Original Article

Dynamic programming for solving a simulated clinical scenario of sepsis resuscitation

Zhongheng Zhang[1,2#], Xiaodian Zhang[2#], Shenhong Gu[2#], Xiaoqing Xu[2], Wei Jiang[2], Chuanzhu Lv[2], Shaojiang Zheng[2]

[1]Department of Emergency Medicine, Sir Run-Run Shaw Hospital, Zhejiang University School of Medicine, Hangzhou, China; [2]Key Laboratory of Emergency and Trauma of Ministry of Education, Hainan Provincial Key Laboratory for Tropical Cardiovascular Diseases Research, The First Affiliated Hospital of Hainan Medical University, Research Unit of Island Emergency Medicine of Chinese Academy of Medical Sciences, Hainan Medical University, Haikou, China

Contributions: (I) Conception and design: Z Zhang, S Zheng; (II) Administrative support: C Lv, S Zheng; (III) Provision of study materials or patients: C Lv, X Zhang, S Gu, X Xu; (IV) Collection and assembly of data: Z Zhang, S Gu, X Xu; (V) Data analysis and interpretation: Z Zhang, X Zhang, W Jiang, S Zheng; (VI) Manuscript writing: All authors; (VII) Final approval of manuscript: All authors.

[#]These authors contributed equally to this work.

Correspondence to: Shaojiang Zheng. The First Affiliated Hospital of Hainan Medical University, Research Unit of Island Emergency Medicine of Chinese Academy of Medical Sciences, Hainan Medical University, Haikou 571199, China. Email: zhengsj2008@163.com; Chuanzhu Lv. The First Affiliated Hospital of Hainan Medical University, Research Unit of Island Emergency Medicine of Chinese Academy of Medical Sciences, Hainan Medical University, Haikou 571199, China. Email: lyuchuanzhu@hainmc.edu.cn; Zhongheng Zhang. 3#, East Qingchun Road, Hangzhou 310016, China. Email: zh_zhang1984@zju.edu.cn.

Background: A major challenge in clinical research is population heterogeneity and we need to consider both historical response and current condition of an individual in considering medical decision making. The idea of precise medicine cannot be fully accounted for in traditional randomized controlled trials. Reinforcement learning (RL) is developing rapidly and has found its way into various fields including clinical medicine in which RL is employed to find an optimal treatment strategy. The key idea of RL is to optimize the treatment policy depending on the current state and previous treatment history, which is consistent with the idea behind dynamic programming (DP). DP is a prototype of RL and can be implemented when the system dynamics can be fully quantified.

Methods: The present article aims to illustrate how to perform DP algorithm in a clinical scenario of Sepsis resuscitation. The state transition dynamics are constructed in the framework of Markov Decision Process. The state space is defined by mean arterial pressure (MAP) and lactate; the action space is comprised of fluid administration and vasopressor. The implementation of policy evaluation, policy improvement and iteration are explained with R code.

Results: the DP algorithm was able to find the optimal treatment policy depending on the current states and previous conditions. The iteration process converged at finite steps. We defined several functions such as nextStep(), policyEval() and policy_iteration() to implement the DP algorithm.

Conclusions: This article illustrates how DP can be used to solve a clinical problem. We show that DP is a potential useful tool to tailor treatment strategy to patients with different conditions/states. Potential audience of the paper are those who are interested in using DP for solving clinical problems with dynamic changing states.

Keywords: Dynamic programming (DP); sepsis; Markov decision process; reinforcement learning (RL)

Submitted Oct 22, 2020. Accepted for publication Dec 31, 2020.
doi: 10.21037/apm-20-2084
View this article at: http://dx.doi.org/10.21037/apm-20-2084

Introduction

Dynamic programming (DP) is both a mathematical optimization method and a computer programming method that solves problems by combining the solutions to subproblems (1). The Wikipedia definition states that "*[DP is a] method for solving complex problems by breaking them down into simpler subproblems*". A dynamic-programming algorithm solves each subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subprogram. The idea of DP is to search a good policy (i.e., the policy to maximize the long-term reward) by using the value function. DP is closely related to the idea of reinforcement learning (RL) but builds on a known Markov Decision Process (MDP). The MDP comprises state, action and reward sets, denoted by $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$, respectively. These terms can be explained in the context of clinical medicine. The state is constructed by features such as blood pressure, body temperature and laboratory tests in clinical medicine. The action refers to any treatment that the physician can prescribe. Reward refers is a quantity that is defined to quantify the goodness of a state transition. If the state is improving, e.g., transitioning from poor state to good state, a positive reward is usually assigned. The transition probability from one state to another under action a is denoted as $p(s',r|s,a)$, where $s\in\mathcal{S}$, $a\in\mathcal{A}(s)$, $r\in\mathcal{R}$ and $s'\in\mathcal{S}^+$. s' is an instance of a successive state of s, with its value updated at the last time step.

The article explains key concepts of DP in a clinical scenario and illustrates how DP can be used to solve clinical questions. The clinical question in this article is how to select an appropriate intervention to rescue patients with septic shock (2-4). Typically, a patient with septic shock should be rapidly managed with sepsis bundles in the emergency room. This bundle includes early antibiotic administration, fluid resuscitation, balance of oxygen supply and demand, and monitoring of urine output and serum lactate (5-7). Many states can be defined for septic shock and here we define the state space with mean arterial pressure (MAP) and lactate. The action space is comprised of vasopressor and fluid administration. The aim of the DP algorithm is to find an optimal policy at any states. In other words, the optimal treatment strategy can be quite different for patients with different states (e.g., high MAP and low lactate versus those with low MAP and low lactate). The adoption of optimal treatment strategy means the patient can get the best clinical outcome given his/her own current states. The idea of DP is well described in the RL bible (8), and we just illustrate this algorithm in a clinical scenario with R code (*Figure 1*).

Methods

Constructing a state space

The background knowledge for the clinical management of sepsis is explained in this section. Since the study did not involve real human subjects, ethical approval was waived for the study. Sepsis is an organ dysfunction syndrome caused by uncontrolled inflammation in response to infection. Septic shock is a severe form of sepsis and is potentially life-threatening, which requires rapid response and initiation of resuscitation bundle. The key to successful treatment is to increase arterial blood pressure and restore effective circulatory volume (9). Physicians typically use vasopressor and fluid to rescue shock status (hypotension and hypoperfusion). Fluid can help to restore total blood volume to raise the blood pressure, and vasopressors can increase vascular tone thereby increasing the arterial blood pressure. Blood pressure is important for the maintenance of tissue perfusion. Here we construct a state space comprising MAP and lactate, with different combinations of the two variables relating to different pathological states. The action space is comprised of treatment options (i.e., fluid and vasopressor) that can be applied to a given patient.

To simplify the problem, the state space is constructed with a two dimensional 4×4 grid, with two feature variables of lactate and MAP. In other words, it is assumed that a patient's condition can be fully captured with these two variables. The termination state is defined as *Died* and *Alive* at hospital discharge; note that patients with extremely low MAP and high lactate are more likely to die. The terminal states are irreversible (*Figure 2*). In real world setting, the state space can be constructed with higher dimensional features, but the idea to solve the problem is the same.

```
> gridSize = 4
> terminationStatesDie <- matrix(c(rep(1, 4),
+   2:4, 1:4, rep(4, 3)), nrow = 7)
> terminationStatesAlive <- c(4, 1)
> actions = c("vaso", "fluid", "both", "none")
> library(ggplot2)
> ggplot() + annotate("text", label = paste("Died",
+   "\n", apply(terminationStatesDie, MARGIN = 1,
```

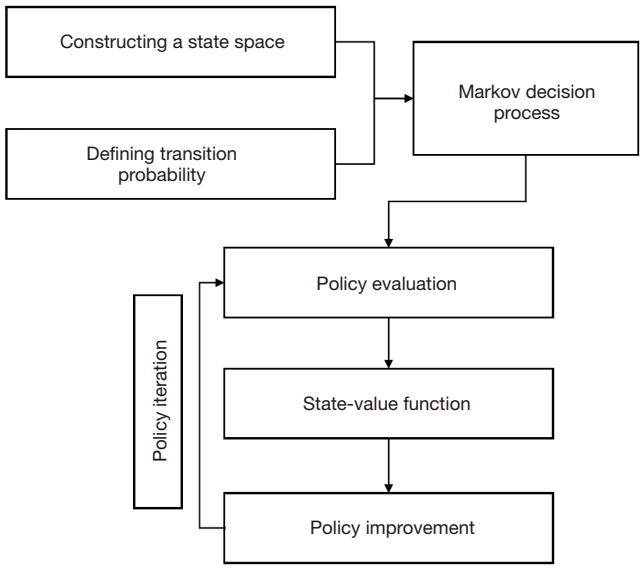

Figure 1 Workflow of the dynamic programming.

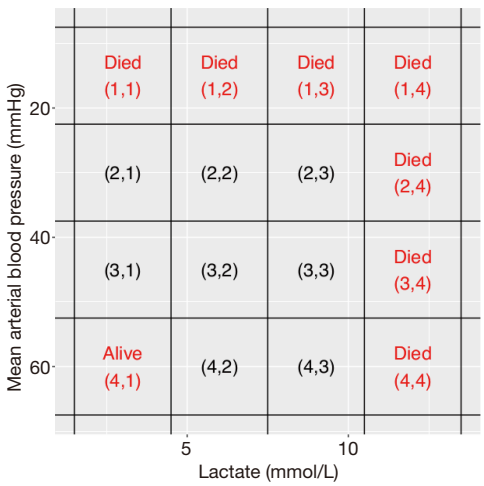

Figure 2 Schematic illustration of the state space with a 4×4 grid. The red annotated cells represent the terminal states. The agent cannot leave the terminal states. The state space is defined by mean arterial blood pressure (y axis) and lactate (x axis). There are nine possible next states s' for a given non-terminal state s.

```
+      FUN = function(xx) {
+        paste("(", xx[1], ",", xx[2],
+              ")", sep = "")
+      }), sep = ""), x = terminationStatesDie[,
+    2] * 3, y = terminationStatesDie[, 1] *
+      15, size = 6, colour = "red") + annotate("text",
+    label = c("Alive\n(4,1)"), x = terminationStatesAlive[2] *
+       3, y = terminationStatesAlive[1] *
+       15, size = 6, colour = "red") + annotate("text",
+    label = c(paste("(2,", 1:3, ")", sep = ""),
+       paste("(3,", 1:3, ")", sep = ""),
+       "(4,2)", "(4,3)"), x = rep(1:3, 3)[-7] *
+       3, y = rep(2:4, each = 3)[-9] * 15,
+     size = 6, colour = "black") + geom_hline(yintercept =
c(0.5,
+    1.5, 2.5, 3.5, 4.5) * 15) + geom_vline(xintercept = c(0.5,
+    1.5, 2.5, 3.5, 4.5) * 3) + xlim(0.5 *
+    3, 4.5 * 3) + ylim(0.5 * 15, 4.5 * 15) +
+    labs(y = "Mean Arterial Blood Pressure (mmHg)",
+         x = "Lactate (mmol/L)") + scale_y_continuous(trans =
"reverse")
## Scale for 'y' is already present. Adding another scale for 'y',
which will
## replace the existing scale.
```

Transition probability

The finite MDP dictates that all components of the set S,A,R have the finite number of elements. The random variable R_t and S_t have well defined probability distributions conditional on previous states S_{t-1} and actions A_{t-1}. For an instance of $s' \in S$ and $r \in R$, the probability of the instance can be written as (8):

$$p(s', r \mid s, a) = Pr(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a) \quad [1]$$

In our example, there is nine possible next positions given a current position. The probability of each transition given a particular action a is defined in the following list of transition matrices. Specifically, the TransPro object is a list with each of the component referring to a specific action, namely, vaso, fluid, both or none. Each component is a matrix of transition probability that defines the state transition probability under each action. The requirement of a known transition matrix for each action is also a limitation of the DP algorithm because the transition matrices are typically not known in reality.

```
> # non-deterministic action
```

```
> vasoX <- c(0.1, 0.7, 0.2)
> vasoY <- c(0.1, 0.8, 0.1)
> Mvaso <- sapply(vasoY, function(xx) vasoX *
+   xx)
> fluidX <- c(0.1, 0.6, 0.3)
> fluidY <- c(0.5, 0.2, 0.3)
> Mfluid <- sapply(fluidY, function(xx) fluidX *
+   xx)
> bothX <- c(0.05, 0.8, 0.15)
> bothY <- c(0.3, 0.1, 0.6)
> Mboth <- sapply(bothY, function(xx) bothX *
+   xx)
> noneX <- c(0.6, 0.2, 0.2)
> noneY <- c(0.1, 0.6, 0.3)
> Mnone <- sapply(noneY, function(xx) noneX *
+   xx)
> TransPro <- list(vaso = Mvaso, fluid = Mfluid,
+   both = Mboth, none = Mnone)
> TransPro$fluid
##      [,1] [,2] [,3]
## [1,] 0.05 0.02 0.03
## [2,] 0.30 0.12 0.18
## [3,] 0.15 0.06 0.09
```

The above output shows the transition probability under action *fluid* (i.e., we give fluid to a patient). The probability of transition from current state to the left cell (e.g., from (2,2) to (2,1) is 0.30; and the probability of staying in current position is 0.12. Note that the sum of all cells equals to 1. The action is called non-deterministic action because when one takes an action, the next position is a stochastic distribution of all possible positions.

```
> nextStep <- function(currentPosition, action,
+   moveInd) {
+   if (!action %in% actions) {
+     stop("Choose action value from vaso,fluid,both and
none!")
+   }
+   if (sum(apply(terminationStatesDie, 1,
+     function(xx) identical(currentPosition,
+       xx)))) {
+     nextPosition = currentPosition
+     reward = -1
+     transProb <- 1/9
+   } else if (identical(currentPosition, terminationStatesAlive))
{
+     nextPosition = currentPosition
+     reward = 1
+     transProb <- 1/9
+   } else {
+     nextPosition <- currentPosition +
+       moveInd
+     reward = 0
+     transProb <- TransPro[[action]][which(c(-1,
+       1, 0) %in% moveInd[1]), which(c(-1,
+       1, 0) %in% moveInd[2])]
+   }
+   if (nextPosition[2] < 1 | nextPosition[1] >
+     gridSize) {
+     nextPosition = currentPosition
+   }
+   return(list(nextPosition = nextPosition,
+     reward = reward, transProb = transProb))
+ }
```

The above code defines the *nextStep()* function, which receives current position, action and move index as input; and output the next position, reward value and transition probability. The move index is a two elements numeric vector which defines the direction of movement for the next step. When the current position is the terminal state *Died*, it will stay in the current position with a reward of −1. Because there are nine possible move index, we assign the transition probability of 1/9 so that the sum of all transition probability for a given terminal state is 1 (i.e., we will loop over all possible movement index in the following functions). When the current position is the terminal state *Alive*, the next position remains unchanged and the reward is 1. Otherwise, the agent moves from current position to the next position by

a given move index. The reward for all non-terminal states is 0. If the next position moves out of the grid, it returns to the current position (position unchanged). Now let's experiment with the *nextStep()* function.

```
> nextStep(c(2, 2), "fluid", moveInd = c(-1,
+   -1))
## $nextPosition
## [1] 1 1
##
## $reward
## [1] 0
##
## $transProb
## [1] 0.05
> nextStep(c(2, 1), "fluid", moveInd = c(-1,
+   -1))
## $nextPosition
## [1] 2 1
##
## $reward
## [1] 0
##
## $transProb
## [1] 0.05
```

The above code shows two instances of movement in the grid. When the current state is (2,2) and the move index is $(left(-1), upper(-1))$, it moves to the position (1,1) with the reward of 0 and the transition probability is 0.05. Also note that a reward is added when the agent moves away from the corresponding cell. Thus, when moving away from (2,2) to (1,1), the agent receives reward 0 rather than −1. In contrast, when the current position is (2,1), the $(left, upper)$ movement will move out of the left limit and the next position remains unchanged.

Statistical analysis (policy evaluation)

Policy evaluation refers to the calculation of state-value function v_π under a given policy π. This is referred to as the prediction problem. If the dynamic system of MDP is well known, the state-value function can be written as (8):

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\left[G_t \mid S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s\right] \\ &= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right] \end{aligned} \quad [2]$$

where G_t is the long term reward from time t, also known as the *return*. The expected return under policy π for state s is the state-value function. Also note that the returns at successive time steps (G_t,G_{t+1},G_{t+2}) are related to each other by a reward R and discounting factor γ. s' is the next posion of s. The equation indicates that the current state-value (k step) is updated upon the last state-value (k−1 step) in the iterative process. A value table can be used to store state values. This table maps each state to real numbers. For instance, the state (2,1) can take a value 10, and the terminal state (1,1) takes 0. The value table is updated iteratively by using values obtained in the last step. Thus, we need two value tables, one is used to store current state values and the other is used to store previous state values. The update rule for state s can be written as (8):

$$\begin{aligned} v_{k+1}(s) &= \mathbb{E}_\pi\left[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s\right] \\ &= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_k(s')\right] \end{aligned} \quad [3]$$

the subscript t refers to the time step. For example, if the current position at time t is S_t=s(2,2), the next position at time t+1 can be S_{t+1}=s(1,1) by a move index of (−1,−1). The subscript k refers to the iterative step that each one will update the state-values for all states once. If the MDP dynamics are known, the Expectation $\mathbb{E}_\pi$ can be obtained by summing over all possible next states S_{t+1}. s' is an instance of S_{t+1} for the state s. The total number of time step T can be very large (e.g., 5,000 in our default setting). The state-value is guaranteed to converge if γ<1 in finite number of iterations.

```
> policyEval <- function(numIterations = 5000,
+   gamma = 0.95, valueMap) {
+   for (ii in 1:numIterations) {
+     delta = 0
+     for (jj in 1:(gridSize * gridSize)) {
+       weightedRewards = 0
+       state = as.numeric(states[jj,
+         ])
```



```
+       for (action in actions) {
+         for (mm in 1:(3 * 3)) {
+           moveInd <- moveIndMatrix[mm,
+             ]
+           NextStepReturn <- nextStep(state,
+             action, moveInd)
+           nextPosition <- NextStepReturn[[1]]
+           reward <- NextStepReturn[[2]]
+           transPro <- NextStepReturn[[3]]
+           weightedRewards = weightedRewards +
+             0.25 * transPro * (reward +
+               gamma * valueMap[nextPosition[1],
+                 nextPosition[2]])
+         }
+       }
+       valueMap1[state[1], state[2]] = weightedRewards
+       delta = max(delta, abs(weightedRewards -
+         valueMap[state[1], state[2]]))
+     }
+     valueMap = valueMap1
+     if (delta < 0.01) {
+       break
+     }
+   }
+   return(valueMap)
+ }
```

The above code defines a policy evaluation function for a random policy, which takes a value map as input and output an updated value map. The value map is a 4×4 grid that each cell represents a value for that corresponding state. Now let's execute the code with random policy. The treatment option is selected at random at any state *S*, thus $\pi(a|s)=0.25$ for any state *s* (i.e., there are four treatment options in the example).

```
> valueMap <- matrix(rep(0, gridSize * gridSize),
+   nrow = gridSize)
> valueMap1 <- matrix(rep(0, gridSize * gridSize),
+   nrow = gridSize)
> states <- as.matrix(expand.grid(1:gridSize,
+   1:gridSize))
> moveIndMatrix <- as.matrix(expand.grid(c(-1,
+   1, 0), c(-1, 1, 0)))
> policyEval(valueMap = valueMap)
##          [,1]     [,2]     [,3]     [,4]
## [1,] -19.812121 -19.812121 -19.81212 -19.81212
## [2,]  -6.755552  -9.465815 -14.30679 -19.81212
## [3,]   1.513571  -4.112980 -13.19934 -19.81212
## [4,]  19.812121  -4.130786 -12.74781 -19.81212
```

The output of the *policyEval()* function is the state-value for the random policy. Note that each state is assigned a value. This value is the long-term expected return (sum of reward) of starting from the corresponding state. The state-value has the lowest value at the terminal state of *Died*, and the highest value at the terminal state of *Alive*. The states approaching the *Alive* cell have greater values than those near the *Died* cells. From this state-value table, the optimal treatment at each state can be identified to maximize the long-term return. However, the state-value table will change if the policy changed.

State-value function for a specified policy

The above function returns a matrix of state-values corresponding to a random policy. It is also interesting to estimate state-value for a given policy. A given policy specifies a treatment option for each of the states.

```
> policy_evaluate <- function(states, PolicyInd,
+   gamma = 0.95, valueMap, numIterations = 5000) {
+   for (ii in 1:numIterations) {
+     delta = 0
+     for (jj in 1:(gridSize * gridSize)) {
+       weightedRewards = 0
+       state = as.numeric(states[jj,
+         ])
+       if (length(unique(as.vector(PolicyInd))) ==
+         1) {
+         for (action in actions) {
+           for (mm in 1:(3 * 3)) {
```

```
+             moveInd <- moveIndMatrix[mm,
+              ]
+             NextStepReturn <- nextStep(state,
+              action, moveInd)
+             nextPosition <- NextStepReturn[[1]]
+             reward <- NextStepReturn[[2]]
+             transPro <- NextStepReturn[[3]]
+             weightedRewards = weightedRewards +
+              0.25 * transPro * (reward +
+               gamma * valueMap[nextPosition[1],
+                nextPosition[2]])
+           }
+         }
+       } else {
+         # use the current greedy policy
+         action = PolicyInd[state[1],
+          state[2]]
+         for (mm in 1:(3 * 3)) {
+          moveInd <- moveIndMatrix[mm,
+           ]
+          NextStepReturn <- nextStep(state,
+           action, moveInd)
+          nextPosition = NextStepReturn[[1]]
+          reward = NextStepReturn[[2]]
+          transPro <- NextStepReturn[[3]]
+          weightedRewards = weightedRewards +
+           transPro * (reward +
+            gamma * valueMap[nextPosition[1],
+             nextPosition[2]])
+         }
+       }
+       valueMap1[state[1], state[2]] = weightedRewards
+       delta = max(delta, abs(weightedRewards -
+         valueMap[state[1], state[2]]))
+     }
+     valueMap <- valueMap1
+     if (delta < 0.01) {
+       break
+     }
+   }
+   return(valueMap)
+   print(valueMap)
+ }
```

For example, we can define an arbitrary policy in a 4×4 grid and calculates the state-value function.

```
> PolicyInd = matrix(rep("none", gridSize *
+   gridSize), nrow = gridSize)
> PolicyInd[c(2, 3), c(1, 4)] <- "vaso"
> PolicyInd[c(2, 3), c(2, 3)] <- "fluid"
> PolicyInd
##      [,1]   [,2]   [,3]   [,4]
## [1,] "none" "none"  "none"  "none"
## [2,] "vaso" "fluid" "fluid" "vaso"
## [3,] "vaso" "fluid" "fluid" "vaso"
## [4,] "none" "none"  "none"  "none"
> policy_evaluate(states, PolicyInd, valueMap = valueMap)
##          [,1]      [,2]      [,3]      [,4]
## [1,] -19.812121 -19.812121 -19.812121 -19.81212
## [2,]  -1.123072  -3.634555  -7.077458 -19.81212
## [3,]  -1.311871   2.514287  -8.215748 -19.81212
## [4,]  19.812121  -4.842859 -13.785760 -19.81212
```

The state-values of the current policy is different from that of the random policy. But the values at terminal states are all the same because actions have no effect on terminal states.

Results

Policy improvement

The purpose of computing the state-value function is to find a better policy. The state-action value function is used to address this problem. This value can be thought of selecting a in s and following policy π thereafter.

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi\left[R_{t+1} + \gamma v_k\left(S_{t+1}\right) \mid S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s,r \mid s,a)\left[r + \gamma v_k(s)\right] \end{aligned} \quad [4]$$

If $q_\pi(s,a) \geq v_\pi(s)$, it will be better to select a in s than follow the policy π all the time. The policy improvement theorem is described as follows in Eq. [5]. Suppose there are two deterministic policies π_1 and π_2 such that for all $s \in S$, $q_{\pi_1}(s,\pi_2(s)) \geq v_{\pi_1}(s)$, then π_2 is better than or equal to π_1. The greedy policy can be the policy to maximize the state-action value at each iterative step.

$$\begin{aligned}\pi_{greedy}(s) &\doteq \underset{a}{\operatorname{argmax}}\, q_\pi(s,a) \\ &= \underset{a}{\operatorname{argmax}}\, \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a\right] \\ &= \underset{a}{\operatorname{argmax}} \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma v_\pi(s')\right]\end{aligned} \quad [5]$$

The greedy policy takes the action that maximize the q value at short time, but it is not necessarily the best action in the long run. The greedy policy for each state under a given state-value table can be implemented with the following *greedify_policy*() function. Note that this function will not change the state-value table. It returns an updated policy for one state. Then the *improve_policy*() function improves policy over all states, except for the terminal states.

```
> # Compute the best action in each state
> greedify_policy <- function(state, PolicyInd,
+   gamma = 0.95, valueMap) {
+   q_values <- rep(0, length(actions))
+   idx = 1
+   for (action in actions) {
+     for (mm in 1:(3 * 3)) {
+       moveInd <- moveIndMatrix[mm,
+         ]
+       NextStepReturn <- nextStep(state,
+         action, moveInd)
+       nextPosition = NextStepReturn[[1]]
+       reward = NextStepReturn[[2]]
+       transPro <- NextStepReturn[[3]]
+       q_values[idx] = q_values[idx] +
+         transPro * (reward + gamma *
+           valueMap[nextPosition[1],
+             nextPosition][2])
+     }
+     idx = idx + 1
+   }
+   # Find the index of the action for which
+   # the q_value is
+   indmax = which(q_values == max(q_values))
+   PolicyInd[state[1], state[2]] = actions[indmax]
+   return(PolicyInd)
+ }
>
> improve_policy <- function(PolicyInd, valueMap,
+   gamma = 0.95) {
+   policy_stable = TRUE
+   for (jj in c(2, 3, 6:8, 10:12)) {
+     state = as.numeric(states[jj, ])
+     old <- PolicyInd[state[1], state[2]]
+     # Greedify policy for state
+     PolicyInd <- greedify_policy(state,
+       PolicyInd, gamma, valueMap)
+     if (!identical(PolicyInd[state[1],
+       state[2]], old)) {
+       policy_stable = FALSE
+     }
+   }
+   return(list(PolicyInd, policy_stable))
+ }
```

Let's see an example to better understand the above two functions.

```
> # The current policy
> PolicyInd
##      [,1]   [,2]   [,3]   [,4]
## [1,] "none" "none"  "none"  "none"
## [2,] "vaso" "fluid" "fluid" "vaso"
## [3,] "vaso" "fluid" "fluid" "vaso"
## [4,] "none" "none"  "none"  "none"
> valueMapExample <- policy_evaluate(states,
+   PolicyInd, gamma = 0.95, valueMap, numIterations = 5000)
> valueMapExample
```

```
##          [,1]     [,2]     [,3]     [,4]
## [1,] -19.812121 -19.812121 -19.812121 -19.81212
## [2,]  -1.123072  -3.634555  -7.077458 -19.81212
## [3,]  -1.311871   2.514287  -8.215748 -19.81212
## [4,]  19.812121  -4.842859 -13.785760 -19.81212
> improve_policy(PolicyInd, valueMap = valueMapExample)
## [[1]]
##      [,1]   [,2]   [,3]   [,4]
## [1,] "none" "none"  "none"  "none"
## [2,] "vaso" "both"  "both"  "vaso"
## [3,] "both" "fluid" "fluid" "vaso"
## [4,] "none" "fluid" "fluid" "none"
##
## [[2]]
## [1] FALSE
```

The results show that the updated greedy policy is different from the initial policy. For example, the initial policy adopts "fluid" intervention in state (2,2), while the update policy adopts "both" intervention in this state.

Policy iteration

The purpose of policy iteration is to iteratively improve the policy π given the constantly updating state-value tables v_π. The process of the policy iteration can be written as:

$$\pi_0 \overset{E}{\rightarrow} v_{\pi_0} \overset{I}{\rightarrow} \pi_1 \overset{E}{\rightarrow} v_{\pi_1} \overset{I}{\rightarrow} \pi_2 \overset{E}{\rightarrow} \cdots \overset{I}{\rightarrow} \pi_* \overset{E}{\rightarrow} v_* \quad [6]$$

The policy evaluation and improvement intersect with each other and the algorithm can finally achieve a converged optimal policy π_* and value function v_*. The policy iteration process can be implemented with the following code. Except for the first policy evaluation which starts from an initial value map with all 0 input, each subsequent policy evaluation is started from the value map of the previous policy.

```
> policy_iteration <- function(gamma = 0.95,
+     theta) {
+     valueMap = matrix(rep(0, gridSize * gridSize),
+         nrow = gridSize)
+     PolicyInd = matrix(rep("none", gridSize *
+         gridSize), nrow = gridSize)
+     policy_stable = FALSE
+     while (!policy_stable) {
+         valueMap = policy_evaluate(states,
+             PolicyInd, valueMap = valueMap)
+         ImpPolicy <- improve_policy(PolicyInd,
+             valueMap)
+         PolicyInd = ImpPolicy[[1]]
+         policy_stable = ImpPolicy[[2]]
+     }
+     return(list(valueMap, PolicyInd))
+ }
> theta = 0.1
> policyIter <- policy_iteration(theta)
> policyIter
## [[1]]
##          [,1]     [,2]     [,3]     [,4]
## [1,] -19.97184 -19.97184 -19.971839 -19.97184
## [2,]  12.94852  10.40723   2.198744 -19.97184
## [3,]  16.89435  12.71012   3.121192 -19.97184
## [4,]  19.97184  12.24205   2.654893 -19.97184
##
## [[2]]
##      [,1]   [,2]   [,3]   [,4]
## [1,] "none" "none"  "none"  "none"
## [2,] "both" "both"  "both"  "none"
## [3,] "both" "fluid" "both"  "none"
## [4,] "none" "fluid" "fluid" "none"
```

The above output shows the optimal value function v_* and optimal policy π_*. At the terminal states, the initial "none" treatment is not changed during policy iteration. The optimal treatment strategy such as "both" and "fluid" are employed in other non-terminal states.

Discussion

This article illustrates how DP can be used to solve a clinical problem. We show that DP is a potential useful tool to tailor treatment strategy to patients with different

conditions/states. The key concepts of DP include policy evaluation, policy improvement and policy iteration. The final output of the DP is an optimal policy that maximizes the final outcome. The complexity of septic shock is the rapidly changing states over the disease course (10). Clinicians need to make rapid decisions to these changing states. There have been numerous clinical practice guidelines for the management of septic shock, but most of these guidelines provide uniform recommendation for all septic shock patients, failing to account for the heterogeneity of individual patients (11-13). On the other hand, many literatures demonstrated that sepsis is a heterogeneous syndrome that the One-size-fit-all paradigm is not working perfectly (14-17). Thus, it is mandatory to utilize the concept of precision medicine for the management of septic shock. The potential implications in clinical practice of the DP algorithm are that it can help to tailor resuscitation strategy conditional on patients' current state. Remember that all states are assigned a value to indicate next appropriate action to take. However, DP is not widely used in real world setting because of its computational complexity. In reality, the state space may be formed by hundreds of features that the size of the feature space is intractable. Another limitation of DP is the requirement of a known MDP, which is usually not the case in clinical researches. Thus, we need more advanced methods such as deep RL, or other methods based on temporal difference. However, the key concept of RL can be captured by DP, as illustrated in this article.

Conclusions

This article illustrates how DP can be used to solve a clinical problem. We show that DP is a potential useful tool to tailor treatment strategy to patients with different conditions/states. Potential audience of the paper are those who are interested in using DP for solving clinical problems with dynamic changing states.

Acknowledgments

Funding: The study was supported by the Key Laboratory of Emergency and Trauma (Hainan Medical University), Ministry of Education (Grant.KLET-202017), the Key Laboratory of Tropical Cardiovascular Diseases Research of Hainan Province (Grant.KLTCDR-202001), and Health Science and Technology Plan of Zhejiang Province (2021KY745).

Footnote

Conflicts of Interest: All authors have completed the ICMJE uniform disclosure form (available at http://dx.doi.org/10.21037/apm-20-2084). The authors have no conflicts of interest to declare.

Ethical Statement: The authors are accountable for all aspects of the work in ensuring that questions related to the accuracy or integrity of any part of the work are appropriately investigated and resolved. Since the study did not involve real human subjects, ethical approval was waived for the study.

References

1. Rahmanian H, Warmuth MK. Online Dynamic Programming. Vol. cs.LG, arXiv.org 2017.
2. Rhodes A, Evans LE, Alhazzani W, et al. Surviving Sepsis Campaign: International Guidelines for Management of Sepsis and Septic Shock: 2016. Intensive Care Med 2017;43:304-77.
3. Hernández G, Ospina-Tascón GA, Damiani LP, et al. Effect of a Resuscitation Strategy Targeting Peripheral Perfusion Status vs Serum Lactate Levels on 28-Day Mortality Among Patients With Septic Shock: The ANDROMEDA-SHOCK Randomized Clinical Trial. JAMA 2019;321:654-64.
4. Elbouhy MA, Soliman M, Gaber A, et al. Early Use of Norepinephrine Improves Survival in Septic Shock: Earlier than Early. Arch Med Res 2019;50:325-32.
5. Rivers E, Nguyen B, Havstad S, et al. Early goal-directed therapy in the treatment of severe sepsis and septic shock. N Engl J Med 2001;345:1368-77.
6. Su L, Tang B, Liu Y, et al. P(v-a)CO2/C(a-v)O2-directed resuscitation does not improve prognosis compared with SvO2 in severe sepsis and septic shock: A prospective multicenter randomized controlled clinical study. J Crit

Care 2018;48:314-20.
7. ProCESS Investigators, Yealy DM, Kellum JA, et al. A randomized trial of protocol-based care for early septic shock. N Engl J Med 2014;370:1683-93.
8. Sutton RS, Barto AG. Reinforcement Learning: An Introduction. Second. The MIT Press, 2018.
9. Angus DC, van der Poll T. Severe sepsis and septic shock. N Engl J Med 2013;369:840-51.
10. Marik PE, Linde-Zwirble WT, Bittner EA, et al. Fluid administration in severe sepsis and septic shock, patterns and outcomes: an analysis of a large national database. Intensive Care Med 2017;43:625-32.
11. Dellinger RP, Levy MM, Rhodes A, et al. Surviving Sepsis Campaign: international guidelines for management of severe sepsis and septic shock, 2012. Intensive Care Med 2013;39:165-228.
12. Perner A, Junttila E, Haney M, et al. Scandinavian clinical practice guideline on choice of fluid in resuscitation of critically ill patients with acute circulatory failure. Acta Anaesthesiol Scand 2015;59:274-85.
13. Dellinger RP, Levy MM, Carlet JM, et al. Surviving Sepsis Campaign: international guidelines for management of severe sepsis and septic shock: 2008. Crit Care Med 2008;36:296-327.
14. Burnham KL, Davenport EE, Radhakrishnan J, et al. Shared and Distinct Aspects of the Sepsis Transcriptomic Response to Fecal Peritonitis and Pneumonia. Am J Respir Crit Care Med 2017;196:328-39.
15. Davenport EE, Burnham KL, Radhakrishnan J, et al. Genomic landscape of the individual host response and outcomes in sepsis: a prospective cohort study. Lancet Respir Med 2016;4:259-71.
16. Wiersema R, Jukarainen S, Vaara ST, et al. Two subphenotypes of septic acute kidney injury are associated with different 90-day mortality and renal recovery. Crit Care 2020;24:150.
17. Zhang Z, Navarese EP, Zheng B, et al. Analytics with artificial intelligence to advance the treatment of acute respiratory distress syndrome. J Evid Based Med 2020;13:301-12.

Cite this article as: Zhang Z, Zhang X, Gu S, Xu X, Jiang W, Lv C, Zheng S. Dynamic programming for solving a simulated clinical scenario of sepsis resuscitation. Ann Palliat Med 2021;10(4):3715-3725. doi: 10.21037/apm-20-2084

