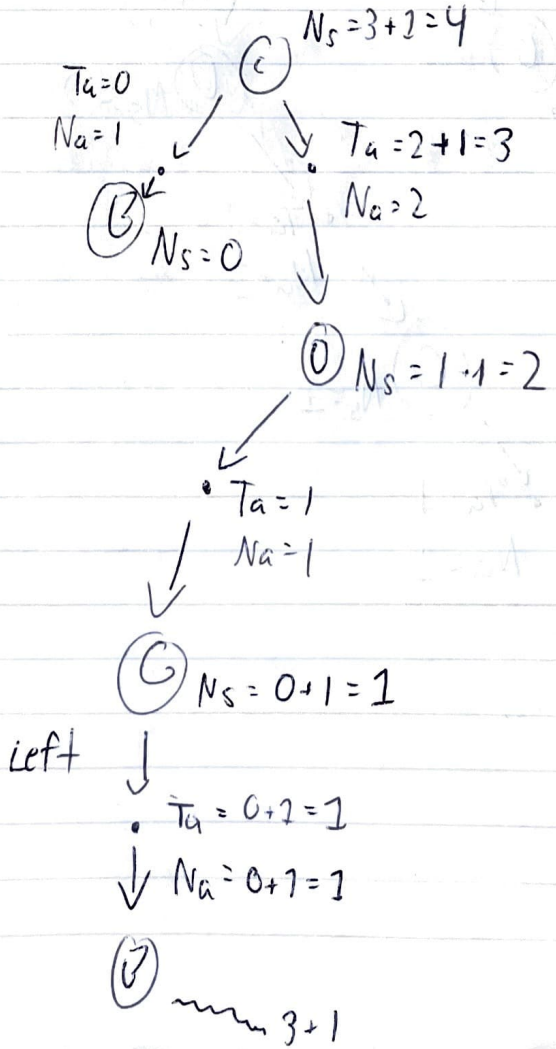
R = 1

Previous Tree:



Reward + 1

Backup:



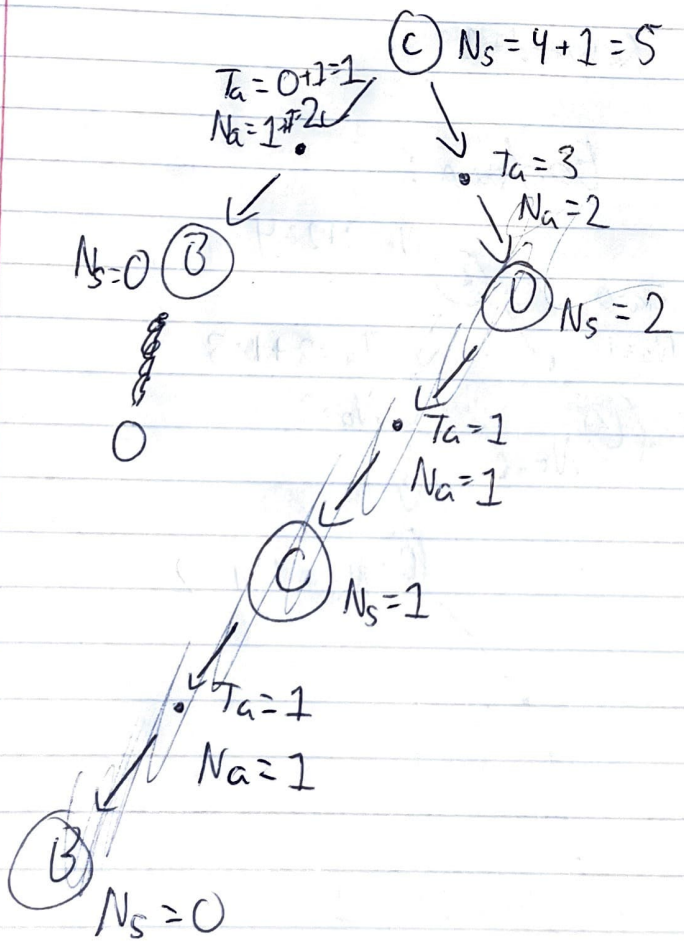
Iteration 5:

Selection: C → B

Expansion: None

Simulation: B → A R = 0

Backup:



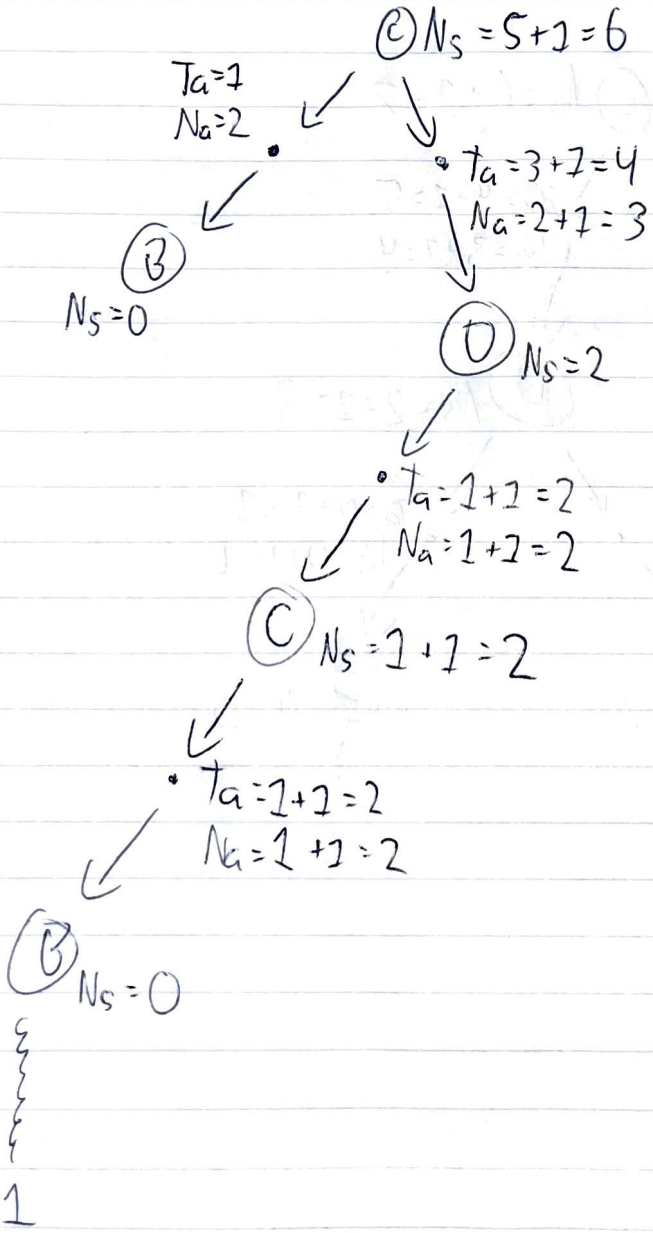
Iteration 6:

Selection: C → U → C → B

Expansion: None

Simulation: B → C → U → E R+1

Backup:



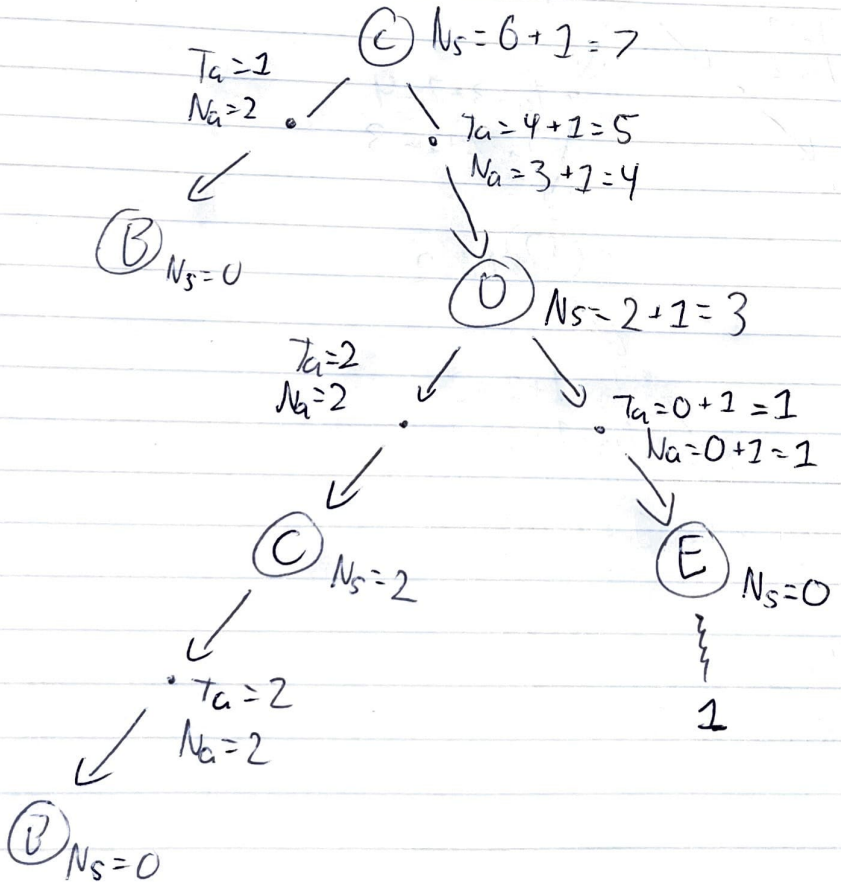
Iteration 7:

Selection: C → D

Expansion: D → E

Simulation: E + 1

Backup:



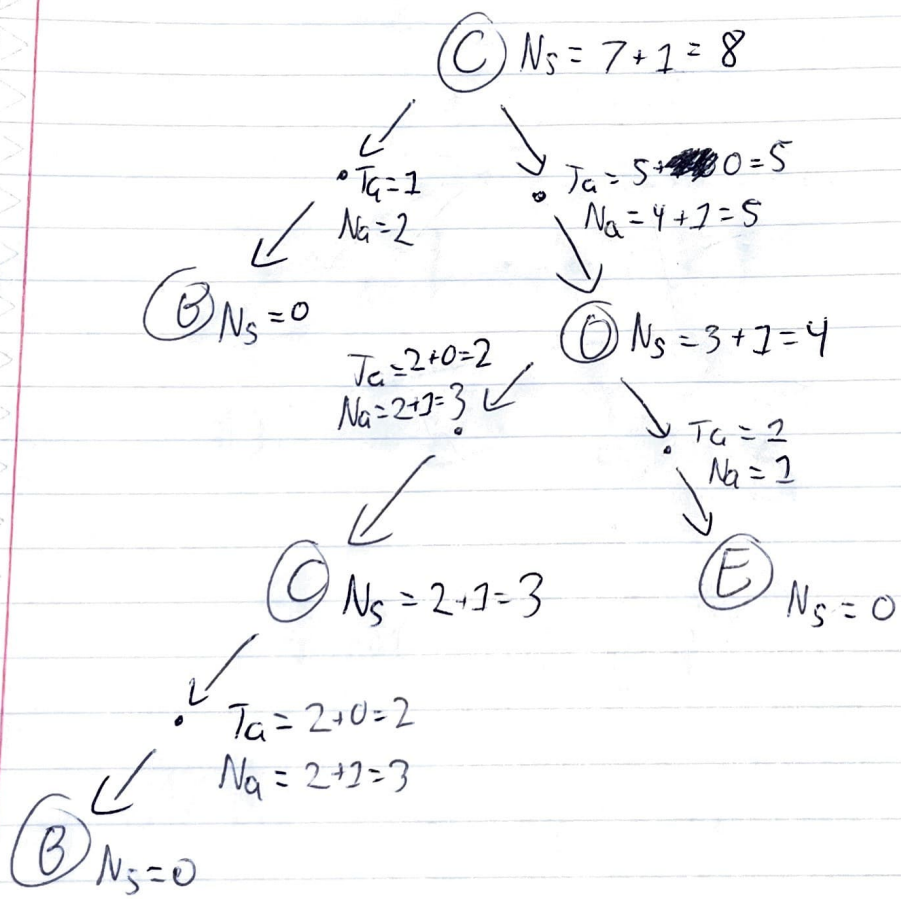
Iteration 8:

Selection: C → D → C → B

Expansion: None

Simulation: B → A → B → A Q=0

Backup:



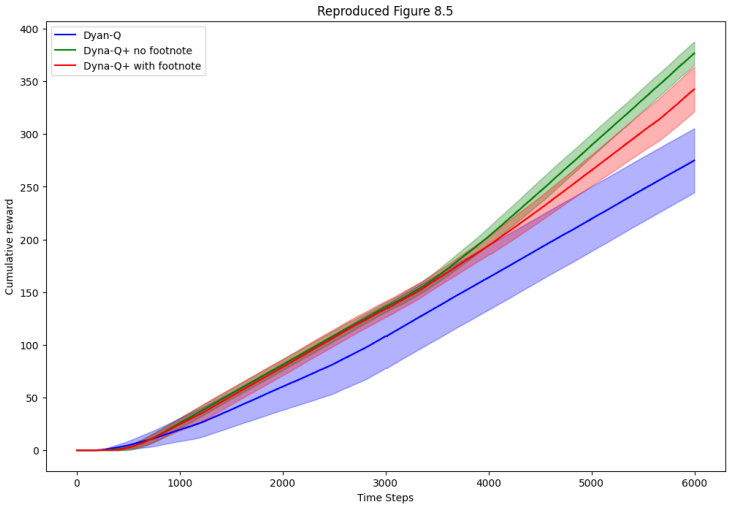


Figure 2: Problem 2(b): Reproduced Figure 8.5

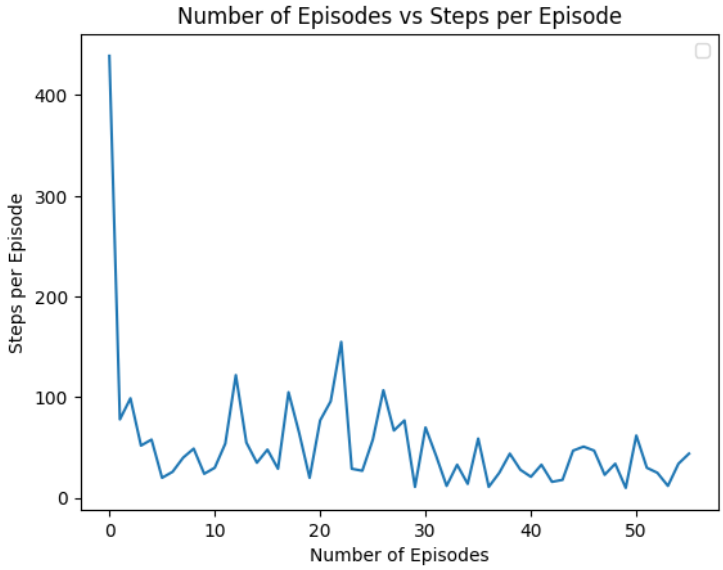


Figure 3: Problem 4(b): Numbers of Episodes vs Steps per Episode

```
Time step = 0, State = [3, 0], Action = down, Reward = -1, Next_state = [4, 0], Done = False
Time step = 1, State = [4, 0], Action = down, Reward = -1, Next_state = [5, 0], Done = False
Time step = 2, State = [5, 0], Action = right, Reward = -1, Next_state = [5, 1], Done = False
Time step = 3, State = [5, 1], Action = down, Reward = -1, Next_state = [6, 1], Done = False
Time step = 4, State = [6, 1], Action = right, Reward = -1, Next_state = [6, 2], Done = False
Time step = 5, State = [6, 2], Action = right, Reward = -1, Next_state = [6, 3], Done = False
Time step = 6, State = [6, 3], Action = right, Reward = -1, Next_state = [4, 4], Done = False
Time step = 7, State = [4, 4], Action = right, Reward = -1, Next_state = [4, 5], Done = False
Time step = 8, State = [4, 5], Action = down, Reward = -1, Next_state = [5, 5], Done = False
Time step = 9, State = [5, 5], Action = right, Reward = -1, Next_state = [4, 6], Done = False
Time step = 10, State = [4, 6], Action = right, Reward = 0, Next_state = [3, 7], Done = True
```

Figure 4: Problem 4(b): Optimal Path