Assessment 4

SIT 796

1. Introduction

This task explains Dynamic Programming (DP) and its applications in the domain of Reinforcement Learning (RL). The task is organised as:

- DP introduction
- Applications of DP in RL
- Exact Policy Iteration: A DP example for RL domain
- Conclusion

2. DP Introduction

Optimization problems that have **overlapping subproblems** and **optimal substructure** can be solved using the property that the entire problem can be solved using the solutions of the subproblems. This method of breaking a problem down into constituent subproblems and solving the problem by using the solution of the subproblem is known as Dynamic Programming. The approach can be classified into two categories: **bottom-up (tabulation)** and **top-down (memoization).**

An example scenario of solving the problem of generating the nth Fibonacci number explains these two dynamic programming methods. Fig 1 shows how the presented problem shows overlap; any Fibonacci number can be represented by the Fibonacci numbers that are already calculated.

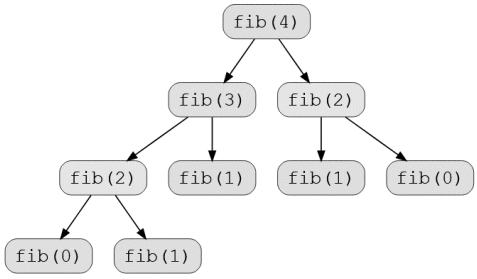

Fig 1: Overlapping substructure in Fibonacci Problem

Top down (memoization):

In the top-down approach, we return a solution as stored in the array, this represents **saving the solutions** to subproblems which can be used to find the nth Fibonacci number. This method is called memoization as the most recent state values are stored (memoized) and returned if it is encountered again.

if $N == 0$ *or* $N == 1$ **then**
  return N
end
if $Array[N]$ *!=* 0 **then**
  return $Array[N]$
end
$Array[N] \longleftarrow F(N-1, Array) + F(N-2, Array)$
return $Array[N]$

Bottom up (Tabulation):

In contrast with the top-down approach, the bottom-up approach does the reverse, it starts from the lowest number and stops when it reaches the nth Fibonacci. This method is called Tabulation because we create a DP table which stores the value of computed states and serves as cache to help calculate the value for next states.

```
if N == 0 or N == 1 then
    return N
end
A ⟵ 0
B ⟵ 1
for I ← 2 to N do
    Temp ⟵ A + B
    A ⟵ B
    B ⟵ Temp
end
return B
```

3. Application of DP in RL

[1] define Reinforcement Learning as a model of learning in which an agent learns to behave through trial-and-error interactions with a dynamic environment. The actions of agents tend to change the state of the environment from one state to the next. These changes in the state via agent actions are associated with transition probabilities. The relationship between a current state and its successor state is denoted by the Bellman equation (shows the optimal substructure):

$$V(s) = \max_a \left(R(s,a) + \gamma V\left(s'\right)\right)$$

Bellman Equation showing optimal substructure

Here,

- V(s) is the value of the state at current state *s*
- $\max_a$ denotes the maximum value for any possible action *a*
- R(s,a) denotes the expected reward for action *a* at state *s*
- YV(s') denotes the discount factor Y multiplied by the value of the next state *s'*

As we can see from the Bellman equation, the current state (*s)* can be represented in terms of the next state (s'). This representation of a state in terms of its successor states fulfils the overlapping subproblems property and allows using DP algorithms to solve Bellman Equation.

Here, we note that the current state is only dependent on the next state therefore we **do not need to *tabulate*** all the values for states, rather we just need to focus on the next state. This is where ***memoization*** helps optimize for space as we prevent storing all the values when using DP and just use the next state and for current state and so on. In this way, *memoization* can be used to reduce the recursion space of the solution when we solve Bellman Equation using DP by only saving values for the current state and next state instead of values of all states . This is especially helpful in RL scenarios as the state space of such problems is huge and *tabulating* these state spaces result in large storage overheads.

Two algorithms which use DP to solve Bellman Equation are:

- Policy Iteration (discussed in detail in next section)
- Value iteration (not discussed here)

These algorithms form the family of Synchronous DP, algorithms which act on selective states are known as Asynchronous DP methods (not discussed here).

4. Exact Policy Iteration: A DP example for RL domain

When we talk about *policy,* it refers to a mapping between one state to all actions possible for that state. The objective of policy iteration algorithm is to find such optimal policy which maximizes the reward. Policy iteration involves two blocks:

In Policy Evaluation, the total reward that an agent will achieve by following a particular policy is calculated. This step includes going through all the policies for all states and calculating the state value function, hence *evaluating* the policy. This can be done iteratively by going through all the states; hence we can utilize the *overlapping subproblems (states) for DP*. This is also known as Policy Prediction step. Also, as we iterate through the states this algorithm is also called iterative Policy Evaluation.

Policy improvement, best action from value function is calculated, if this action trumps the current policy action, then the policy is replaced (greedy), hence the name policy improvement. This step is also known as the Policy Control step. The updated greedy policy is given by:

$$\pi'(s) = \underset{a}{\operatorname{argmax}}\, q_\pi(s,a)$$

$$\pi'(s) = \underset{a}{\operatorname{argmax}} \sum_{s'r} p(s',r|s,a)\,[r + \gamma v_\pi(s')]$$

- $argmaxa$ denotes the value of a when the expression is maximised
- **qπ(s,a)** denotes the quality of selecting the action a in state s. Expanded using Bellman Equation.
- **π'(s)** denotes the new greedy policy based on **qπ(s,a).**

This greedy selection process is known as the policy improvement step, where the current policy is updated in a greedy way (first better policy chosen to replace current policy).

The general principle of Dynamic Programming of breaking the problems into subproblems and using overlapping substructures is utilized during Policy Iteration (pseudocode in Fig 2)

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s)) \left[r + \gamma V(s')\right]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   $policy\text{-}stable \leftarrow true$
   For each $s \in \mathcal{S}$:
      $a \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma V(s')\right]$
      If $a \neq \pi(s)$, then $policy\text{-}stable \leftarrow false$
   If $policy\text{-}stable$, then stop and return V and π; else go to 2

Pseudocode for policy iteration [2]

The above pseudocode has three steps:

1: Initialization: In this step, policy is initialized randomly, i.e., for all states the value function is set arbitrarily.
2: Policy Evaluation: In this step, policy is evaluated by state value function V(s).
3. Policy Improvement: As discussed earlier, we look ahead and choose the best policy in a greedy way. This means that the first policy which yields values better than the current policy is chosen as the new policy.

These steps are repeated until a stable policy is obtained. This is decided when we get **a == π(s**) i.e.: a stable policy. This algorithm is known as **Policy Iteration**.

The repeated evaluation and improvement of policy has underlying overlapping pattern to it, which is shown by Bellman Equation earlier, by showing how consecutive states can be represented as the function of previous states. This is where DP is used to prevent re-computation of state values and *memoization* is used to prevent storing all the states.

5. Conclusion

This paper introduces DP and explains its two approaches. The importance of *memoization* is also shown when used in RL scenarios to reduce storage overhead related to huge state spaces. Finally, an example algorithm: Policy Iteration is shown to use DP to solve RL problems.

6. References

[1] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "An introduction to reinforcement learning. The Biology and Technology of Intelligent Autonomous Agents," pp. 90–127, 1995.

[2] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," IEEE Trans. Neural Netw., vol. 9, no. 5, pp. 1054–1054, 1998.