Reinforcement Learning: Homework #5 84

Due on May 15, 2020 at 11:59pm

Professor Ziyu Shao

Yiwei Yang
2018533218

Problem 1 10

1. Applying the Bellman function for value iteration $U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$, as we can see for the optimal value recursive, $\forall N, n > N, V(s_n) = \frac{1}{1-\gamma}$. The induction equation is $V(s_{n-1}) = \gamma V(s_n), for\ n \in \mathbb{N}$ and the starting state: $V(s_1) = \frac{\gamma^{n-1}}{1-\gamma}$. The problem can be reduced to the isometric summation: $V(G) = \Sigma_{k=1}^{\infty} V(s_k) = \Sigma_{k=1}^{\infty} \frac{\gamma^k}{1-\gamma} = \frac{1}{1-\gamma}$

2. An initial policy with action a in both states leads to an unsolvable problem. The initial value determination problem has the form:

$$V(s_1) = \frac{\gamma^{n-1}}{1-\gamma}$$
$$V(s_{n-1}) = \gamma V(s_n), for\ n \in \mathbb{N}$$
$$V(G) = \frac{1}{1-\gamma}$$

   For $\gamma = 0$, it's clear to see $V(s_k) = 1, for\ k \in \mathbb{N}$, meanwhile we have $\forall k, \pi(s_k) = a_0$, thus, it stucks to optimal policy.

   If $\gamma > 0$, from the inconsistent state induction equaiton, we have that the value of γ does not change the order. So, we can find the optimal policy after all; though, the value function has relation with γ. The difference between $\gamma \in (0, 1)$ and $\gamma \in [1, \infty)$, is that the former value can converge to a constant, while the latter diverges to ∞.

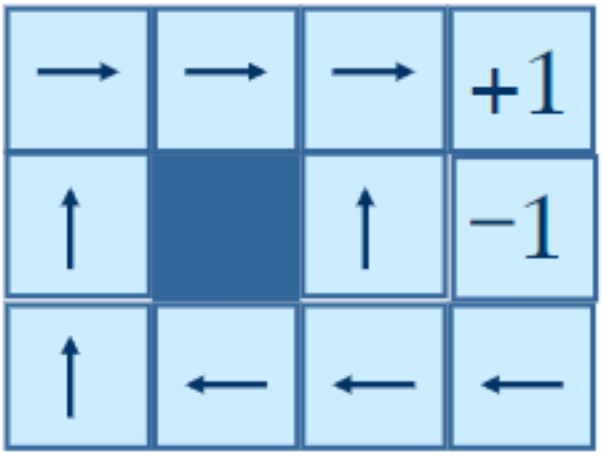
3. Adding constants does not affect the optimal policy, but it does change the value function.

   For theoratical proof:

$$v_{\text{new}}^{\pi}(s_i) = \sum_{t=0}^{\infty} \gamma^t (r_t + c) = \sum_{t=0}^{\infty} \gamma^t r_t + \sum_{t=0}^{\infty} \gamma^t c = v_{\text{old}}^{\pi}(s_i) + \frac{c}{1-\gamma}$$

   For simulation result:

   We set up a literally same model as UCB AI courses does, denote $n = 9$, the setup is elaborated as below:



   function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
   inputs:mdp, an MDP with states S, actions $A(s)$, transition model $P(s'|s,a)$ rewards $R(s)$, discount γ, ϵ, the maximum error allowed in the utility of any state
   local variables: U, U', vectors of utilities for states in S, initially zero.δ, the maximum change in the utility of any state in an iteration

repeat

$$U \leftarrow U'; \delta \leftarrow 0$$

for each state s in S do

$$U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P\left(s' | s, a\right) U\left[s'\right]$$

if $|U'| s] - U|s|| > \delta$ then $\delta \leftarrow |U'| s] - U[s]|$

until $\delta < \epsilon(1-\gamma)/\gamma$

return

The result is done by merely adding the constant reward c, we take the step 15.

before adding c:

0.56999141	0.71061644	0.83561644	1.
0.44499134	0.66504707	0.52054795	$-1.$
0.30913886	0.22232089	0.34732113	0.08650751

Adding $c = 1$:

0.56999142	0.71061644	0.83561644	1.
0.44499138	0.66504707	0.52054795	$-1.$
0.30913898	0.22232101	0.34732117	0.0865076

As we can see, adding constants c does not affect the optimal policy.

4.

$$v^{\pi}_{\text{new}}\ (s_i) = \sum_{t=0}^{\infty} \gamma^t a\left(r_t + c\right) = a \sum_{t=0}^{\infty} \gamma^t r_t + \sum_{t=0}^{\infty} \gamma^t ac = a v^{\pi}_{\text{old}}\ (s_i) + \frac{ac}{1-\gamma}$$

(a) Similarly if $a > 0$, the optimal policy remains unchanged.

(b) $a = 0$, all policies are optimal

(c) $a < 0$,optimal policy is to stay away from G.

Problem 2 10

1. As the $\sum_{t=0}^{\infty} \gamma^t r_t$ and the value function:

Iteration	$V(s_0)$	$V(s_1)$	$V(s_2)$
0	∞	0	$\frac{\gamma^2}{1-\gamma}$
1	∞	1	∞
2	∞	∞	0

As for the action a_1 from state s_0 at time step $t = 0$, we can simply apply the isometric series and get $V = 0 + \gamma + \gamma^2 + \cdots = \frac{\gamma}{1-\gamma}$.

2. As the value function depicted above, for the action a_2 from state s_0 at time step $t = 0$ can be written as the constant series $V = \frac{\gamma^2}{1-\gamma} + 0 + 0 + \cdots = \frac{\gamma^2}{1-\gamma}$. So, it's clear that the optimal stucks to a_1.

Problem 2 continued on next page...

3. We can simply get the bounding by letting Q_{n+1} be the $Q(s_0, a_2) = \frac{\gamma^2}{1-\gamma}$ constant situation when $V_n(s_2) = 0$, because value iteration keep choosing the sub-optimal action while $Q(s_0, a_2) > Q(s_0, a_1)$, this bound is clear. According to the above transition matrix, we have teh value iteration updates recursive:$Q_{n+1}(s_0, a_1) = 0 + \gamma V_n(s_1)$ and $V_{n+1}(s_1) = 1 + \gamma V_n(s_1)$

$$Q_{n+1}(s_0, a_1) = 0 + \gamma(1 + \gamma V_n(s_1)) = \gamma\left(1 + \gamma + \cdots + \gamma^{n-1} + \gamma^n V_{n=0}(s_1)\right) = \gamma\left(\frac{1-\gamma^n}{1-\gamma}\right)$$

Then we apply the basic unequality.

$$\begin{aligned}
Q_{n+1}(s_0, a_1) &= Q(s_0, a_2) \\
\gamma\left(\frac{1-\gamma^{n^*}}{1-\gamma}\right) &= \frac{\gamma^2}{1-\gamma} \\
n^* &= \frac{\log 1 - \gamma)}{\log(\gamma)} \\
&= \frac{\log(1-\gamma)}{\log(1+\gamma-1)} \\
&\geq \log(1-\gamma)\frac{2+\gamma-1}{2(\gamma-1)} \\
&= -\log 1/(1-\gamma)\frac{\gamma+1}{-2(1-\gamma)} \\
&\geq \frac{1}{2}\log\left(\frac{1}{1-\gamma}\right)\frac{1}{1-\gamma}
\end{aligned}$$

Because $\log(1+x) \leq \frac{2x}{2+x}$ for $x \in (-1, 0]$, O.E.D.

Problem 3 10

1. Generally, we have the error bound if $\|U_i - U^*\| < \varepsilon$ then $\|U^{\pi_i} - U^*\| < 2\varepsilon\gamma/(1-\gamma)$. Because of the fact that $\pi(s, \pi(s))$ gives the expected return when starting in s, and $\pi^*(s)$ is $\pi^*(s) = \arg\max_a \sum P(s'|s, a) U(s')$, we can simply get the unequality $\tilde{Q}(s, \pi(s)) \geq \tilde{Q}(s, \pi^*(s))$ by proving $||\tilde{Q} - Q^*|| \leq \epsilon$.

$$\begin{aligned}
V^*(s) - Q^*(s, \pi(s)) &= V^*(s) - \tilde{Q}(s, \pi(s)) + \tilde{Q}(s, \pi(s)) - Q^*(s, \pi(s)) \\
&\leq V^*(s) - \tilde{Q}(s, \pi^*(s)) + \varepsilon \\
&= Q^*(s, \pi^*(s)) - \tilde{Q}(s, \pi^*(s)) + \varepsilon \\
&\leq 2\varepsilon
\end{aligned}$$

2. $V^*(s) - V_\pi(s)$ can be unfolded.

$$\begin{aligned}
V^*(s) - V_\pi(s) &= V^*(s) - Q^*(s, \pi(s)) + Q^*(s, \pi(s)) - V_\pi(s) \\
&\leq 2\varepsilon + Q^*(s, \pi(s)) - Q_\pi(s, \pi(s))
\end{aligned}$$

By getting the expectation of both side. $\mathbb{E}_{s'}[V^*(s') - V_\pi(s')] \leq \frac{2\epsilon}{1-\gamma}$

$$Ans = 2\varepsilon + \gamma\mathbb{E}_{s'}[V^*(s') - V_\pi(s')]$$

O.E.D

3. $Q^*(s, \pi(s))$ represents that the strategy is currently used and the optimal strategy is continued later, recursively. Because in that case, from the graph we have $Q^*(s_1, go) = \frac{2\epsilon}{1-\gamma}$, $Q^*(s_1, stay) == \frac{2\epsilon\gamma}{1-\gamma}$. Meanwhile, the $V^*(s_1) = \frac{2\epsilon}{1-\gamma}$, $V^*(s_2) = \frac{2\epsilon}{1-\gamma}$

4. The question equals to finding a proof that one can find a policy that makes the equation. First, observe that 2ϵ is the gap between two state-value function, the policy with state-action value function $\tilde{Q}$ that $\pi(s_1) = stay$ always holds can be constructed. In that case, to prove the boundness, define the situation near that consistent tie rule. let $\tilde{Q}(s_1, go) = Q^*(s_1, go) - \epsilon$ and $\tilde{Q}(s_1, stay) = Q^*(s_1, stay) + \epsilon$. Eventually we have(a little bit complicated) $V_\pi(s_1) - V*(s_1) = -\frac{2\epsilon}{1-\gamma}$ the bound is tight. O.E.D.

Problem 4 10

1. *lemma Some Denotion*

   (a) *Cumulative discounted return*

   $$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma r_{t+k+1}$$

   (b) *Policy* Policy can be written as $\pi(s, a)$. That is, the probability that action A was performed in state S, which is used to describe a series of actions. It is a function that can take a state and an action and return the probability of taking that action in the current state.

   (c) *Value function*

   i. *state value function*

   $$V^\pi(s) = \mathbb{E}_\pi \left[R_t | s_t = s\right]$$

   ii. *action value function*

   $$Q^\pi(s, a) = \mathbb{E}_\pi \left[R_t | s_t = s, a_t = a\right]$$

   (d) *Probability of state transfer*

   $$P^a_{ss'} = \Pr\left(s_{t+1} = s' | s_t = s, a_t = a\right)$$

   (e) *return expectation*

   $$R^a_{ss'} = \mathbb{E}\left[r_{t+1} | s_t = s, s_{t+1} = s', a_t = a\right]$$

   (f) *Markov decision-making process* It is compose of 4 state $M = \{S, A, P, R\}$, all the state transform observe the Markov property.

   (g) *Optimal Policy* Define a partial ordering over policies, we have $\pi \geq \pi'$ if $v_\pi(s) \geq v_{\pi'}(s), \forall s$

   (h) *Optimal Policy Funciton* The optimal state-value function $v_*(s)$ is the maximum value function over all policies

   $$v_*(s) = \max_\pi v_\pi(s)$$

   The optimal action-value function $q_*(s, a)$ is the maximum action-value function over all policies

   $$q_*(s, a) = \max q_\pi(s, a)$$

2. *lemma Bellman function*

   $$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t | S_t = s\right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p\left(s', r | s, a\right) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s'\right]\right] \text{ As the definition below} \\ &= \sum_a \pi(a|s) \sum_{s', r} p\left(s', r | s, a\right) \left[r + \gamma v_\pi\left(s'\right)\right] \quad \text{for all } s \in S \end{aligned}$$

   $$q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$$

we first have $q_\pi(S_t, A_t) = E_\pi[G_t|S_t, A_t]$ and $q_\pi(S_{t+1}, A_{t+1}) = E_\pi[G_{t+1}|S_{t+1}, A_{t+1}]$.

Then according to Adam's law with extra conditioning we have $E[E(Y|X,Z)|Z] = E(Y|Z)$, denote $Y = G_{t+1}$, $Z = (S_t, A_t)$ and $X = (S_{t+1}, A_{t+1})$

We have

$$E[G_{t+1}|S_t, A_t] = E[E[G_{t+1}|S_{t+1}, A_{t+1}, S_t, A_t]|S_t, A_t] \quad = E[E[G_{t+1}|S_{t+1}, A_{t+1}]|S_t, A_t] = E[q_\pi(S_{t+1}, A_{t+1})|S_t, A_t]$$

Thus, we have $E_\pi[G_{t+1}|S_t = s, A_t = a] = E_\pi[q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$
After that, we can tale the above equation in:$q_\pi(s,a) = E_\pi[G_t|S_t, A_t] = E_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a] = E_\pi[R_{t+1} + \gamma E_\pi[q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] = E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$

3. *the proof of Bellman equation for Markov Reward Processes* The value function can be decomposed into two part which is the immediate reward R_{t+1} and discounted value of successor state $\gamma v(S_{t+1})$.

$$v(s) = \mathbb{E}\left[G_t|S_t = s\right]$$

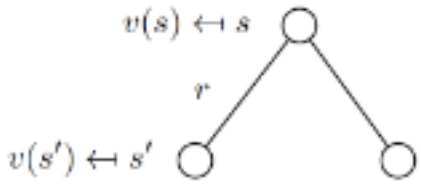
Because of Adam's theorem and Markov Property, we have $V(S_{t+1}) = E[G_{t+1}|S_{t+1}]$ Because of Adam's Law $E[E(Y|X)] = E(Y)$, we also have Adam's Law with extra conditioning $\hat{E}(\hat{E}(Y|X)) = \hat{E}(Y)$ ¡=¿ $E[E(Y|X,Z)|Z] = E(Y|Z)$

$$\begin{aligned} Ans &= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma\left(R_{t+2} + \gamma R_{t+3} + \ldots\right)|S_t = s\right] \end{aligned}$$

We have

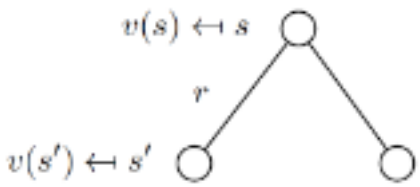
$$\begin{aligned} Ans &= \mathbb{E}[G_t|S_{t=s}] \\ &= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1}|S_t = s\right] \\ &= \mathbb{E}[R_{t+1}|S_{t=s}] + \gamma E[G_{t+1}|S_{t=s}] \\ &= \mathbb{E}[R_{t+1}|S_{t=s}] + \gamma E[V(S_{t+1})|S_{t=s}] \\ &= \mathbb{E}\left[R_{t+1} + \gamma v\left(S_{t+1}\right)|S_t = s\right] \end{aligned}$$

Continue, we can apply the *LOTE*



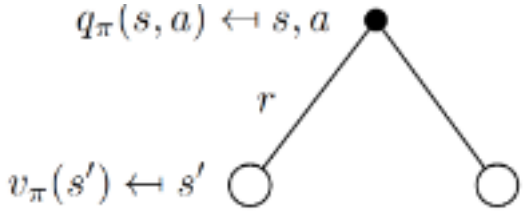
$$\begin{aligned} Ans &= R_s + \gamma \sum_{s' \in S} E[V(S_{t+1})|S_t = s, S_{t+1=s'}]P(S_{t+1=s'}|S_t = s) \\ &= R_s + \gamma \sum_{s' \in S} V(s')P_{ss'} \end{aligned}$$

4. *the proof of Bellman Expectation equation for Markov Process Processes*



For V^π, we have simply apply the MDP property and LOTP to get

$$V_\pi(s) = E_\pi[G_t|S_t = s] = \sum_{a\in A} E_\pi[G_t|S_t = s, A_t = a] = \sum_{a\in A} \pi(a|s)q_\pi(s,a)$$



For Q_π, the difference is merely value and reward function difference:

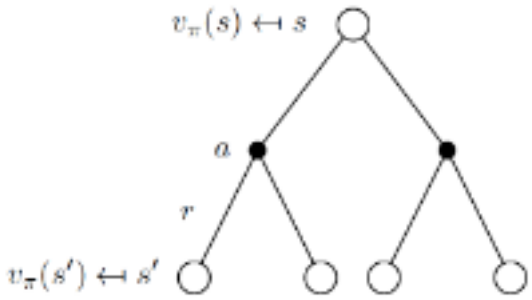
$$\begin{aligned} E[q_\pi(S_{t+1}, A_{t+1})|S_{t+1} = s', S_t = s, A_t = a] &= E[q_\pi(S_{t+1}|A_{t+1}|S_{t+1} = s')] \\ &= \sum_{a\in A} E[q_\pi(S_{t+1}, A_{t+1})|S_{t+1} = s', A_{t+1} = a] * P(A_{t+1} = a|S_{t+1} = s') \\ &= \sum_{a\in A} a_\pi(s', a)\pi(a|s') \\ &= v_\pi(s') \end{aligned}$$

So, we can take in to from above and applyt the $LOTE$.

$$\begin{aligned} Ans &= \sum_{s'\in S} E[q_\pi(S_{t+1}, A_{t+1})|S_{t+1} = s', S_t = s, A_t = a] * P(S_{t+1} = s'|S_t = s, A_t = a) \\ &= \sum_{a\in A} v_\pi(s') * P^\pi_{ss'} \end{aligned}$$

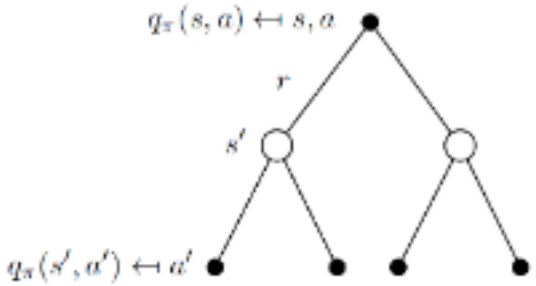
So,

$$\begin{aligned} q_\pi(s,a) &= E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \\ &= E[R_{t+1}|S_t = s, A_t = a] + \gamma E[q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \\ &= R^a_s + \gamma \sum_{s'\in S} P^a_{ss'} r_\pi(s') \end{aligned}$$



As for the second layer. Take the $q_\pi(s,a) = R_s^a + \gamma \sum_{n' \in S} P_{ss'}^a r_\pi(s')$ for the second layer to the first layer $V_\pi(s) = \sum_{a \in A} \pi(a|s) q_\pi(s,a)$

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$



Similarly, reversely take in the equation above we have: $q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}}$ and $\mathcal{P}_{ss'}^a = \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a')$

The Bellman expectation equation can also be expressed concisely using the induced MRP.

$$v_x = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_\pi$$

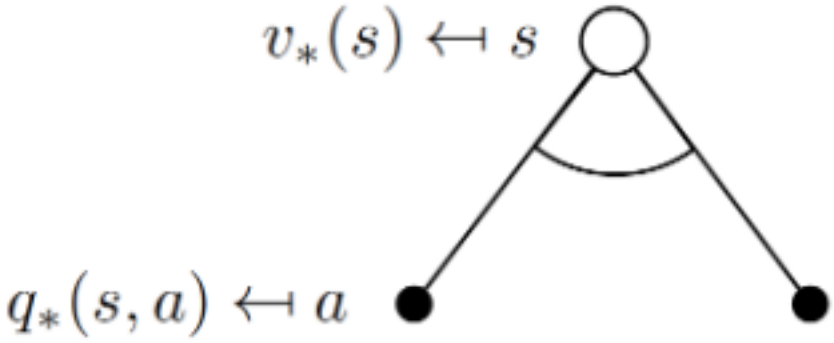
with direct solution

$$v_\pi = (I - \gamma \mathcal{P}^\pi)^{-1} \mathcal{R}^\pi$$

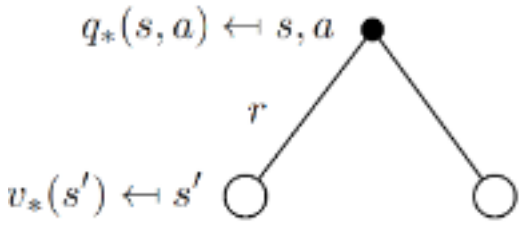
5. *the proof of Bellman Optimality equation for Markov Process Processes* An optimal policy can be found by maximizing over $q_*(s,a)$

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{s \in \mathcal{A}}{\operatorname{argmax}}\, q_*(s,a) \\ 0 & \text{otherwise} \end{cases}$$

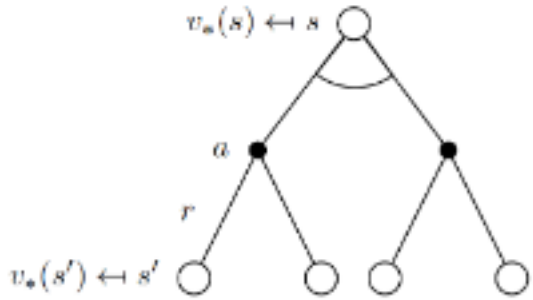
Because, if π' is a random optimal policy, $0 < \pi'(a|s) < 1, \forall a \in A$, we have $V_*(s) = V_{\pi'(s)} = \sum_P a \in A \pi'(a|s) q_{\pi'}(s,a) \leq \sum_P a \in A \pi'(a|s) q_*(s,a) \leq \sum_P a \in A \pi'(a|s) q_*(s,a^*) = q_*(s,a^*)$
So, we get $V_*(s) \geq V_{\pi^*}(s) = \sum_{a \in A} \pi^*(a|s) q_{\pi^*}(s,a) = q_*(s,a^*)$



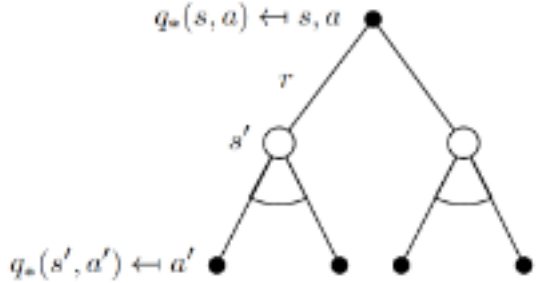
From the above equation we know that the choice is between picking exactly the action $\pi_*(a|s)$ and picking a probability distribution over potentially optianal and non-optional actions, and pick best action preferably. After all based on first and second equation, we have $v_\pi(s) = q_*(s,a^*) = max_{a \in A} q_*(s,a)$



Firstly, $q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a r_\pi(s')$, so we get $q_*(s,a) = max_\pi q_\pi(s,a) = R_n^a + \gamma \sum_{s' \in S} P_{ss'}^a max_\pi v_\pi(s') = R_n^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')$
Then applying the LOTE, we get $Ans = \sum_{s' \in S} E[v_*(S_{t+1}) | S_{t+1} = s", S_t = s, A_t = a] * P(S_{t+1} = S' | S_t = s, A_T = a) = \sum_{s' \in S} V_*(s') * P_{ss'}^a$
Lastly, we have, $q_*(s,a) = E[R_{t+1} + \gamma V_*(S_{t+1}) | S_t, A_t] = \mathbb{R}_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')$



Apply the same techniques with Expectation function we have: $v_*(s) = \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right)$



Apply the same techniques with Expectation function we have: $q_*(s,a) = \mathcal{R}_s^2 + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a')$
Also, we have the matrix forms of Optimality equation

$$v_*(s_1) == \max \left\{ \begin{array}{l} p(s_1|s_1,a_1)\left[r(s_1,a_1,s_1) + \gamma v_*(s_1)\right] + p(s_2|s_1,a_1)\left[r(s_2,a_1,s_1) + \gamma v_*(s_2)\right] \\ p(s_1|s_1,a_2)\left[r(s_1,a_2,s_1) + \gamma v_*(s_1)\right] + p(s_2|s_1,a_2)\left[r(s_2,a_2,s_1) + \gamma v_*(s_2)\right] \\ \cdots \\ p(s_1|s_1,a_n)\left[r(s_1,a_n,s_1) + \gamma v_*(s_1)\right] + p(s_2|s_1,a_n)\left[r(s_2,a_n,s_1) + \gamma v_*(s_2)\right] \end{array} \right\}$$

Problem 5 10

1. First solve the MRP by taking them into the matrix solving equation. We apply the state transition probability matrix $P =$

	$C1$	$C2$	$C3$	P_{ass}	P_{Ub}	FB	Sleep
$C1$		0.5				0.5	
$C2$			0.8				0.2
$C3$				0.0	0.4		
$Pass$							1.0
P_{ub}	0.2	0.4	0.4				
FB	0.1					0.9	
$Sleep$							1

Taking the matrix into $\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{11} & \dots & \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}$, ew get $R = \begin{bmatrix} C1 & -2 \\ C2 & -2 \\ C3 & -2 \\ Pass & 10 \\ P_{ub} & 1 \\ FB & -1 \\ Sleep & 0 \end{bmatrix}$

Eventually, applyin the equation $V = (I - \gamma P)^{-1} R$. We have if $\gamma = 1$, we have $\begin{bmatrix} \text{-12.5432} \\ 1.4568 \\ 4.3210 \\ 10.0 \\ 0.8025 \\ \text{-22.5432} \\ 0 \end{bmatrix}$ which is in accordance with the simulaiton result.

2. First let us compute some of the inferior π value of the state. We get $\pi(study|s_1) = 0.5$, $\pi(Facebook|s_1) = 0.5$, $\pi(sleep|s_2) = 0.5$, $\pi(study|s_3) = 0.5$, $\pi(pub|s_3) = 0.5$. And we can get the value of P. $P^{study}_{s_1,s_2} = 1$, $P^{Pub}_{s_3,s_1} = 0.2$, $P^{Pub}_{s_3,s_2} = 0.4$, $P^{Pub}_{s_3,s_3} = 0.4$.
Then we can get $v_1 = v_\pi(s_1) = 0.5(R^{Study}_{s_1} + 1 * 1 * V_\pi(s_2)) + 0.5(R^{Facebook}_{s_1} + 1 * 1 * V_\pi(s_4)) = 0.5(-2 + v_2) + 0.5(-1 + v_4)$
Similarly, after get all the function, we can get $\begin{bmatrix} v1 & \text{-1.3} \\ v2 & \text{-2.7} \\ v3 & 7.4 \\ v4 & \text{-2.3} \end{bmatrix}$ Similarly, take the computed v_i into the bellmen equation for MDP q_π, we have $q_\pi(s_1, study) = -2+1*v_2 = -2+2.7 = 0.7$,$q_\pi(s_1, Facebook) = -1 + 1 * v_4 = -1 + 2.3 = -3.3$, $q_\pi(s_2, sleep) = 0 + 0 = 0$, $q_\pi(s_2, study) = -2 + 1 * v_3 = -2 + 7.4 = 5.4$, $q_\pi(s_3, study) = 10 + 0 = 10$, $q_\pi(s_3, pub) = 1 + 0.2 * (-1.3) + 0.4 * (2.7) + 0.4 * 7.8 = 4.78$, $q_\pi(s_4, facebook) = -1 + v_4 = -1 - 2.3 = -3.3$, $q_\pi(s_4, quit) = 0 + v_1 = 0 - 1.3 = -1.3$ As for the state action function, we have

3. First let us test the optimality bellman equation of s_1, we have $V_*(s_1) = max\{-2+V_*(s_2), -1+V_*(s_4)\}$, the former is aciton=study and the latter is when action=facebook. $Ans = max\{-2+8, -1+6\} = 6$, so $a^*(s_1) = study$.
Then apply the q optimality funtion to calculate the following one, that is $V_\pi(sleep) = 0 \forall \pi$, so we have $q_\pi(s_2, sleep) = R^{sleep}_{s_2} + 1 * v_\pi(sleep) = 0 + 0 \forall \pi$, so $\pi(s_2, sleep) = 0$
Also, on the other hand, we have $q_\pi(s_3.study) = R^{study}_{s_3} + 1 * V_\pi(sleep) = 10 + 0 = 10 \forall \pi$, so

$q_*(s_3, study) = 10$
Similarly, we have $q_*(s_1, study) = -2+max[0, q_*(s_2, study)]$, $q_*(s_1, facebook) = R_{s_1}^{facebook}+\gamma P_{s1,s4}^{Facebook}*$ $max\ q_*(s_4, a') = -1 + max[q_*(s_4, facebook), q_*(s_4, quit)]$, and $q_*(s_2, sleep) = 0$, $q_*(s_2, study) = -2 +$ $max[10, 1_*(s_3, pub)]$, $q_*(s_3, study) = 10$, $q_*(s_3, pub) = 1+0.2*max[q_*(s_1, study), q_*(s_1, framework)]+$ $0.4*max[q_*(s_2, study), 0]+0.4*max[q_*(s_3, pub), 10]$, $q_*(s_4, facebook) = -1+max[q_*(s_4, Facebook), q_*(s_4, quiy)]$, $q_*(s_4, quit)$.
Also, we have the bound got by max() func. We have $q_*(s_2, study) = -2 + max[10, q_*(s_3, pub)] \geq 8$, $q_*(s_1, study) \geq 6$, $q_*(s_3, pub) = 0.6 + 0.6 * q_*(s_2, study) + 0.4max[q_*(s_3, \pi_6), 10]$
By max() function's bound, we have $q_*(s_3, pub) = 0.6 + 0$,

Enentually, we have $\begin{bmatrix} v_*(s_1) = 6 \\ v_*(s_2) = 8 \\ v_*(s_3) = 10 \\ v_*(s_4) = 6 \\ v_*(sleep) = 0 \end{bmatrix}$

Problem 6 10

1. The original formula is:
$$\sum_{s},_{\in S} \sum_{r \in R} p\left(s', r|s, a\right) = 1, \forall s \in S, a \in A(s)$$
As in a finite MDP, the sets of states, actions, and rewards ($8, \mathcal{A}$, and $\mathcal{R}$) all have a finite number of elements. In this case, the random variables R_t and S_t have well defined discrete probability distributions dependent only on the preceding state and action. That is, for particular values of these random variables, $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, there is a probability of those values occurring at time t, given particular values of the preceding state and action:
$$p\left(s', r|s, a\right) \doteq \Pr\left\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\right\}$$
for all $s', s \in \mathcal{S}, r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$. The function p defines the dynamics of the MDP. The dot over the equals sign in the equation reminds us that it is a definition rather than a fact that follows from previous definitions. The dynamics function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is an ordinary deterministic function of four arguments. The | in the middle of it comes from the notation for conditional probability, but here it just reminds us that p specifies a probability distribution for each choice of and a, that is, that $\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p\left(s', r|s, a\right) = 1$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
For episodic tasks the set of terminal and non-terminal states can be denoted as S+. Therefore, the $\sum s' \in S \sum r \in Rp(s', r|s, a) = 1, \forall s \in S^+, a \in A(s)$ as the dynamics of the MDP in an episodic task include as a possible transition those ending in a terminal state.

2. Applying the simulation:
   (a) The optimal value funtion over all possible policies:

21.974724.4163	21.974719.4163	17.4747
19.775421.9747	19.775417.7979	16.0181
17.797919.7754	17.797916.0181	14.4163
16.018117.7979	16.018114.4163	12.9747
14.416316.0181	14.416312.9747	11.6754

Problem 6 continued on next page...

(b) The optimal policy:

$['e']['n','w','s','e']$	$['w']['n','w','s','e']$	$['w']$
$['n','e']['n']$	$['n','w']['w']$	$['w']$
$['n','e']['n']$	$['n','w']['n','w']$	$['n','w']$
$['n','e']['n']$	$['n','w']['n','w']$	$['n','w']$
$['n','e']['n']$	$['n','w']['n','w']$	$['n','w']$

3. Let us do the calculation similar to the prevvious problem
$q_\star(A,W) = q_\star(A,E) = 0.9{*}v_\star(s_0) = 0.9{*}22 = 19.8$ $q_\star(A,N) = -1{+}0.9{*}v_\star(A) = -1{+}0.9{*}24.4 = 21.0$
$q_\star(A,S) = -1 + 0.9 * v_\star(') = 10 + 0.9 * 16.0 = 24.4$
$q_\star(B,W) = 0.9 * v_\star(s_2) = 0.9 * 22.0 = 19.8$ $q_\star(B,S) = 6 + 0.8 * 16.0 = 19.8$

The following can be gotten as:

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

The optimal policy is:

$['e']['n','w','s','e']$	$['w']['n','w','s','e']$	$['w']$
$['n','e']['n']$	$['n','w']['w']$	$['w']$
$['n','e']['n']$	$['n','w']['n','w']$	$['n','w']$
$['n','e']['n']$	$['n','w']['n','w']$	$['n','w']$
$['n','e']['n']$	$['n','w']['n','w']$	$['n','w']$

(a) The optimal value funtion over all possible policies:

21.974724.4163	21.974719.4163	17.4747
19.775421.9747	19.775417.7979	16.0181
17.797919.7754	17.797916.0181	14.4163
16.018117.7979	16.018114.4163	12.9747
14.416316.0181	14.416312.9747	11.6754

(b) The optimal policy:

$['e']['n','w','s','e']$	$['w']['n','w','s','e']$	$['w']$
$['n','e']['n']$	$['n','w']['w']$	$['w']$
$['n','e']['n']$	$['n','w']['n','w']$	$['n','w']$
$['n','e']['n']$	$['n','w']['n','w']$	$['n','w']$
$['n','e']['n']$	$['n','w']['n','w']$	$['n','w']$

Problem 7 6

The question equivalent to the random walk denoted by the following definition. if in position n move to $n+1$ with probability p move to $n-1$ with probability q stay at n with probability r with

$$p + q + r = 1$$

In many such games the odds of winning are very close to 1 with p typically around .49. Using our second order difference equations we will show that even though the odds are only very slightly in favor of the casino, this enough to ensure that in the long run, the casino will makes lots of money and the gambler not so much. We also investigate what is the better strategy for a gambler, play small amounts of money (be cautious) or play big amounts of money (be bold). We shall see that being bold is the better strategy if odds are not in your favor (i.e. in casino), while if the odds are in your favor the better strategy is to play small amounts of money. We say that the game is fair if $p = q$ subfair if $p < q$ superfair if $p > q$
The gambler's ruin equation: In order to make the previous problem precise we imagine the following situation.

- You starting fortune if \$j
- In every game you bet \$1
- Your decide to play until you either loose it all (i.e., your fortune is 0) or you fortune reaches \$N and you then quit.

To compute x_j we use the formula for conditional probability and condition on what happens at the first game, win, lose, or tie. For every game we have

$$P(\text{ win }) = p, \quad P(\text{ lose }) = q, \quad , P(\text{ tie }) = r$$

We have

$$\begin{aligned} x_j &= P(A_j) \\ &= P(A_j|\text{ win }) P(\text{ win }) + P(A_j|\text{ lose }) P(\text{ lose }) + P(A_j|\text{ tie }) P(\text{ tie }) \\ &= x_{j+1} \times p \quad + \quad x_j \times q \quad + \quad x_{j-1} \times r \end{aligned}$$

since if we win the first game, our fortune is then $j + 1$, and so $P(A_j|\text{ win }) = P(A_j + 1)$ is simply x_{j+1}, and son on...
Note also that we have $x_0 = P(A_0) = 0$ since we have then nothing more to gamble and $x_N = P(A_N)$ since we have reached our goal and then stop playing. Using that $p + q + r = 1$ we can rewrite this as the second order equation Gambler's ruin

$$px_{j+1} - (p + q)x_j + qx_{j-1} = 0, \quad x_0 = 0, x_N = 1$$

With $x_j = \alpha^j$ we find the quadratic equation

$$p\alpha^2 - (p + q)\alpha + q = 0$$

with solutions

$$\alpha = \frac{-p \pm \sqrt{(p+q)^2 - 4pq}}{2p} = \frac{-p \pm \sqrt{p^2 + q^2 - 2pq}}{2p} = \frac{-p \pm \sqrt{(p-q)^2}}{2p} = \begin{cases} 1 \\ q/p \end{cases}$$

If $p \neq$ we have two solutions and and so the general solution is given by

$$x_n = C_1 1^n + C_2 \left(\frac{q}{p}\right)^n$$

We will consider the case $p = q$ later. To determine the constants C_1 and C_2 we use that

$$x_0 = 0, \quad \text{and } x_N = 1$$

which follow from the definition of x_j as the probability to win (i.e. reaching a fortune of N) starting with a fortune of j. We find

$$0 = C_1 + C_2, \quad 1 = C_1 + C_2 \left(\frac{q}{p}\right)^N$$

Problem 7 continued on next page...

which gives

$$C_1 = -C_2 = \left(1 - \left(\frac{q}{p}\right)^N\right)^{-1}$$

and so we find Gambler's ruin probabilities $x_n = \frac{1-(q/p)^n}{1-(q/p)^N} \quad p \neq q$

Formula for bold play probabilities: We derive the basic equations for the bold play strategy. If your fortune z is less than $1/2$ then you bet z and ends up with fortune of $2z$ if you win and nothing if you loose. So by conditioning we find

$$Q(z) = Q(z|W)pP(w) + Q(z|L)P(L) = pQ(2z) + Q(0)q = pQ(2z)$$

On the other hand if your fortune z exceeds $1/2$ you will bet only $1 - z$ to reach 1. By conditioning you find

$$Q(z) = Q(z|W)pP(w) + Q(z|L)P(L) = Q(1)p + Q(z - (1 - z))q = p + qQ(2z - 1)$$

In summary we have Bold play conditional probabilities

$$\begin{aligned} Q(z) &= pQ(2z) \quad \text{if } z \leq 1/2 \\ Q(z) &= p + qQ(2z - 1) \text{ if } z \geq 1/2 \\ Q(0) &= 0, \quad Q(1) = 1 \end{aligned}$$

正如习题课上分享的，可以从动态规划的角度思考。

Problem 8 10

1. We can regard this situation as a sequential decision process in which we say that we are in state i if the i th offer has just been presented and it is the best of the i offers already presented. Letting $V(i)$ denote the best we can do in this position, we find that V satisfies

   $$V(i) = \max[P(i), H(i)]$$

   where $P(i)$, the probability that the best offer will be realized if the i th is accepted, is given by $P(i) = P($ offer is best of $n|$ offer is best of first $i)$

   $$= \frac{1/n}{1/i} = \frac{i}{n}$$

   and where $H(i)$ represents the best we can do if we reject the ith offer. Hence we have

   $$V(i) = \max\left[\frac{i}{n}, H(i)\right], \quad i = 1, \ldots, n$$

   is now easy to see that $H(i)$ is just the maximal probability of accepting the best offer when we have rejected the first i offers. But because the situation in which the first i offers have been rejected is clearly at least as good as that in which the first $i + 1$ have been rejected (because the next one can always be rejected), it follows that $H(i)$ is decreasing in i Because i/n increases and $H(i)$ decreases in i, it follows that for some

   $$\begin{aligned} \tfrac{i}{n} &\leq H(i) \quad (i \leq j) \\ \tfrac{i}{n} &> H(i) \quad (i > j) \end{aligned}$$

   Hence, the optimal policy is of the following form: for some j $j \leq n - 1$, reject the first j offers and then accept the first candidate offer to appear, where an offer is said to be a candidate if it is of higher value than any of its predecessors.

Problem 8 continued on next page...

Letting P_j (best) denote the probability of obtaining the best prize under such a strategy, we have (conditioning on the prize that is accepted.

$$P_j(\text{ best }) = \sum_{i=1}^{n-j} P_j(\text{ best } | i+j \text{ prize is accepted }) P_j(i+j \text{ accepted })$$

Now,

$$\begin{aligned} P_j(\text{ best } | i+j \text{ accepted }) &= P(\text{ best of } n | \text{ best of } i+j) \\ &= \frac{i+j}{n} \end{aligned}$$

Also, $P_j(i+j \text{ accepted }) = P(\text{ best of first } j = \text{ best of first } i+j-1$

$$\begin{aligned} & i+j = \text{ best of first } i+j \text{)} \\ &= P(\text{ best of first } j = \text{ best of first } i+j-1) \\ & xP(i+j = \text{ best of first } i+j) \\ &= \left(\tfrac{j}{i+j-1}\right)\left(\tfrac{1}{i+j}\right) \end{aligned}$$

Hence,

$$\begin{aligned} P_j(\text{ best }) &= \frac{j^{n-j}}{n} \sum_{i=1}^{n-j} \frac{1}{i+j-1} \\ &= \frac{j}{n} \sum_{k=j}^{n-1} \frac{1}{k} \\ &\approx \frac{j}{n} \int_j^{n-1} \frac{1}{x} dx \\ &= \frac{j}{n} \log\left(\frac{n-1}{j}\right) \\ &\approx \frac{j}{n} \log\left(\frac{n}{j}\right) \end{aligned}$$

Now, if we let $g(x) = (x/n)\log(n/x)$, then

$$g'(x) = \frac{1}{n} \log \frac{n}{x} - \frac{1}{n}$$

so

$$g'(x) = 0 \Rightarrow \log \frac{n}{x} = 1 \Rightarrow x = \frac{n}{e}$$

Also, because

$$g\left(\frac{n}{e}\right) = \frac{1}{e}$$

we see that the optimal policy is, for n large, approximately to let the fraction 1/e of all prizes go by and then accept the first candidate. The probability that this procedure will result in the best prize is roughly $1/e$

Problem 9 8

1. The intuitive way can be the gittins indices, which works quite well.The idea behind Gittins indices works as follows. Assume that we are playing a single slot machine, and that we have the choice of continuing to play the slot machine or stopping and switching to a process that pays a reward r. If we choose not to play, we receive r, and then find ourselves in the same state (since we did not collect any

new information). If we choose to play, we earn a random amount W, plus we earn $\mathbb{E}\left\{V\left(S^{n+1},r\right)|S^n\right\}$, where S^{n+1} represents our new state of knowledge resulting from our observed winnings. For reasons that will become clear shortly, we write the value function as a function of the state S^{n+1} and the stopping reward r The value of being in state S^n, then, can be written as

$$V\left(S^n,r\right)=\max\left[r+\gamma V\left(S^n,r\right),\mathbb{E}\left\{W^{n+1}+\gamma V\left(S^{n+1},r\right)|S^n\right\}\right]$$

The first choice represents the decision to receive the fixed reward r, while in the second choice we get to observe W^{n+1} (which is random when we make the decision). When we have to choose x^n, we will use the expected value of our return if we continue playing, which is computed using our current state of knowledge. For example, in the Bayesian normal-normal model, $\mathbb{E}\left\{W^{n+1}|S^n\right\}=\theta^n$, which is our estimate of the mean of W given what we know after the first n measurements.

If we choose to stop playing at iteration n, then S^n does not change, which means we earn r and face the identical problem again for our next play. In this case, once we decide to stop playing, we will never play again, and we will continue to receive r (discounted) from now on. For this reason, r is called the retirement reward. The infinite horizon, discounted value of retirement is $r/(1-\gamma)$. This means that we can rewrite our optimality recursion as

$$V\left(S^n,r\right)=\max\left[\frac{r}{1-\gamma},\mathbb{E}\left\{W^{n+1}+\gamma V\left(S^{n+1},r\right)|S^n\right\}\right]$$

Here is where we encounter the magic of Gittins indices. We compute the value of r that makes us indifferent between stopping and accepting the reward r (forever), versus continuing to play the slot machine. That is, we wish to solve the equation

$$\frac{r}{1-\gamma}=\mathbb{E}\left\{W^{n+1}+\gamma V\left(S^{n+1},r\right)|S^n\right\}$$

for r. The Gittins index $I^{Gitt,n}$ is the particular value of r that solves (6.3). This index depends on the state S^n. If we use a Bayesian perspective and assume normally distributed rewards, we would use $S^n=(\theta^n,\beta^n)$ to capture our distribution of belief about the true mean μ. If we use a frequentist perspective, our state variable would consist of our estimate $\bar{\theta}^n$ of the mean, our estimate $\hat{\sigma}^{2,n}$ of the variance, and the number N^n of observations (this is equal to n if we only have one slot machine) If we have multiple slot machines, we consider every machine separately, as if it were the only machine in the problem. We would find the Gittins index $I_x^{Gitt,n}$ for every machine x. Gittins showed that, if $N\to\infty$, meaning that we are allowed to make infinitely many measurements, it is optimal to play the slot machine with the highest value of $I_x^{\text{Git }t,n}$ at every time n. Notice that we have not talked about how exactly (6.3) can be solved. In fact, this is a major issue, but for now, assume that we have some way of computing $I_x^{\text{Git },n}$.

Recall that, in ranking and selection, it is possible to come up with trivial policies that are asymptotically optimal as the number of measurements goes to infinity. For example, the policy that measures every alternative in a round-robin fashion is optimal for ranking and selection: If we have infinitely many chances to measure this policy will measure every alternative infinitely often, thus discovering the true best alternative in the limit. However, in the multi-armed bandit setting, this simple policy is likely to work extremely badly. It may discover the true best alternative in the limit, but it will do poorly in the early iterations. If $\gamma<1$, the early iterations are more important than the later ones, because they contribute more to our objective value. Thus, in the online problem, it can be more important to pick good alternatives in the early iterations than to find the true best alternative. The Gittins policy is the only policy with the ability to do this optimally.

2. Let us consider the beta-Bernoulli model for a single slot machine. Each play has a simple 0/1 outcome (win or lose), and the probability of winning is ρ. We do know this probability exactly, so we assume

that ρ follows a beta distribution with parameters α^0 and β^0. Recall that the beta-Bernoulli model is conjugate, and the updating equations are given by

$$\begin{aligned}\alpha^{n+1} &= \alpha^n + W^{n+1}\\ \beta^{n+1} &= \beta^n + \left(1 - W^{n+1}\right)\end{aligned}$$

where the distribution of W^{n+1} is Bernoulli with success probability ρ. After n plays, the distribution of ρ is beta with parameters α^n and β^n. The knowledge state for a single slot machine is simply $S^n = (\alpha^n, \beta^n)$. Consequently,

$$\begin{aligned}\mathbb{E}\left(W^{n+1}|S^n\right) &= \mathbb{E}\left[\mathbb{E}\left(W^{n+1}|S^n, \rho\right)|S^n\right]\\ &= \mathbb{E}\left(\rho|S^n\right)\\ &= \frac{\alpha^n}{\alpha^n + \beta^n}\end{aligned}$$

Then, writing $V\left(S^n, r\right)$ as $V\left(\alpha^n, \beta^n, r\right)$, we obtain

$$\begin{aligned}\mathbb{E}\left\{W^{n+1} + \gamma V\left(S^{n+1}, r\right)|S^n\right\} =& \frac{\alpha^n}{\alpha^n + \beta^n} + \gamma\frac{\alpha^n}{\alpha^n + \beta^n}V\left(\alpha^n + 1, \beta^n, r\right)\\ &+ \gamma\frac{\beta^n}{\alpha^n + \beta^n}V\left(\alpha^n, \beta^n + 1, r\right)\end{aligned}$$

For fixed α and β, the quantity $V(\alpha, \beta, r)$ is a constant. However, if the observation W^{n+1} is a success, we will transition to the knowledge state $(\alpha^n + 1, \beta^n)$; and if it is a failure, the next knowledge will be $(\alpha^n, \beta^n + 1)$. Given S^n, the conditional probability of success is $\frac{\alpha^n}{\alpha^n+\beta^n}$

If we let $R() = V()$ and take $\alpha_{1,2}$ and $\beta_{1,2}$ in, we have $R_1\left(\alpha_1, \beta_1\right) = \frac{\alpha_1}{\alpha_1+\beta_1}\left[1 + \gamma R\left(\alpha_1 + 1, \beta_1, \alpha_2, \beta_2\right)\right] + \frac{\beta_1}{\alpha_1+\beta_1}\left[\gamma R\left(\alpha_1, \beta_1 + 1, \alpha_2, \beta_2\right)\right]$ $R_2\left(\alpha_2, \beta_2\right) = \frac{\alpha_2}{\alpha_2+\beta_2}\left[1 + \gamma R\left(\alpha_1, \beta_1, \alpha_2 + 1, \beta_2\right)\right] + \frac{\beta_2}{\alpha_2+\beta_2}\left[\gamma R\left(\alpha_1, \beta_1, \alpha_2, \beta_2 + 1\right)\right]$

For the upper boundness, if we fix a value of r. If $\alpha + \beta$ is very large, it is reasonable to suppose that

$$V(\alpha, \beta, r) \approx V(\alpha + 1, \beta, r) \approx V(\alpha, \beta + 1, r)$$

Then, we can combine 2 equations to approximate the Gittins recursion as

$$V(\alpha, \beta, r) = \max\left[\frac{r}{1-\gamma}, \frac{\alpha}{\alpha+\beta} + \gamma V(\alpha, \beta, r)\right]$$

In this case, it can be shown that the obove equation has the solution

$$V(\alpha, \beta, r) = \frac{1}{1-\gamma}\max\left(r, \frac{\alpha}{\alpha+\beta}\right)$$

Thus, we have $R\left(\alpha_1, \beta_1, \alpha_2, \beta_2\right) = \max\left\{R_1\left(\alpha_1, \beta_1\right), R_2\left(\alpha_2, \beta_2\right)\right\}$.

3. For it have to take n times loop and a larger effective time horizon with the larger of times. the complexity is $O(k * G(\frac{1}{\sqrt{k}}, \gamma)) = O(k * \gamma)$. For approximation for speedup, we can make $G(s, r) = \sqrt{-log\gamma} * b * (-\frac{s^2}{(}log\ \gamma))$, b is given by$\tilde{b}(s) = \begin{cases} \frac{s}{\sqrt{2}}, & s \le \frac{1}{7} \\ e^{-0.02645(\log s)^2 + 0.89106\log s - 0.4873}, & \frac{1}{7} < s \le 100 \\ \sqrt{s}(2\log s - \log\log s - \log 16\pi)^{\frac{1}{2}}, & s > 100 \end{cases}$ This approximation reduce the time of taking γ times to evaluate G, which only takes k times. The quality of the approximation is quite well.

4. The technique to get the optimal solution is called Gittins Index Theorem plus dynamic programing. This is a forward algorithm.

Problem 9 continued on next page...

The performance objective in the Bayesian Bernoulli MABP is to maximize the Expected Total Discounted (ETD) number of successes after T observations, letting $0 \leq d < 1$ be the discount factor. Then, the corresponding bandit optimization problem is to find a discount-optimal policy such that

$$V_D^*\left(\widetilde{\mathbf{x}}_0\right) = \max_{\pi \in \Pi} \mathrm{E}^{\pi}\left[\sum_{t=0}^{T-1} \sum_{k=1}^{K} d^t \frac{s_{k,0}+S_{k,t}}{s_{k,0}+f_{k,0}+S_{k,t}+F_{k,t}} \cdot a_{k,t} | \widetilde{\mathbf{x}}_0 = \left(\mathrm{x}_{k,0}\right)_{k=1}^{K}\right]$$

The regret is measured by $\rho = T \max_k \{p_k\} - \mathrm{E}^{\pi}\left[\sum_{t=0}^{T-1} \sum_{k=1}^{K} a_{k,t} Y_{k,t}\right]$. We now review the solution giving for some $(p_k)_{k=1}^{K}$
the optimal policy to optimization problem in the infinite-horizor setting by letting $T = \infty$. In general, as MABPs are a special class of MCPs, the traditional technique to address them is via a dynamic programming (DP) approach. Thus, the solution to , according to Bellman's principle of optimality (Bellman, 1952), is such that for every $t = 0, 1, \ldots$ the following DP equation holds:

$$\begin{aligned} V_D^*\left(\mathbf{x}_{1,t}, \ldots, \mathbf{x}_{K,t}\right) = \max_k \Big\{ & \tfrac{s_{k,0}+s_{k,t}}{s_{k,0}+f_{k,0}+s_{k,t}+f_{k,t}} \\ & + d\Big(\tfrac{s_{k,0}+s_{k,t}}{s_{k,0}+f_{k,0}+s_{k,t}+f_{k,t}} \cdot V_D^*\left(\mathbf{x}_{1,t}, \mathbf{x}_{k,t}+\mathbf{e}_1, \ldots, \mathbf{x}_{K,t}\right) \\ & + \tfrac{f_{k,0}+f_{k,t}}{s_{k,0}+f_{k,0}+s_{k,t}+f_{k,t}} \cdot V_D^*\left(\mathbf{x}_{1,t}, \mathbf{x}_{k,t}+\mathbf{e}_2, \ldots, \mathbf{x}_{K,t}\right)\Big)\Big\} \end{aligned}$$

The optimality is ensured by the Gittins index theorem, we have the passive aciton $a_{k,t}$, ensure that

$$\begin{aligned} P_k\left(x_k' | x_k, 0\right) &= P_k\left\{X_{k,t+1} = x_k' | X_{k,t} = x_k, a_{k,t} = 0\right\} \\ &= 1_{\{x_{k'} = x_k\}} \end{aligned}$$

for any $x_k, x_k' \in \mathbb{X}_k$, where $1_{\{x_\mu = x_k\}}$ is an indicator variable for the event that the state variable value at time $t+1 : x_{k'}$ equals the state variable value of state $t : x_k$, and (4) the set of feasible polices Π contains all polices π such that for all

$$\sum_{k=1}^{K} a_{k,t} \leq 1$$

then there exists a real-valued index function $G\left(x_{k,t}\right)$, which recovers the optimal solution to such a MABP when the objective function is defined under a ETD criterion, as in (2.3). Such a function is defined as follows:

$$G_k\left(x_{k,t}\right) = \sup_{\tau \geq 1} \frac{\mathrm{E}_{X_{k,t}=x_{k,t}} \sum_{i=0}^{\tau-1} R\left(X_{k,t+i}, 1\right) d^i}{\mathrm{E}_{X_{k,t}=x_{k,t}} \sum_{i=0}^{\tau-1} C\left(X_{k,t+i}, 1\right) d^i}$$

Such computational savings are particularly well illustrated in the Bayesian Bernoulli MABP where the Gittins index (3.4) is given by

$$G_k\left(\mathbf{x}_{k,t}\right) = \sup_{\tau \geq 1} \frac{\varepsilon \cdot \sum_{i=0}^{\tau-1} \frac{s_{k,0}+S_{k,t+i}}{s_{k,0}+f_{k,0}+S_{k,t+i}+F_{k,t+i}} d^i}{\mathrm{E} \cdot \sum_{i=0}^{\tau-1} d^i}$$

where $\mathrm{E}. = \mathrm{E}_{\mathrm{x}_{k,t}} = \left(s_{k,0} + s_{k,t} f_{k,0} + f_{k,t}\right)$ The Gittins index policy assigns a number to every treatment. based on the values of $s_{k,t}$ and $f_{k,t}$ observed, and then prioritizes sampling the one with the highest value. Thus, provided that we adjust for each treatment prior, the same table can be used for making the allocation decision of all treatments in a trial.

Homework5 for Reinforce-Learning SI-252

Yiwei Yang 2018533218

i. Environment setup

In [59]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
import cv2
from typing import Generic, Union, Sequence, Tuple,Callable,TypeVar,Mapping, Any
from scipy import linalg
import math
import random
from tabulate import tabulate
from collections import defaultdict
from gym import Env, spaces
from gym.utils import seeding
from gym.envs.toy_text.discrete import categorical_sample
```

ii. n-state simulation

In [6]:
```
class GridWorldMDP:

    # up, right, down, left
    _direction_deltas = [
        (-1, 0),
        (0, 1),
        (1, 0),
        (0, -1),
    ]
    _num_actions = len(_direction_deltas)

    def __init__(self,
                 reward_grid,
                 terminal_mask,
                 obstacle_mask,
                 action_probabilities,
                 no_action_probability):

        self._reward_grid = reward_grid
        self._terminal_mask = terminal_mask
        self._obstacle_mask = obstacle_mask
        self._T = self._create_transition_matrix(
            action_probabilities,
            no_action_probability,
            obstacle_mask
        )

    @property
    def shape(self):
        return self._reward_grid.shape

    @property
    def size(self):
        return self._reward_grid.size

    @property
    def reward_grid(self):
        return self._reward_grid

    def run_value_iterations(self, discount=1.0,
                             iterations=10):
        utility_grids, policy_grids = self._init_utility_policy_storage(iterations)

        utility_grid = np.zeros_like(self._reward_grid)
        for i in range(iterations):
            utility_grid = self._value_iteration(utility_grid=utility_grid)
            policy_grids[:, :, i] = self.best_policy(utility_grid)
            utility_grids[:, :, i] = utility_grid
        return policy_grids, utility_grids

    def run_value_iterations_with_c(self, discount=1.0,
                                    iterations=10,c=1):
        utility_grids, policy_grids = self._init_utility_policy_storage(iterations)

        policy_grid = np.random.randint(0, self._num_actions,
                                        self.shape)
        utility_grid = self._reward_grid.copy()

        for i in range(iterations):
            policy_grid, utility_grid = self._policy_iteration(
                policy_grid=policy_grid,
                utility_grid=utility_grid
            )
            policy_grids[:, :, i] = policy_grid
            utility_grids[:, :, i] = utility_grid
        return policy_grids, utility_grids

    def generate_experience(self, current_state_idx, action_idx):
        sr, sc = self.grid_indices_to_coordinates(current_state_idx)
        next_state_probs = self._T[sr, sc, action_idx, :, :].flatten()

        next_state_idx = np.random.choice(np.arange(next_state_probs.size),
                                          p=next_state_probs)

        return (next_state_idx,
                self._reward_grid.flatten()[next_state_idx],
                self._terminal_mask.flatten()[next_state_idx])

    def grid_indices_to_coordinates(self, indices=None):
        if indices is None:
            indices = np.arange(self.size)
        return np.unravel_index(indices, self.shape)

    def grid_coordinates_to_indices(self, coordinates=None):
        # Annoyingly, this doesn't work for negative indices.
        # The mode='wrap' parameter only works on positive indices.
        if coordinates is None:
            return np.arange(self.size)
        return np.ravel_multi_index(coordinates, self.shape)

    def best_policy(self, utility_grid):
        M, N = self.shape
        return np.argmax((utility_grid.reshape((1, 1, 1, M, N)) * self._T)
                         .sum(axis=-1).sum(axis=-1), axis=2)

    def _init_utility_policy_storage(self, depth):
        M, N = self.shape
        utility_grids = np.zeros((M, N, depth))
        policy_grids = np.zeros_like(utility_grids)
        return utility_grids, policy_grids

    def _create_transition_matrix(self,
                                  action_probabilities,
                                  no_action_probability,
                                  obstacle_mask):
        M, N = self.shape

        T = np.zeros((M, N, self._num_actions, M, N))

        r0, c0 = self.grid_indices_to_coordinates()

        T[r0, c0, :, r0, c0] += no_action_probability
```

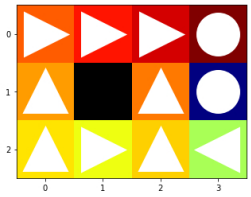
```
        for action in range(self._num_actions):
            for offset, P in action_probabilities:
                direction = (action + offset) % self._num_actions

                dr, dc = self._direction_deltas[direction]
                r1 = np.clip(r0 + dr, 0, M - 1)
                c1 = np.clip(c0 + dc, 0, N - 1)

                temp_mask = obstacle_mask[r1, c1].flatten()
                r1[temp_mask] = r0[temp_mask]
                c1[temp_mask] = c0[temp_mask]

                T[r0, c0, action, r1, c1] += P

        terminal_locs = np.where(self._terminal_mask.flatten())[0]
        T[r0[terminal_locs], c0[terminal_locs], :, :, :] = 0
        return T

    def _value_iteration(self, utility_grid, discount=1.0):
        out = np.zeros_like(utility_grid)
        M, N = self.shape
        for i in range(M):
            for j in range(N):
                out[i, j] = self._calculate_utility((i, j),
                                                    discount,
                                                    utility_grid)
        return out

    def _policy_iteration(self, *, utility_grid,
                          policy_grid, discount=1.0):
        r, c = self.grid_indices_to_coordinates()

        M, N = self.shape

        utility_grid = (
            self._reward_grid +
            discount * ((utility_grid.reshape((1, 1, 1, M, N)) * self._T)
                        .sum(axis=-1).sum(axis=-1))[r, c, policy_grid.flatten()]
            .reshape(self.shape)
        )

        utility_grid[self._terminal_mask] = self._reward_grid[self._terminal_mask]

        return self.best_policy(utility_grid), utility_grid

    def _calculate_utility(self, loc, discount, utility_grid):
        if self._terminal_mask[loc]:
            return self._reward_grid[loc]
        row, col = loc
        return np.max(
            discount * np.sum(
                np.sum(self._T[row, col, :, :, :] * utility_grid,
                       axis=-1),
                axis=-1)
        ) + self._reward_grid[loc]

    def plot_policy(self, utility_grid, policy_grid=None):
        if policy_grid is None:
            policy_grid = self.best_policy(utility_grid)
        markers = "^>v<"
        marker_size = 200 // np.max(policy_grid.shape)
        marker_edge_width = marker_size // 10
        marker_fill_color = 'w'

        no_action_mask = self._terminal_mask | self._obstacle_mask

        utility_normalized = (utility_grid - utility_grid.min()) / \
                             (utility_grid.max() - utility_grid.min())

        utility_normalized = (255*utility_normalized).astype(np.uint8)

        utility_rgb = cv2.applyColorMap(utility_normalized, cv2.COLORMAP_JET)
        for i in range(3):
            channel = utility_rgb[:, :, i]
            channel[self._obstacle_mask] = 0

        plt.imshow(utility_rgb[:, :, ::-1], interpolation='none')

        for i, marker in enumerate(markers):
            y, x = np.where((policy_grid == i) & np.logical_not(no_action_mask))
            plt.plot(x, y, marker, ms=marker_size, mew=marker_edge_width,
                     color=marker_fill_color)

        y, x = np.where(self._terminal_mask)
        plt.plot(x, y, 'o', ms=marker_size, mew=marker_edge_width,
                 color=marker_fill_color)

        tick_step_options = np.array([1, 2, 5, 10, 20, 50, 100])
        tick_step = np.max(policy_grid.shape)/8
        best_option = np.argmin(np.abs(np.log(tick_step) - np.log(tick_step_options)))
        tick_step = tick_step_options[best_option]
        plt.xticks(np.arange(0, policy_grid.shape[1] - 0.5, tick_step))
        plt.yticks(np.arange(0, policy_grid.shape[0] - 0.5, tick_step))
        plt.xlim([-0.5, policy_grid.shape[0]-0.5])
        plt.xlim([-0.5, policy_grid.shape[1]-0.5])
```

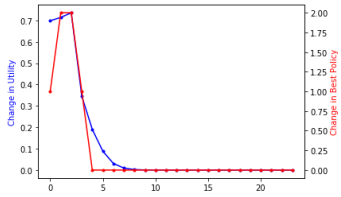
In [7]:
```
def plot_convergence(utility_grids, policy_grids):
    fig, ax1 = plt.subplots()
    ax2 = ax1.twinx()
    utility_ssd = np.sum(np.square(np.diff(utility_grids)), axis=(0, 1))
    ax1.plot(utility_ssd, 'b.-')
    ax1.set_ylabel('Change in Utility', color='b')

    policy_changes = np.count_nonzero(np.diff(policy_grids), axis=(0, 1))
    ax2.plot(policy_changes, 'r.-')
    ax2.set_ylabel('Change in Best Policy', color='r')
```

In [8]:
```
shape = (3, 4)
goal = (0, -1)
trap = (1, -1)
obstacle = (1, 1)
start = (2, 0)
default_reward = -0.1
goal_reward = 1
trap_reward = -1
reward_grid = np.zeros(shape) + default_reward
reward_grid[goal] = goal_reward
reward_grid[trap] = trap_reward
reward_grid[obstacle] = 0
terminal_mask = np.zeros_like(reward_grid, dtype=np.bool)
terminal_mask[goal] = True
terminal_mask[trap] = True
obstacle_mask = np.zeros_like(reward_grid, dtype=np.bool)
obstacle_mask[1, 1] = True
gw = GridWorldMDP(reward_grid=reward_grid,
                  obstacle_mask=obstacle_mask,
                  terminal_mask=terminal_mask,
                  action_probabilities=[
                      (-1, 0.1),
                      (0, 0.8),
                      (1, 0.1),
                  ],
                  no_action_probability=0.0)
mdp_solvers = {'Value Iteration': gw.run_value_iterations,
               'Value Iteration with c': gw.run_value_iterations_with_c}
for solver_name, solver_fn in mdp_solvers.items():
    print('Final result of {}:'.format(solver_name))
    policy_grids, utility_grids = solver_fn(iterations=25, discount=0.5)
    print(policy_grids[:, :, -1])
    print(utility_grids[:, :, -1])
    plt.figure()
    gw.plot_policy(utility_grids[:, :, -1])
    plot_convergence(utility_grids, policy_grids)
    plt.show()
```

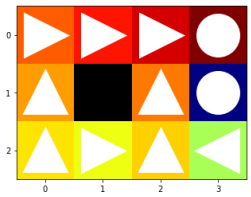
```
Final result of Value Iteration:
[[1 1 1 0]
```

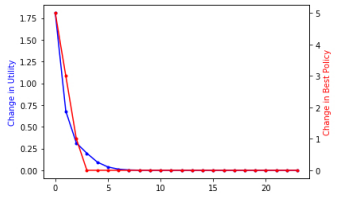
```
 [0. 0. 0. 0.]
 [0. 1. 0. 3.]]
[[ 0.56999141  0.71061644  0.83561644  1.        ]
 [ 0.44499134  0.66504707  0.52054795 -1.        ]
 [ 0.30913886  0.22232089  0.34732113  0.08650751]]
```






```
Final result of Value Iteration with c:
[[1. 1. 1. 0.]
 [0. 0. 0. 0.]
 [0. 1. 0. 3.]]
[[ 0.56999142  0.71061644  0.83561644  1.        ]
 [ 0.44499138  0.66504707  0.52054795 -1.        ]
 [ 0.30913898  0.22232101  0.34732117  0.0865076 ]]
```





iii. Student MRP & MDP

In [20]:
```python
def get_all_states(state_transition_matrix: Mapping[S, Any]) -> Sequence[S]:
    state_list = [state for state in state_transition_matrix.keys()]
    return state_list
```

In [60]:
```python
class MP(Generic[S]):
    def __init__(self, state_transition_matrix: SSf) -> None:
        try:
            valid: bool = self.check_if_valid(state_transition_matrix)
            if not valid:
                raise ValueError()
        except ValueError:
            exit('Input is not a valid Markov Process')

        self.all_states: Sequence[S] = get_all_states(state_transition_matrix)
        self.state_transition_matrix: SSf = state_transition_matrix

    def check_if_valid(self, state_transition_matrix: SSf) -> bool:
        for s1 in state_transition_matrix:
            if math.fsum(state_transition_matrix[s1].values()) != 1.0:
                return False
            for s2 in state_transition_matrix[s1].keys():
                if state_transition_matrix.get(s1).get(s2) < 0 or state_transition_matrix.get(s1).get(s2) > 1:
                    return False
        return True

    def get_stationary_distribution(self) -> Mapping[S, float]:
        sz = len(self.all_states)
        P = np.zeros((sz, sz), dtype=float)
        # construct state transition matrix as a 2-d np array
        for i, s1 in enumerate(self.all_states):
            for j, s2 in enumerate(self.all_states):
                if self.state_transition_matrix.get(s1) is not None:
                    if self.state_transition_matrix[s1].get(s2) is not None:
                        P[i][j] = self.state_transition_matrix[s1][s2]
        # find a row vector v s.t. v * P = v && sum(v) == 1 by solving an overdetermined linear equation
        # ref: https://stephens999.github.io/fiveMinuteStats/stationary_distribution.html
        a = np.concatenate((np.transpose(P) - np.identity(sz), np.ones((1, sz))), axis=0)
        b = np.concatenate((np.zeros((sz, 1)), np.ones((1, 1))), axis=0)
        x = linalg.lstsq(a, b)[0]
        return {s: x[i][0].astype(float) for i, s in enumerate(self.all_states)}
class DiscreteLimitActionsEnv(Env):
    def __init__(self, nS, vA, P, isd):
        self.P = P
        self.isd = isd
        self.lastaction = None # for rendering
        self.nS = nS
        self.vA = np.array(vA)

        assert (self.vA >= 0).all(), "Number of actions per state must be nonnegative."
        self.observation_space = spaces.Discrete(self.nS)
        self.action_space = spaces.Tuple(tuple(spaces.Discrete(nA) for nA in self.vA))

        self.seed()
        self.s = categorical_sample(self.isd, self.np_random)

    def seed(self, seed=None):
        self.np_random, seed = seeding.np_random(seed)
        return [seed]

    def reset(self):
        self.s = categorical_sample(self.isd, self.np_random)
        self.lastaction = None
        return self.s

    def step(self, a):
        if not self.action_space.spaces[self.s].contains(a):
            raise ValueError(
                f"Action must be < {self.action_space.spaces[self.s].n} in space {self.s}, attempted {a}"
```

```
            )
        transitions = self.P[self.s][a]
        i = categorical_sample([t[0] for t in transitions], self.np_random)
        p, s, r, d= transitions[i]
        self.lastaction = (self.s, a)
        self.s = s
        return (s, r, d, {"prob" : p})

class StudentEnv(DiscreteLimitActionsEnv):
    def __init__(self):
        # states / observations
        FACEBOOK = 0
        CLASS1 = 1
        CLASS2 = 2
        CLASS3 = 3
        SLEEP = 4  # terminal state
        observations = [FACEBOOK, CLASS1, CLASS2, CLASS3, SLEEP]

        nS = len(observations)

        # initial state distribution (uniform)
        isd = np.ones(nS) / nS

        # P is a dict of dict of lists, where
        #   P[s][a] == [(probability, nextstate, reward, done), ...]
        P = {}
        for s in observations:
            P[s] = {}

        P[FACEBOOK][0] = [(1, FACEBOOK, -1, False)]
        P[FACEBOOK][1] = [(1, CLASS1, 0, False)]
        P[CLASS1][0] = [(1, FACEBOOK, -1, False)]
        P[CLASS1][1] = [(1, CLASS2, -2, False)]
        P[CLASS2][0] = [(1, SLEEP, 0, True)]
        P[CLASS2][1] = [(1, CLASS3, -2, False)]
        P[CLASS3][0] = [(0.2, CLASS1, 1, False),
                        (0.4, CLASS2, 1, False),
                        (0.4, CLASS3, 1, False)]
        P[CLASS3][1] = [(1, SLEEP, 10, True)]
        P[SLEEP][0] = [(1, SLEEP, 0, True)]

        vA = []
        for s in observations:
            vA.append(len(P[s]))

        super().__init__(nS, vA, P, isd)
```

In [22]:

```
class MRP(MP):
    def __init__(self, mrp_input: Union[SSTff, STSff], discount_factor: float) -> None:
        # TODO: inherit check_if_valid from MP
        self.type_indicator: bool = MRP.input_type(mrp_input)
        self.gamma: float = discount_factor
        self.terminal_states, self.non_terminal_states = self._categorize_states(mrp_input)
        self.state_transition_matrix, self.reward_matrix = self._assign_transition_matrix(
            mrp_input)
        self.reward_function: Rf = self._assign_reward_function()
        self.value_function: Vf = self._assign_value_function()

    # return True if SSTff, False if STSff
    @staticmethod
    def input_type(mrp_input: Union[SSTff, STSff]) -> bool:
        first_value = mrp_input.get(next(iter(mrp_input)))
        return type(first_value) is dict

    def _categorize_states(self, mrp_input: Union[SSTff, STSff]) -> Tuple[Sequence[S], Sequence[S]]:
        # list of all states
        all_states = [state for state in mrp_input.keys()]
        # list of terminal states
        terminal_states = []
        non_terminal_states = []
        for s in all_states:
            if self.type_indicator is True:
                if mrp_input.get(s).get(s) is not None and mrp_input.get(s).get(s)[0] == 1:
                    terminal_states.append(s)
                else:
                    non_terminal_states.append(s)
            else:
                if mrp_input.get(s)[0].get(s, 0) == 1:
                    terminal_states.append(s)
                else:
                    non_terminal_states.append(s)
        return terminal_states, non_terminal_states

    def _assign_transition_matrix(self, mrp_input: Union[SSTff, STSff]) -> Tuple[SSf, SSf]:
        # Note that both state_transition_matrix and reward_matrix should only include non-terminal states
        # otherwise assume s is terminal state, P_ss = 1, which means (I - gamma * P) will have a 0 eigenvalue
        # when gamma = 1, which makes the inversion method inapplicable
        state_transition_matrix = {}
        reward_matrix = {}
        if self.type_indicator is True:
            for s1 in self.non_terminal_states:
                state_value = {}
                reward_value = {}
                for s2 in mrp_input.get(s1):
                    state_value.update({s2: mrp_input.get(s1).get(s2)[0]})
                    reward_value.update({s2: mrp_input.get(s1).get(s2)[1]})
                state_transition_matrix.update({s1: state_value})
                reward_matrix.update({s1: reward_value})
        else:
            for s1 in self.non_terminal_states:
                state_transition_matrix.update({s1: mrp_input.get(s1)[0]})
                reward_value = {}
                for s2 in mrp_input:
                    reward_value.update({s2: mrp_input.get(s1)[1]})
                reward_matrix.update({s1: reward_value})
        return state_transition_matrix, reward_matrix

    def _assign_reward_function(self) -> Rf:
        reward_list = {}
        for current_state in self.reward_matrix:
            expected_reward = 0
            for next_state in self.state_transition_matrix.get(current_state):
                expected_reward += self.state_transition_matrix.get(current_state).get(next_state, 0.0
                                                                                       ) * self.reward_matrix.get(
                    current_state).get(next_state, 0.0)
            reward_list.update({current_state: round(expected_reward, 5)})
        return reward_list

    def _assign_value_function(self) -> Vf:
        # V = R + gamma * P * V
        # convert R to np array
        R = np.array([reward for reward in self.reward_function.values()])
        print("R:", R)
        print("state list: ", self.non_terminal_states)
        # convert P to np array
        sz = len(self.non_terminal_states)
        P = np.empty([sz, sz])
        for index1 in range(sz):
            for index2 in range(sz):
                P[index1, index2] = self.state_transition_matrix.get(self.non_terminal_states[index1]).get(
                    self.non_terminal_states[
                        index2], 0.0)
        # calculate value using matrix inversion
        # https://stackoverflow.com/questions/9155478/how-to-try-except-an-illegal-matrix-operation-due-to-singularity-in-numpy
        print("P: ", P)
        print("A: ", np.eye(sz) - self.gamma * P)
        try:
            V = np.linalg.solve(np.eye(sz) - self.gamma * P, R)
        except np.linalg.LinAlgError as err:
            if 'Singular matrix' in str(err):
                print("matrix not invertible, will use least square, but most likely result may be incorrect")
                V = np.linalg.lstsq(np.eye(sz) - self.gamma * P, R, rcond=None)[0]
            else:
                raise
        # convert V from np array to dict
        vf = {state: float(value) for state, value in zip(self.non_terminal_states, np.nditer(V))}
        return vf
```

```
    # R(s)
    def get_expected_reward(self, state: S) -> float:
        return self.reward_function.get(state)

    # r(s, s')
    def get_state_transition_reward(self, current_state: S, next_state: S) -> float:
        return self.reward_matrix.get(current_state).get(next_state, 0.0)

    # v(s)
    def get_value(self, state: S) -> float:
        return self.value_function.get(state)

    def get_random_sample(self, initial_state: S = None) -> Tuple[Sequence[S], float]:
        g = 0
        sample = []
        ts = 0
        state = initial_state if initial_state is not None else random.choice(self.non_terminal_states +
                                                                              self.terminal_states)
        while state in self.non_terminal_states:
            sample.append(state)
            g += self.gamma ** ts * self.get_expected_reward(state)
            ts += 1
            next_state_lst = [s for s in self.state_transition_matrix.get(state).keys()]
            print(next_state_lst)
            pr_list = [pr for pr in self.state_transition_matrix.get(state).values()]
            print(pr_list)
            state = np.random.choice(next_state_lst, p=pr_list)
        sample.append(state)
        return sample, g
```

In [66]:
```
def policy_eval(policy, env, discount_factor=1.0, theta=0.00001):
    V = np.zeros(env.nS)
    while True:
        delta = 0
        # For each state, perform a "full backup"
        for s in range(env.nS):
            v = 0
            # Look at the possible next actions
            for a, action_prob in enumerate(policy[s]):
                # For each action, look at the possible next states...
                for  prob, next_state, reward, done in env.P[s][a]:
                    # Calculate the expected value. Ref: Sutton book eq. 4.6.
                    v += action_prob * prob * (reward + discount_factor * V[next_state])
            # How much our value function changed (across any states)
            delta = max(delta, np.abs(v - V[s]))
            V[s] = v
        # Stop evaluating once our value function change is below a threshold
        if delta < theta:
            break
    return np.array(V)
# mapping from integer to state names
obs = {0: 'FACEBOOK', 1: 'CLASS1', 2: 'CLASS2', 3: 'CLASS3', 4: 'SLEEP'}
random_policy = dict()
for s, actions in actions_for_obs.items():
    n_actions = len(actions)
    random_policy[s] = np.ones(n_actions) / n_actions
    # equivalent to pi(a|s)=0.5 for all states except the terminal sleep state

# allowed actions per state
actions_for_obs = {
    0: {0: 'facebook', 1: 'quit'},
    1: {0: 'facebook', 1: 'study'},
    2: {0: 'sleep', 1: 'study'},
    3: {0: 'pub', 1: 'study'},
    4: {0: 'sleep'}
}
env = StudentEnv()
value_fxn = policy_eval(random_policy, env)
def student_MDP_Policy():
    for s, value in enumerate(value_fxn):
        print(f"optimal state-value in state {obs[s]}: ", round(value,1))
def value_iteration(env, theta=0.0001, discount_factor=1.0):
    def one_step_lookahead(state, V):
        A = np.zeros(env.vA[state])
        for a in range(env.vA[state]):
            for prob, next_state, reward, _ in env.P[state][a]:
                A[a] += prob * (reward + discount_factor * V[next_state])
        return A

    V = np.zeros(env.nS)
    while True:
        # Stopping condition
        delta = 0
        # Update each state...
        for s in range(env.nS):
            # Do a one-step lookahead to find the best action
            A = one_step_lookahead(s, V)
            best_action_value = np.max(A)
            # Calculate delta across all states seen so far
            delta = max(delta, np.abs(best_action_value - V[s]))
            # Update the value function. Ref: Sutton book eq. 4.10.
            V[s] = best_action_value
        # Check if we can stop
        if delta < theta:
            break

    # Create a deterministic policy using the optimal value function
    policy = [np.zeros(nA) for nA in env.vA]
    for s in range(env.nS):
        # One step lookahead to find the best action for this state
        A = one_step_lookahead(s, V)
        best_action = np.argmax(A)
        # Always take the best action
        policy[s][best_action] = 1.0
    return policy, V
def student_MDP_Value():
    optimal_policy, optimal_value_fxn = value_iteration(env)
    print(optimal_policy)
    print(optimal_value_fxn)

    for s, actions in enumerate(optimal_policy):
        print(f"In state {obs[s]}")
        print(f"optimal state-value: ", optimal_value_fxn[s])
        print(f"action for optimal policy: ", actions_for_obs[s][np.argmax(actions)], '\n')
```

In [23]:
```
S = TypeVar('S')  # state
A = TypeVar('A')  # action
SSf = Mapping[S, Mapping[S, float]]  # state transition
SSTff = Mapping[S, Mapping[S, Tuple[float, float]]]  # state transition + r(s,s') reward
STSff = Mapping[S, Tuple[Mapping[S, float], float]]  # state transition + R(s) reward
# Rf = Mapping[Union[S, Tuple[S, A]], float]  # reward function
Rf = Mapping[Tuple[S, A], float]  # reward function
Vf = Mapping[S, float]  # state value function
Qf = Mapping[Tuple[S, A], float]  # action value function
Pf = Mapping[S, A]  # (deterministic) policy
SAf = Mapping[S, Mapping[A, float]]  # policy
SASf = Mapping[Tuple[S, A, S], Mapping[S, float]]  # state-action transition
SASTff = Mapping[Tuple[S, A], Mapping[S, Tuple[float, float]]]  # state-action transition + r(s,a,s') reward
SATSff = Mapping[Tuple[S, A], Tuple[Mapping[S, float], float]]  # state-action transition + R(s,a) reward
```

MRP

In [46]:
```
student_mrp = {
        'Facebook': ({'Facebook': 0.9, 'Class 1': 0.1}, -1),
        'Class 1': ({'Facebook': 0.5, 'Class 2': 0.5}, -2),
        'Class 2': ({'Sleep': 0.2, 'Class 3': 0.8}, -2),
        'Class 3': ({'Pass': 0.6, 'Pub': 0.4}, -2),
        'Pass': ({'Sleep': 1.0}, 10),
        'Pub': ({'Class 1': 0.2, 'Class 2': 0.4, 'Class 3': 0.4}, 1),
        'Sleep': ({'Sleep': 1.0}, 0)
}
```

In [47]:
```
mrp_obj = MRP(student_mrp, 1.0)
print(mrp_obj.state_transition_matrix)
print(mrp_obj.reward_matrix)
```

```
print(mrp_obj.reward_function)
print(mrp_obj.value_function)
```

```
R: [-1. -2. -2. -2. 10.  1.]
state list:  ['Facebook', 'Class 1', 'Class 2', 'Class 3', 'Pass', 'Pub']
P:  [[0.9 0.1 0.  0.  0.  0. ]
 [0.5 0.  0.5 0.  0.  0. ]
 [0.  0.  0.  0.8 0.  0. ]
 [0.  0.  0.  0.  0.6 0.4]
 [0.  0.  0.  0.  0.  0. ]
 [0.  0.2 0.4 0.4 0.  0. ]]
A:  [[ 0.1 -0.1  0.   0.   0.   0. ]
 [-0.5  1.  -0.5  0.   0.   0. ]
 [ 0.   0.   1.  -0.8  0.   0. ]
 [ 0.   0.   0.   1.  -0.6 -0.4]
 [ 0.   0.   0.   0.   1.   0. ]
 [ 0.  -0.2 -0.4 -0.4  0.   1. ]]
{'Facebook': {'Facebook': 0.9, 'Class 1': 0.1}, 'Class 1': {'Facebook': 0.5, 'Class 2': 0.5}, 'Class 2': {'Sleep': 0.2, 'Class
3': 0.8}, 'Class 3': {'Pass': 0.6, 'Pub': 0.4}, 'Pass': {'Sleep': 1.0}, 'Pub': {'Class 1': 0.2, 'Class 2': 0.4, 'Class 3': 0.
4}}
{'Facebook': {'Facebook': -1, 'Class 1': -1, 'Class 2': -1, 'Class 3': -1, 'Pass': -1, 'Pub': -1, 'Sleep': -1}, 'Class 1': {'Fa
cebook': -2, 'Class 1': -2, 'Class 2': -2, 'Class 3': -2, 'Pass': -2, 'Pub': -2, 'Sleep': -2}, 'Class 2': {'Facebook': -2, 'Cla
ss 1': -2, 'Class 2': -2, 'Class 3': -2, 'Pass': -2, 'Pub': -2, 'Sleep': -2}, 'Class 3': {'Facebook': -2, 'Class 1': -2, 'Class
2': -2, 'Class 3': -2, 'Pass': -2, 'Pub': -2, 'Sleep': -2}, 'Pass': {'Facebook': 10, 'Class 1': 10, 'Class 2': 10, 'Class 3': 1
0, 'Pass': 10, 'Pub': 10, 'Sleep': 10}, 'Pub': {'Facebook': 1, 'Class 1': 1, 'Class 2': 1, 'Class 3': 1, 'Pass': 1, 'Pub': 1,
'Sleep': 1}}
{'Facebook': -1.0, 'Class 1': -2.0, 'Class 2': -2.0, 'Class 3': -2.0, 'Pass': 10.0, 'Pub': 1.0}
{'Facebook': -22.543209876543223, 'Class 1': -12.543209876543218, 'Class 2': 1.4567901234567893, 'Class 3': 4.320987654320986,
'Pass': 10.0, 'Pub': 0.8024691358024668}
```

MDP

In [64]:
```
student_MDP_Policy()
```

```
optimal state-value in state FACEBOOK:  -2.3
optimal state-value in state CLASS1:  -1.3
optimal state-value in state CLASS2:  2.7
optimal state-value in state CLASS3:  7.4
optimal state-value in state SLEEP:  0.0
```

In [67]:
```
student_MDP_Value()
```

```
[array([0., 1.]), array([0., 1.]), array([0., 1.]), array([0., 1.]), array([1.])]
[ 6.  6.  8. 10.  0.]
In state FACEBOOK
optimal state-value:  6.0
action for optimal policy:  quit

In state CLASS1
optimal state-value:  6.0
action for optimal policy:  study

In state CLASS2
optimal state-value:  8.0
action for optimal policy:  study

In state CLASS3
optimal state-value:  10.0
action for optimal policy:  study

In state SLEEP
optimal state-value:  0.0
action for optimal policy:  sleep
```

iv. 5x5 grid

In [68]:
```
class BaseGridworld:
    def __init__(self, width, height, start_state=None, goal_state=None, terminal_states=[], blocked_states=[]):
        self.width = width
        self.height = height
        self.start_state = start_state
        self.goal_state = goal_state
        self.terminal_states = terminal_states
        self.blocked_states = blocked_states
        self.reset_state()

    def get_possible_actions(self, state):
        all_actions = [(0,1), (-1,0), (0,-1), (1,0)]
        return all_actions

    def get_states(self):
        return [(x,y) for x in range(self.width) for y in range(self.height)]

    def get_reward(self, state, action, next_state):
        raise NotImplementedError

    def get_state_reward_transition(self, state, action):
        # perform action
        next_state = np.array(state) + np.array(action)
        # clip to grid in case action resulted in off-the-grid state
        next_state = self._clip_state_to_grid(next_state)
        # return to old state in case action resulted in moving to a blocked state
        if self.is_blocked(next_state): # State is blocked; check action selection.'
            next_state = state

        # make into tuple of ints
        next_state = int(next_state[0]), int(next_state[1])

        # get reward
        reward = self.get_reward(state, action, next_state)

        return next_state, reward

    def _clip_state_to_grid(self, state):
        x, y = state
        return np.clip(x, 0, self.width-1), np.clip(y, 0, self.height-1)

    def is_goal(self, state):
        return tuple(state) == self.goal_state

    def is_terminal(self, state):
        return tuple(state) in self.terminal_states

    def is_blocked(self, state):
        return tuple(state) in self.blocked_states

    def reset_state(self):
        self.state = self.start_state
        return self.state


def action_to_nwse(action):
    x, y = action
    ret = ''
    if y == +1: ret += 'n'
    if y == -1: ret += 's'
    if x == +1: ret += 'e'
    if x == -1: ret += 'w'
    return ret
```

In [73]:
```
class Gridworld(BaseGridworld):
    def get_reward(self, state, action, next_state):
        if state == next_state:  # ie going off grid results in return to the same state
            return -1
        if self._is_special(state)[0]:
            return self._is_special(state)[1][1]
        return 0

    def get_state_reward_transition(self, state, action):
        if self._is_special(state)[0]:
            return self._is_special(state)[1]
```

```
        else:
            return super().get_state_reward_transition(state, action)

    def _is_special(self, state):
        A = (1, 4)
        A_prime = (1, 0)
        B = (3, 4)
        B_prime = (3, 2)

        if state == A:
            return True, (A_prime, 10)
        if state == B:
            return True, (B_prime, 5)
        return False, (None, None)



class UniformPolicyAgent:
    def __init__(self, mdp, discount=0.9, eps=1e-2, n_iterations=1000):
        self.mdp = mdp
        self.discount = discount

        # initialize values
        self.values = np.zeros((self.mdp.width, self.mdp.height))
        self.policy = {}

        # Iterative policy evaluation algorithm (Ch 4, p 59)
        for i in range(n_iterations):
            new_values = np.zeros_like(self.values)

            for state in self.mdp.get_states():
                if state in self.mdp.terminal_states:
                    continue

                q_values = {}
                for action in self.mdp.get_possible_actions(state):
                    # uniform action probability
                    action_prob = 1/len(self.mdp.get_possible_actions(state))
                    # compute q_value and update value estimate
                    q_values[action] = self.compute_q_value(state, action)
                    new_values[state] += action_prob * q_values[action]  # Bellman equation (eq. 3.14)

            # if improvement less then eps (after at least 1 iteration), stop iteration
            if np.sum(np.abs(new_values - self.values)) < eps:
                break

            # update values with new_values for the next iteration loop
            self.values = new_values

            # record optimal policy
            self.policy = self.update_policy()

    def compute_q_value(self, state, action):
        # get next state and reward from the transition model
        next_state, reward = self.mdp.get_state_reward_transition(state, action)
        return reward + self.discount * self.values[next_state]

    def update_policy(self):
        policy = {}
        for state in self.mdp.get_states():
            if state in self.mdp.terminal_states:
                continue
            q_values = {}
            for action in self.mdp.get_possible_actions(state):
                q_values[action] = self.compute_q_value(state, action)
            policy[state] = [a for a, v in q_values.items() if round(v, 5) == round(max(q_values.values()), 5)]
        return policy


class OptimalValueAgent:
    def __init__(self, mdp, discount=0.9, eps=1e-2, n_iterations=1000):
        self.mdp = mdp

        # initialize values
        self.values = np.zeros((self.mdp.width, self.mdp.height))  # eq 3.
        self.policy = {}

        # Iterative policy evaluation algorithm (Ch 4, p 59)
        for i in range(n_iterations):
            new_values = np.zeros_like(self.values)

            for state in self.mdp.get_states():
                # if terminal state, nothing to recurse down
                if state in self.mdp.terminal_states:
                    continue

                # if not terminal state, use Bellman eq to recurse value calculation
                q_values = {}
                for action in self.mdp.get_possible_actions(state):
                    # uniform action probability
                    action_prob = 1/len(self.mdp.get_possible_actions(state))
                    # get next state and reward from the transition model
                    next_state, reward = self.mdp.get_state_reward_transition(state, action)
                    # compute q_value and update value estimate
                    q_values[action] = reward + discount * self.values[next_state]

                # record optimal value
                new_values[state] = max(q_values.values())  # Bellman optimality equation (eq. 3.19)

                # record optimal policy
                self.policy[state] = [a for a, v in q_values.items() if v == max(q_values.values())]

            # if improvement less then eps (after at least 1 iteration), stop iteration
            if np.sum(np.abs(new_values - self.values)) < eps:
                break

            # update values with new_values for the next iteration loop
            self.values = new_values


def fig_unif_random():
    mdp = Gridworld(width=5, height=5)
    agent = UniformPolicyAgent(mdp)

    print('fig_unif_random: State-value function (V) for uniform random policy. (V = ∑ action_prob * q_value)')
    print(tabulate(np.flipud(agent.values.T), tablefmt='grid'))



def fig_optimal():
    mdp = Gridworld(width=5, height=5)
    agent = OptimalValueAgent(mdp)

    print('fig_optimal: Optimal solutions to the gridworld example. (V = max (q_value))')
    print(tabulate(np.flipud(agent.values.T), tablefmt='grid'))  # transform coordinates so (0,0) is bottom left

    grid = [['' for x in range(mdp.width)] for y in range(mdp.height)]
    for (x,y), v in agent.policy.items():
        grid[y][x] = [action_to_nwse(v_i) for v_i in v]

    # invert vertical coordinate so (0,0) is bottom left of the displayed grid
    grid = grid[::-1]

    print('Optimal policy:')
    print(tabulate(grid, tablefmt='grid'))
```

In [71]:
```
fig_unif_random()
```

```
fig_unif_random: State-value function (V) for uniform random policy. (V = ∑ action_prob * q_value)
+-----------+----------+----------+----------+----------+
|  3.31311  |  8.79335 |  4.43144 |  5.32599 |  1.49562 |
+-----------+----------+----------+----------+----------+
|  1.52567  |  2.99631 |  2.25393 |  1.91118 | 0.550869 |
+-----------+----------+----------+----------+----------+
| 0.0548918 | 0.742147 | 0.676931 | 0.361836 | -0.399599 |
+-----------+----------+----------+----------+----------+
| -0.969506 | -0.431488 | -0.351013 | -0.581873 | -1.17943 |
+-----------+----------+----------+----------+----------+
```

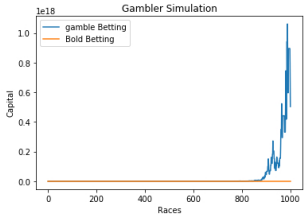
```
| -1.85359   | -1.3412   | -1.22535  | -1.41913  | -1.97146  |
+------------+-----------+-----------+-----------+-----------+
```

In [74]:
```
fig_optimal()
```

```
fig_optimal: Optimal solutions to the gridworld example. (V = max (q_value))
+---------+---------+---------+---------+---------+
| 21.9747 | 24.4163 | 21.9747 | 19.4163 | 17.4747 |
+---------+---------+---------+---------+---------+
| 19.7754 | 21.9747 | 19.7754 | 17.7979 | 16.0181 |
+---------+---------+---------+---------+---------+
| 17.7979 | 19.7754 | 17.7979 | 16.0181 | 14.4163 |
+---------+---------+---------+---------+---------+
| 16.0181 | 17.7979 | 16.0181 | 14.4163 | 12.9747 |
+---------+---------+---------+---------+---------+
| 14.4163 | 16.0181 | 14.4163 | 12.9747 | 11.6754 |
+---------+---------+---------+---------+---------+
Optimal policy:
+-----------+-------------------+-----------+-------------------+-----------+
| ['e']     | ['n', 'w', 's', 'e'] | ['w']     | ['n', 'w', 's', 'e'] | ['w']     |
+-----------+-------------------+-----------+-------------------+-----------+
| ['n', 'e'] | ['n']             | ['n', 'w'] | ['w']             | ['w']     |
+-----------+-------------------+-----------+-------------------+-----------+
| ['n', 'e'] | ['n']             | ['n', 'w'] | ['n', 'w']        | ['n', 'w'] |
+-----------+-------------------+-----------+-------------------+-----------+
| ['n', 'e'] | ['n']             | ['n', 'w'] | ['n', 'w']        | ['n', 'w'] |
+-----------+-------------------+-----------+-------------------+-----------+
| ['n', 'e'] | ['n']             | ['n', 'w'] | ['n', 'w']        | ['n', 'w'] |
+-----------+-------------------+-----------+-------------------+-----------+
```

v. Gambler

In [105]:
```
gamble_capital = intuitive_capital = 100
horses = ('A', 'B', 'C')
odds = (2, 3, 6)
gamble_bet = probabilities = (0.5, 0.25, 0.25)
intuitive_bet = (0.5, 0.17, 0.33)
gamble_plot = []
intuitive_plot = []

for i in range(0, 1000):
    # Determine winner of horse race randomly
    rand_num = random.random()
    winner = -1
    cumulative = 0
    for index, probability in enumerate(probabilities):
        cumulative += probability
        if rand_num < cumulative:
            winner = index
            break

    # Calculate capital after each race
    gamble_capital *= gamble_bet[winner] * odds[winner]
    intuitive_capital *= intuitive_bet[winner] * odds[winner]

    # Add point to the array for plotting of graph
    gamble_plot.append(gamble_capital)
    intuitive_plot.append(intuitive_capital)

x = [i for i in range(1, 1001)]
plt.plot(x, gamble_plot, label='gamble Betting')
plt.plot(x, intuitive_plot, label='Bold Betting')
plt.xlabel('Races')
plt.ylabel('Capital')
plt.title('Gambler Simulation')

plt.legend()
plt.show()
```



In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: