MS&E 310 Course Project Report:
Markov Decision Process

Jongho Kim
jkim22@stanford.edu

Hyunsik Kim
hsik@stanford.edu

Markove Decision Process (MDPs) provide a mathematical framework for modeling sequential decision-making in situations where outcomes are partly random and partly under the control of a decision maker. The MDP problem with m states and total n actions can be formulated as a standard form linear program with m equality constraints and n variables.

$$\begin{aligned} &\min_x \sum_{j\in\mathcal{A}_1} c_j x_j + \cdots + \sum_{j\in\mathcal{A}_m} c_j x_j \\ &\text{subject to} \sum_{j\in\mathcal{A}_1} (\mathbf{e}_1 - \gamma \mathbf{p}_j) x_j + \cdots + \sum_{j\in\mathcal{A}_m} (\mathbf{e}_m - \gamma \mathbf{p}_j) x_j = \mathbf{e} \\ &\qquad x_j \geq 0, \quad \forall j \end{aligned} \tag{1}$$

where $\mathcal{A}_i$ represents the set of all actions available in state i, $\mathbf{p}_j$ is the state transition probabilities from state i to all states and c_j is the immediate cost when action j is taken, and $0 < \gamma < 1$ is the discount factor. Also, $\mathbf{e} \in \mathbb{R}^m$ is the vector of ones, and $\mathbf{e}_i$ is the unit vector with 1 at the i-th position and zeros everywhere else. Variable x_j, $j \in \mathcal{A}_i$ is the state-aciton frequency or flux, or the expected present value of the number of times in which the process visits state i and takes state-action $j \in \mathcal{A}_i$.
This LP formulation can be *re-formulated* as following:

1. Let $P_{ik}(j)$ be the transition probabilities from state i to state k when action j taken. Also, let $x(i,j)$ be the state-action frequency of state i and action j and $c(i,j)$ be the immediate cost when action j is taken at state i. Then, the objective function can be re-formulated as following:

$$\text{minimize} \sum_{i=1}^{m} \sum_{j\in\mathcal{A}_i} c(i,j) x(i,j)$$

2. Using above notation, x_j and $\mathbf{p}_j$ has following formulation:

$$x_j = x(i,j) \quad \forall i \in \{1,\ldots,m\}, \qquad \mathbf{p}_j = \begin{bmatrix} p_{i1}(j) \\ p_{i2}(j) \\ \vdots \\ p_{im}(j) \end{bmatrix}$$

Then, $\mathbf{e}_i^T x_j = x(i,j)$ and $\mathbf{p}_j x_j$ is a vector multiplied by scalar. Then, the equality constraints can be formulated as following:

$$\begin{aligned}
&\sum_{j\in\mathcal{A}_1}(\mathbf{e}_1 - \gamma\mathbf{p}_j)x_j + \cdots + \sum_{j\in\mathcal{A}_m}(\mathbf{e}_m - \gamma\mathbf{p}_j)x_j = \mathbf{e} \\
&\Rightarrow \sum_{j\in\mathcal{A}_i} x(i,j) - \sum_{k=1}^{m}\gamma\sum_{j\in\mathcal{A}_k} p_{ki}(j)x(k,j) = 1 \quad \forall i \\
&\Rightarrow \sum_{j\in\mathcal{A}_i} x(i,j) = 1 + \sum_{k=1}^{m}\gamma\sum_{j\in\mathcal{A}_k} p_{ki}(j)x(k,j) \quad \forall i \\
&\Rightarrow \sum_{j\in\mathcal{A}_i} x(i,j) = 1 + \gamma\sum_{k=1}^{m}\sum_{j\in\mathcal{A}_k} p_{ki}(j)x(k,j) \quad \forall i
\end{aligned}$$

Reformulated MDP primal problem is:

$$\begin{aligned}
&\text{minimize } \sum_{i=1}^{m}\sum_{j\in\mathcal{A}_i} c(i,j)x(i,j) \\
&\text{subject to} \sum_{j\in\mathcal{A}_i} x(i,j) = 1 + \gamma\sum_{k=1}^{m}\sum_{j\in\mathcal{A}_k} p_{ki}(j)x(k,j) \quad \forall i \\
&\qquad\qquad x_j \geq 0, \quad \forall j
\end{aligned} \tag{2}$$

Question 1

1. Prove that in (2) every basic feasible solution represent a policy, *i.e.*, the basic variables have exactly one variable from each state i.
   [Answer]
   First, we show that a basic feasible solution (BFS) x corresponds to a stationary policy. Let B denote the matrix set of basic variables, then $|B| = m$. Suppose B does not contain any state-action pair for a certain state z. Then, $\sum_{j \in \mathcal{A}_z} x(z,j) = 0$. However,

$$\sum_{j \in \mathcal{A}_z} x(z,j) = 1 + \gamma \sum_{k=1}^{m} \sum_{j \in \mathcal{A}_k} p_{kz}(j) x(k,j) \neq 0$$

   Hence, B contains exactly one state-action pair for each state, and corresponds to a stationary policy of the discounted MDPs.

   Second, we show that a stationary policy π corresponds to a BFS. Suppose $\pi(i) = a_i$ for $i = 1, \ldots, m$ and let $B = \{(i, a_i) \mid i = 1, \ldots, m \;\; a_i \in \mathcal{A}_i\}$. Then, for (i, a_i), we have

$$x_j = x(i, a_i), \quad \mathbf{p}_j = \begin{bmatrix} p_{i1}(a_i) \\ p_{i2}(a_i) \\ \vdots \\ p_{im}(a_i) \end{bmatrix}$$

   From (1), the equality constraints can be written as the form $A_B x_B = e$ where $A_B = I - \gamma P_B$ and $P_B = p_{ji}(a_j)$. It can be observed that the diagonal entries of A_B are positive and the off diagonal entriees are non-positive. This implies that A_B is a full rank matrix and A_B is a basis. Hence, x_B and x_N where $x_B = A_B^{-1}\mathbf{e}$ and x_N is a basic solution.
   Then, we need to show that x is feasible (in other words, $x_B \geq 0$). Suppose converse is true. Then, $\{A_B x_B = \mathbf{e}, \;\; x_B \geq 0\}$ is infeasible. Applying the Farkas' lemma, then there exists y such that $y^T A_B \leq 0$ and $y^T \mathbf{e} > 0$. Suppose y_1 is the maximum entry in y. Then, $y^T \mathbf{e} > 1$ so $y_1 > 0$. Given $y^T A_B \leq 0$, then the first entry of $y^T A_B$, $(y^T A_B)_1$ also holds. (*i.e.*, $(y^T A_B)_1 \leq 0$).

$$\begin{aligned} 0 &\geq (y^T A_B)_1 \\ &= (y^T (I - \gamma P_B))1 \\ &= y_1 - \gamma y^T P_1 \quad P_1 \text{ is the first column of } P_B \\ &= y_1(1 - \gamma) \\ &> 0 \quad \text{(Contradiction)} \end{aligned}$$

   Therefore, x_B, x_N is feasible and therefore a stationary policy π corresponds to a BFS.

2. Prove each basic variable value is no less than 1 and the sum of all basic variable values is $\frac{m}{1-\gamma}$.
   [**Answer**] From (2), the equality constraints is following:

$$\sum_{j\in\mathcal{A}_i} x(i,j) = 1 + \gamma \sum_{k=1}^{m} \sum_{j\in\mathcal{A}_k} p_{ki}(j)x(k,j) \quad \forall i$$

   Sum up all equality constraints, then we get

$$\begin{aligned}
\mathbf{e}^T x &= m + \gamma \sum_{i=1}^{m} \sum_{k=1}^{m} \sum_{j\in\mathcal{A}_k} p_{ki}(j)x(k,j) \\
&= m + \gamma \sum_{k=1}^{m} \sum_{j\in\mathcal{A}_k} \sum_{i=1}^{m} p_{ki}(j)x(k,j) \\
&= m + \gamma \sum_{k=1}^{m} \sum_{j\in\mathcal{A}_k} \underbrace{\sum_{i=1}^{m} p_{ki}(j)}_{=1} x(k,j) \\
&= m + \gamma \sum_{k=1}^{m} \sum_{j\in\mathcal{A}_k} x(k,j) \\
&= m + \gamma \mathbf{e}^T x
\end{aligned}$$

   Hence, $\mathbf{e}^T x = \frac{m}{1-\gamma}$. Therefore, the sum of all basic variable values is $\frac{m}{1-\gamma}$.

   Let x^π be a basic feasible solution (BFS). Then,

$$\begin{aligned}
\sum_{j\in\mathcal{A}_i} x^\pi(i,j) &= 1 + \gamma \sum_{k=1}^{m} \sum_{j\in\mathcal{A}_k} p_{ki}(j)x^\pi(k,j) \quad \forall i \\
&\geq 1 \quad \forall i
\end{aligned}$$

   Therefore, each variable value is no less than 1.

Question 2

Prove $\left\|y^{k+1} - y^*\right\|_\infty \leq \gamma \left\|y^k - y^*\right\|_\infty \ \forall k$, where

$$y_i^* = \min_{j \in \mathcal{A}_i} \{c_j + \gamma p_j^T y^*\} \quad \forall i$$
$$y_i^{k+1} = \min_{j \in \mathcal{A}_i} \{c_j + \gamma p_j^T y^k\} \quad \forall i$$

[**Answer**]
First, we prove the following:

$$\left|\min_j f(j) - \min_j g(j)\right| \leq \max_j |f(j) - g(j)| \tag{3}$$
$$\text{where} \quad f(j) = c_j + \gamma \mathbf{p}_j^T y^k \text{ and } g(j) = c_j + \gamma \mathbf{p}_j^T y^*$$

Let $F = \text{argmin}_j f(j) = \text{argmin}_j \{c_j + \gamma \mathbf{p}_j^T y^k\}$ and $G = \text{argmin}_j g(j) = \text{argmin}_j \{c_j + \gamma \mathbf{p}_j^T y^*\}$. Then, clearly, $f(F) \leq f(G)$ and $g(G) \leq g(F)$. Therefore,

$$\begin{aligned} f(F) - g(G) &\leq f(G) - g(G) \\ &\leq |f(G) - g(G)| \\ &\leq \max_j |f(j) - g(j)| \end{aligned}$$

and similarly,

$$\begin{aligned} g(G) - f(F) &\leq g(F) - f(F) \\ &\leq |g(F) - f(F)| \\ &\leq \max_j |g(j) - f(j)| = \max_j |f(j) - g(j)| \end{aligned}$$

Therefore,

$$|f(F) - g(G)| = \left|\min_j f(j) - \min_j g(j)\right| \leq \max_j |f(j) - g(j)|$$

Then, we have following:

$$
\begin{aligned}
\left\|y^{k+1}-y^*\right\|_\infty &= \max_i \left|y_i^{k+1}-y_i^*\right| \\
&= \max_i \left|\min_{j\in\mathcal{A}_i}\{c_j+\gamma p_j^T y^k\} - \min_{j\in\mathcal{A}_i}\{c_j+\gamma p_j^T y^*\}\right| \\
&\le \max_i \max_{j\in\mathcal{A}_i} \left|(c_j+\gamma \mathbf{p}_j^T y^k) - (c_j+\gamma \mathbf{p}_j^T y^*)\right| \quad \text{(By (3))} \\
&= \max_i \max_{j\in\mathcal{A}_i} \left|\gamma \mathbf{p}_j^T (y^k-y^*)\right| \\
&= \max_i \max_{j\in\mathcal{A}_i} \left|\gamma \sum_{i'=1}^m p_{ii'}(j)(y_{i'}^k-y_{i'}^*)\right| \\
&\le \max_i \max_{j\in\mathcal{A}_i} \gamma \left|\sum_{i'=1}^m p_{ii'}(j)(y_{i'}^k-y_{i'}^*)\right| \\
&\le \max_i \max_{j\in\mathcal{A}_i} \gamma \underbrace{\sum_{i'=1}^m p_{ii'}(j)\left|(y_{i'}^k-y_{i'}^*)\right|}_{\text{expectation}} \quad \text{(Here, I used } |x+y|\le|x|+|y|\text{)} \\
&\le \max_i \max_{j\in\mathcal{A}_i} \gamma \max_{i'} \left|y_{i'}^k-y_{i'}^*\right| \\
&= \gamma \max_{i'} \left|y_{i'}^k-y_{i'}^*\right| \\
&= \gamma \left\|y^k-y^*\right\|_\infty
\end{aligned}
$$

Therefore, $\left\|y^{k+1}-y^*\right\|_\infty \le \gamma \left\|y^k-y^*\right\|_\infty \ \forall k$ is proved.

From office hour: SKIP Question 3

Algorithms

From **question 4 to 6**, we have following questions:

(i) What can you tell the convergence of the algorithm in this question?

(ii) Does it make a difference with the classical VI method?

(iii) If there is any sample size present, how is the sample size affect the performance?

(iv) Use simulated computational experiments to verify your claims.

Therefore, we present and briefly explain about each algorithm and answer above questions on **Experiment** section. For each algorithm, c denotes Immdeidate cost, p denotes State-transition probabilities, γ denotes Discount factor and ϵ denotes threshold for convergence. The output $\mathbf{y}^*$ denotes the optimal cost-to-go value for each state.

- `OriginalVI` algorithm
  Original version of VI method is following:

Algorithm 1: Original Value Iteration (OriginalVI)

Input : c, p, γ, ϵ
Output: $\mathbf{y}^*$

$\mathbf{y}^0 \longleftarrow$ `Initialize()`
$k = 0$
while ***True*** **do**
  for $i = 1, \ldots, m$ **do**
    $y_i^{k+1} = \min_{j \in \mathcal{A}_i}\{c_j + \gamma \mathbf{p}_j^T \mathbf{y}^k\}$
  if $\max_i\{\left|\mathbf{y}_i^{k+1} - \mathbf{y}_i^k\right|\} < \epsilon$ **then**
    return $\mathbf{y}^{k+1}$
  $k = k + 1$

- `RandomVI` on **question 4**
  Motivation: Rather than go through all state values in each iteration, in the kth iteration, randomly select a subset of states B^k and do

$$y_i^{k+1} = \min_{j \in \mathcal{A}_i} \{c_j + \gamma \mathbf{p}_j^T \mathbf{y}^k\} \quad \forall i \in B^k$$

  In RandomVI, we only update a subset of state values at random in each iteration.

Algorithm 2: Random Value Iteration (EM-RandomVI)

Input : c, p, γ, ϵ, α: state subset size
Output: $\mathbf{y}^*$

$k = 0$, $\mathbf{y}^0 \longleftarrow$ `Initialize()`
while ***True*** **do**
  $B^k \longleftarrow$ `Sample(`$[1, \ldots, m]$`, `α`)`
  for $i \in B^k$ **do**
    $y_i^{k+1} = \min_{j \in \mathcal{A}_i} \{c_j + \gamma \mathbf{p}_j^T \mathbf{y}^k\}$
    $B^k \longleftarrow B^k \setminus i$
  if $\max_i \{\left|\mathbf{y}_i^{k+1} - \mathbf{y}_i^k\right|\} < \epsilon$ **then**
    return $\mathbf{y}^{k+1}$
  $k = k + 1$

- `EM-RandomVI` on **question 4**
  Motivation: In this algorithm, we build an empirical distribution for each action being selected as the winning action in the final policy: the probability of action j is the past frequency of action j is being selected as the argmin in the previous iterations.

Algorithm 3: Empirical Random Value Iteration (EM-RandomVI)

Input : c, p, γ, ϵ, α: action subset size
Output: $\mathbf{y}^*$

$k = 0$, $\mathbf{y}^0 \longleftarrow$ `Initialize()`
while ***True*** **do**
  for $i = 1, \ldots, m$ **do**
    $\mathcal{A}_i =$ `weightedSample(`$\mathcal{A}_i, \alpha,$ `distribution` $= \mathcal{P}$`)`
    $y_i^{k+1} = \min_{j \in \mathcal{A}_i} \{c_j + \gamma \mathbf{p}_j^T \mathbf{y}^k\}$
    update $\mathcal{P}$
  if $\max_i \{\left|\mathbf{y}_i^{k+1} - \mathbf{y}_i^k\right|\} < \epsilon$ **then**
    return $\mathbf{y}^{k+1}$
  $k = k + 1$

$\mathcal{P}$ is empirical distribution for each action j based on by book-keeping previous frequency of action j being selected for each state.

- `CyclicVI` on **question 5**
  Motivation: In the `CyclicVI` method, as soon as a state value is updated, we use it to update the rest of state values.

Algorithm 4: Cyclic Value Iteration (CyclicVI)

Input : c, p, γ, ϵ
Output: $\mathbf{y}^*$

$k = 0$, $\mathbf{y}^0 \longleftarrow$ `Initialize()`
while ***True*** **do**
  $\tilde{\mathbf{y}}^k = \mathbf{y}^k$
  for $i = 1, \ldots, m$ **do**
    $\tilde{y}_i^k = \min_{j \in \mathcal{A}_i} \{c_j + \gamma \mathbf{p}_j^T \tilde{\mathbf{y}}^k\}$
  $\mathbf{y}^{k+1} = \tilde{\mathbf{y}}^k$
  if $\max_i \{\left|\mathbf{y}_i^{k+1} - \mathbf{y}_i^k\right|\} < \epsilon$ **then**
    return $\mathbf{y}^{k+1}$
  $k = k + 1$

- `RPCyclicVI` on **question 6**
  Motivation: In the `RPCyclicVI` method, rather than with the fixed cycle order from 1 to m, we follow a random permutation order, or sample without replacement to update the state values.

Algorithm 5: Randomly Permuted Cyclic Value Iteration (RPCyclicVI)

Input : c, p, γ, ϵ
Output: $\mathbf{y}^*$

$k = 0$, $\mathbf{y}^0 \longleftarrow$ `Initialize()`
while ***True*** **do**
  $\tilde{\mathbf{y}}^k = \mathbf{y}^k$
  $B^k \longleftarrow$ `Permutation([1,...,m])`
  for $i \in B^k$ **do**
    $\tilde{y}_i^k = \min_{j \in \mathcal{A}_i} \{c_j + \gamma \mathbf{p}_j^T \tilde{\mathbf{y}}^k\}$
    $B^k \longleftarrow B^k \setminus i$
  $\mathbf{y}^{k+1} = \tilde{\mathbf{y}}^k$
  if $\max_i \{\left|\mathbf{y}_i^{k+1} - \mathbf{y}_i^k\right|\} < \epsilon$ **then**
    return $\mathbf{y}^{k+1}$
  $k = k + 1$

Extension version

Here, we present **two** different version of VI methods that our team experimented.

- α-RandomVI
  In this algorithm, for each kth iteration, we randomly (uniformly) select a subset size, α. Then, we select subset of states B^k and perform RandomVI

Algorithm 6: Random Value Iteration with α-Subset Size (α-RandomVI)

Input : c, p, γ, ϵ
Output: $\mathbf{y}^*$

$k = 0$, $\mathbf{y}^0 \longleftarrow$ `Initialize()`
while ***True*** **do**
  $\alpha \longleftarrow$ `Uniform(1, m)`
  $B^k \longleftarrow$ `Sample(`$[1, \ldots, m]$`, ` α`)`
  for $i \in B^k$ **do**
    $y_i^{k+1} = \min_{j \in \mathcal{A}_i} \{c_j + \gamma \mathbf{p}_j^T \mathbf{y}^k\}$
    $B^k \longleftarrow B^k \setminus i$
  if $\max_i \{\left|\mathbf{y}_i^{k+1} - \mathbf{y}_i^k\right|\} < \epsilon$ **then**
    return $\mathbf{y}^{k+1}$
  $k = k + 1$

- α-RPCyclicVI
  In this algorithm, for each kth iteration, we randomly (uniformly) select a subset size, α. Then, we select subset of states B^k and perform RPCyclicVI

Algorithm 7: RPCyclicVI with α-Subset Size (α-RPCyclicVI)

Input : c, p, γ, ϵ
Output: $\mathbf{y}^*$

$k = 0$, $\mathbf{y}^0 \longleftarrow$ `Initialize()`
while ***True*** **do**
  $\tilde{\mathbf{y}}^k = \mathbf{y}^k$
  $\alpha \longleftarrow$ `Uniform(1, m)`
  $B^k \longleftarrow$ `Sample(`$[1, \ldots, m]$`, ` α`)`
  for $i \in B^k$ **do**
    $\tilde{y}_i^k = \min_{j \in \mathcal{A}_i} \{c_j + \gamma \mathbf{p}_j^T \tilde{\mathbf{y}}^k\}$
    $B^k \longleftarrow B^k \setminus i$
  $\mathbf{y}^{k+1} = \tilde{\mathbf{y}}^k$
  if $\max_i \{\left|\mathbf{y}_i^{k+1} - \mathbf{y}_i^k\right|\} < \epsilon$ **then**
    return $\mathbf{y}^{k+1}$
  $k = k + 1$

Experiments

1. MDPs in the small 2D Grid World

In this experiment, we present a small 2-dimensional (2D) grid world. The main motivation for this scenario experiment is to show that various algorithms in **algorithms** section provides correct convergence output, $\mathbf{y}^*$ and their policies. Below figure shows the state representations:

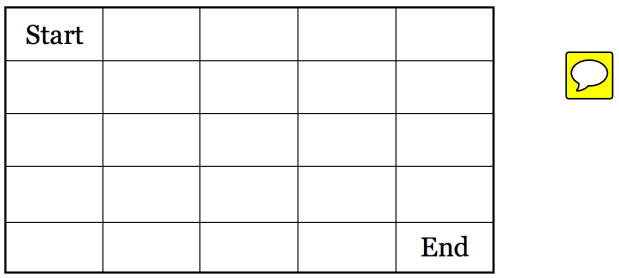

Figure 1: MDP grid world

MDP formulation:

- States, $s = (x, y)$ where x is x-coordinate and y is y-coordinate
- Start state, $s_{start} = (0, 0)$
- Actions at state s, $\mathcal{A}_s = (\texttt{Left}, \texttt{Right}, \texttt{Down}, \texttt{Up})$
  - Here, in some states, only certain actions are valid. For example, in s_{start}, valid actions are $\mathcal{A} = (\texttt{Right}, \texttt{Down})$
- End state, $s_{end} = (4, 4)$
- Transition Probaiblity
  From state s to state s' when action $a \in \mathcal{A}$ is taken is:

$$P(s, a, s') = \begin{cases} 1 & \text{if valid state} \\ 0 & \text{else} \end{cases}$$

  The state is *valid* if next state, s' is correct based on current state s and chosen action a. For example, if $s = s_{start}$ and $a = \texttt{down}$, then $P(s, a, s') = 1$ if $s' = (1, 0)$ and 0 otherwise.

- Cost, c_j for red, white and gray boxes
  If action j makes entering Red box: -100
  If action j makes entering White box : -30
  If action j makes entering Gray box : 100

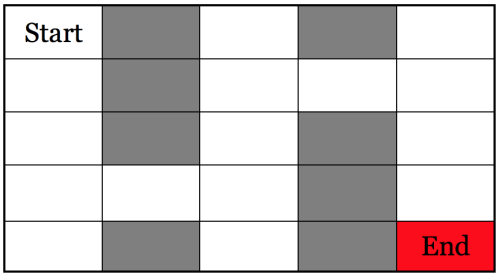

Figure 2: Cost of MDP grid world

We can interpret this scenario as obtaining a policy that makes a person to reach **end** state while maximizing the cost. He/she receives 30 dollars when in white box state and loses 100 dollars in gray boxes. If he reaches red box, the game ends and he/she receives the money based on visited states. The optimal policy we hope to obtain is following:

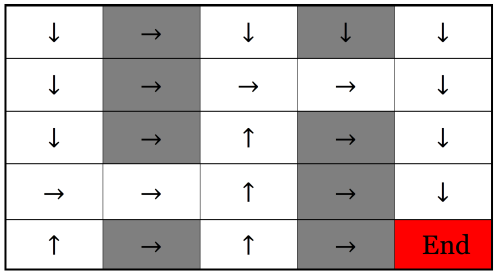

Figure 3: Target Policy

We performed four algorithms which corresponds to question 4 to 6 (`OriginalVI`,`RandomVI`, `CyclicVI`,`RPCyclicVI`) and obtained following results:

Algorithm	Number of Iterations	Execution Time (sec)
`OriginalVI`	14	0.015
`RandomVI`	24	0.075
`CyclicVI`	12	0.012
`RPCyclicVI`	9	0.009

Table 1: Result on small 2D small grid world

In this experiment, we omitted EM-RandomVI part because total number of actions is four and possible actions are depend on which state the player is in. Some states contains less than four actions. Therefore, keeping frequency of previous actions and select subset of actions is not much. We incorporated EM-RandomVI on the next experiment.

(i) **Convergence of each algorithms**
In this experiment, we could observe that each algorithm *converges very well*. Each algorithm provides $\mathbf{y}^*$ that we initially hoped as shown in Figure 3. The result on **terminal** can be found in Appendix B.

(ii) **Comparison on results**
OriginalVI took 0.015 seconds with 14 number of iterations. This classical method was not worst among all algorithms. The worst result was actually provided by RandomVI. It took the longest execution time (0.075 seconds) and also had the largest number of iterations ($k = 24$). The best performance was done by RPCyclicVI which took 0.009 seconds and the fewest number of iterations ($k = 9$).

(iii) **Difference with the classical VI method**
RandomVI uses randomly select a subset of states and apply update rule as folowing:

$$y_i^{k+1} = \min_{j \in \mathcal{A}_i} \{c_j + \gamma \mathbf{p}_j^T \mathbf{y}^k\} \quad \forall i \in B^k$$

In this setting, for each kth iteration, the number of updates is less than the total number of states (which is m) because we only select a subset of states and update on these subset state values. Therefore, each arithmatic (multiply, addition etc) operation in one iteration of k is actually cheaper than that of other algorithms'. Therefore, instead of comparing the number of iteration, we also used execution time of each algorithm. In fact, RandomVI showed the longest execution time in table 1. This shows that randomly selecting states *does not improve much* in terms of convergence.

One possible reason why RandomVI perform worse than other algorithms is that in this small 2D grid world, state transition probabilities are deterministic. Therefore, applying sampling method on states and update on these states might not be efficient than updating state value for all states.

CyclicVI uses the updated state value to update the rest of state values. Although this algorithm does not provide significant improvement on both the number of iterations and execution time, it shows some improvements in Table 1.

Interestingly, RPCyclicVI showed some promising results. This algorithm permutes the set of states, B^k and same approach as CyclicVI, it uses the updated state value to update the rest of state values. This algorithm not only usually has the fewest number of iterations, but also provides shortest execution time as shown in Table 1.

2. General MDPs

We experimented on sample size on `RandomVI` and various values of total number of states, total number of actions, discount factors on algorithms in **Algorithms** section.

i) **Sample size effect on** `RandomVI`

- *Sample size of states and its effect on the performance*
  Figure 4 shows the graph of execution time versus different sample size of states.

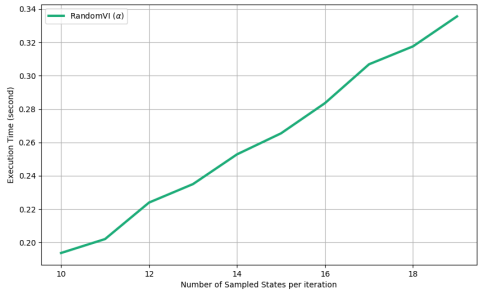

Figure 4: Experiment on different subset size, α

As the sample size increases, the execution time of `RandomVI` increases ($\approx$ linearly). However, when we decrease state subset size too small, `RandomVI` does not provide well converged output, $\mathbf{y}^*$. Even though it does not show good performance on MDP with deterministic transition probabilities, RandomVI shows better performance on general MDP cases. From this, we can infer that RandomVI works well for more stochastic mdp environment.

- *Sample size of actions and its effect on the performance*
  Figure 5 shows the graph of execution time versus different sample size of actions. Here, we build an empirical distribution for each action being selected as the winning action in the final policy and use it for sampling of actions. As the sample

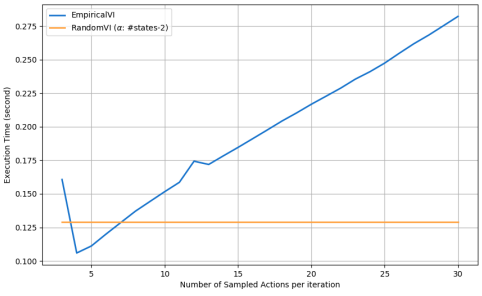

Figure 5: Experiment on different subset size, α

size increases, the execution time of `EM-RandomVI` increases ($\approx$ linearly). When we choose action subset size too large, the execution time becomes much larger than that of `RandomVI`. However, when we choose action subset size appropriately (*e.g.*, 5), we can achieve the reduced execution time and same convergence output.

For experiments on total number of states, total number of acitons and discount factors, we omitted `EM-RandomVI` part because if the total number of actions are not small enough, `EM-RandomVI` produces worst execution time among all algorithms. The reason is that keep tracking the frequency of previous actions, creating empirical distribution based on frequencies and then selecting a subset of actions from the distribution significantly contributes on execution time.

ii) **Various total number of states**

In this experiment, we use `TotalNumberOfStates` $\in [10, 19]$ and for each choice of `TotalNumberOfStates`, we run four algorithms 30 times and obtain the execution times for each run. (note that `TotalNumberOfStates` $\in \mathbb{Z}$ (integer)) After that, we took the median value. We used 'median' instead of 'mean' because since two algorithms uses randomness (stochastic approach), some outliers are present in our results and taking 'median' is robust to these outliers.

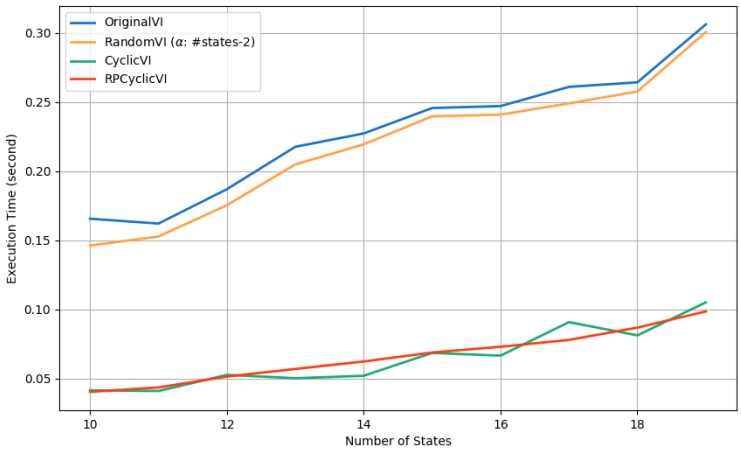

Figure 6: Execution Time vs. the Total Number of States

Figure 6 shows the result of experimenting on different number of states on four algorithms. When number of states are 10, classic VI (`OriginalVI`) shows the longest execution time (it took about 0.18 seconds.) and unlike previous experiment on small 2D grid world, `RandomVI` showed better performance than that of classic VI. Both `CyclicVI` and `RPCyclicVI` provided the best performances which results in the shortest execution time (both took about 0.05 seconds.).

Here, we can also observe that as the total number of states *increases*, the execution times for each algorithms also *increases*. However, even though execution time increases, both `CyclicVI` and `RPCyclicVI` shows best performances in terms of execution time and `RandomVI` still shows better performance than `OriginalVI`. In addition, we can observe that in few cases, `CyclicVI` performs better than `RPCyclicVI`. In general, we can rank the algorithms based on the performance as following:

$$\texttt{RPCyclicVI} \geq \texttt{CyclicVI} > \texttt{RandomVI} > \texttt{OriginalVI}$$

iii) **Various total number of actions**

In this experiment, we use `TotalNumberOfActions` $\in [50, 69]$ and for each choice of `TotalNumberOfActions`, we run four algorithms 30 times and obtain the execution times for each run. (note that `TotalNumberOfActions` $\in \mathbb{Z}$ (integer)) After that, we took the median value. We used 'median' instead of 'mean' because since two algorithms uses randomness (stochastic approach), some outliers are present in our results and taking 'median' is robust to these outliers.

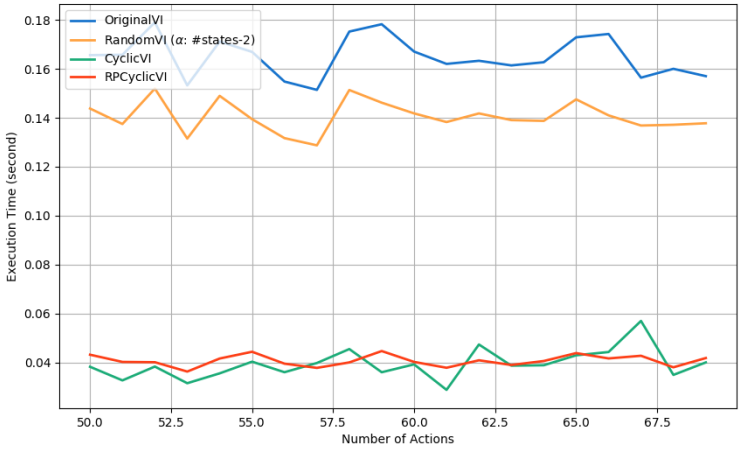

Figure 7: Execution Time vs. the Total Number of Actions

Figure 7 shows the result of experimenting on different number of actions on four algorithms. When number of states are 50, classic VI (`OriginalVI`) shows the longest execution time (it took about 0.18 seconds.) and unlike previous experiment on small 2D grid world, `RandomVI` showed *better* performance than `OriginalVI` (it took about 0.15 seconds.). Both `CyclicVI` and `RPCyclicVI` provided the best performances which results in the shortest execution time (both took about 0.04 seconds.).

In this figure, we can also observe that the total number of actions *does not affect* the execution time much on each algorithms. For example, when the total number of actions is 67, still `OriginalVI` took about 0.18 seconds and the result is still the worst performance among all.

Note that even though the total number of actions increases, both `CyclicVI` and `RPCyclicVI` show best performances in terms of execution time and `RandomVI` generally provides better performance than `OriginalVI`. In addition, we can notice that in few cases, `CyclicVI` performs better than `RPCyclicVI`. In general, we can rank the algorithms based on the performance as following:

$$\texttt{RPCyclicVI} \approx \texttt{CyclicVI} > \texttt{RandomVI} > \texttt{OriginalVI}$$

iv) **Various discount factors**
In this experiment, we use $\gamma \in [0.1, 0.9]$ and for each choice of γ, we run four algorithms 30 times and obtain the execution times for each run. After that, we took the median value. We used 'median' instead of 'mean' because since two algorithms uses randomness (stochastic approach), some outliers are present in our results and taking 'median' is robust to these outliers.

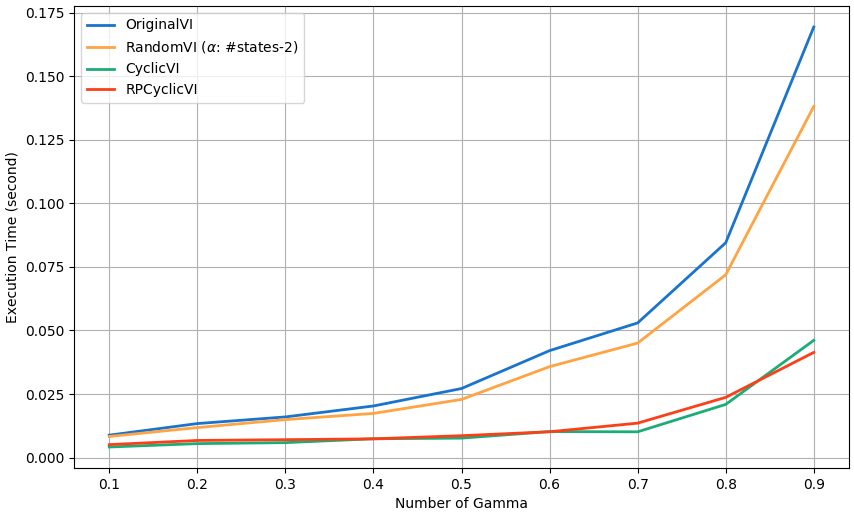

Figure 8: Execution Time vs. γ

Figure 8 shows the result of experimenting on different discount factor values (γ) on four algorithms. When $\gamma = 0.1$, both `CyclicVI` and `RPCyclicVI` shows best performances while the classic VI method and `RandomVI` shows the worst performance (However, note that the difference is very small). When $\gamma = 0.6$, we can notice that `RPCyclicVI` shows the best performance and the classic VI method shows the worst performance.

It is interesting to note that as γ increases, the execution time for all of four algorithms increases exponentially. However, even though execution time increases for all of four algorithms, the best performance is mostly achieved by `RPCyclicVI`. When γ is more than about 0.85, `CyclicVI` achieved slightly better performance than `RPCyclicVI`.

3. General MDPs with Two Extended Algorithms

Here, we show the performances of our two extended algorithm in comparison to other four algorithms in **Experiment section 2**.

i) **Various total number of states**
Figure 9 shows the experiment on all six algorithms about varying total number of states.

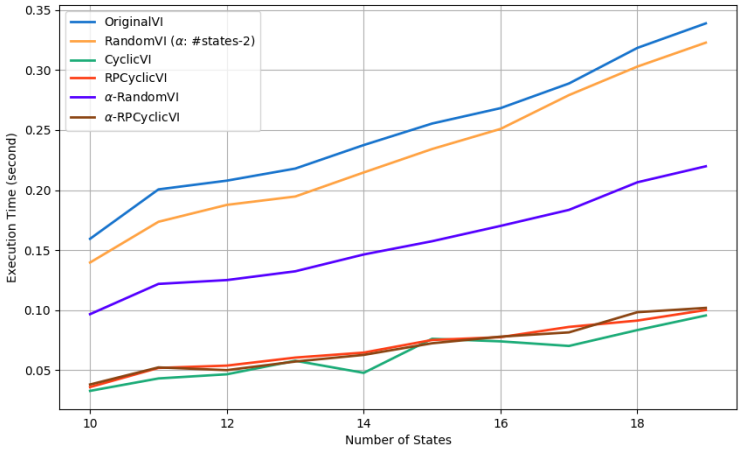

Figure 9: Execution Time vs. the Total Number of States

In this figure, we can observe that *increasing* total number of states also *increases* the execution time. We can also notice that α-`RandomVI` does not achieve good performances as `CyclicVI` or `RPCyclicVI` but clearly shows much *improved performances* than RandomVI or the classic VI method.

In addition, α-`RPCyclicVI` showed similar results with `RPCyclicVI`. The graph of α-`RPCyclicVI` looks almost same as `RPCyclicVI`.

ii) **Various total number of actions**

In this figure, we can observe that *increasing* total number of actions *does not* have

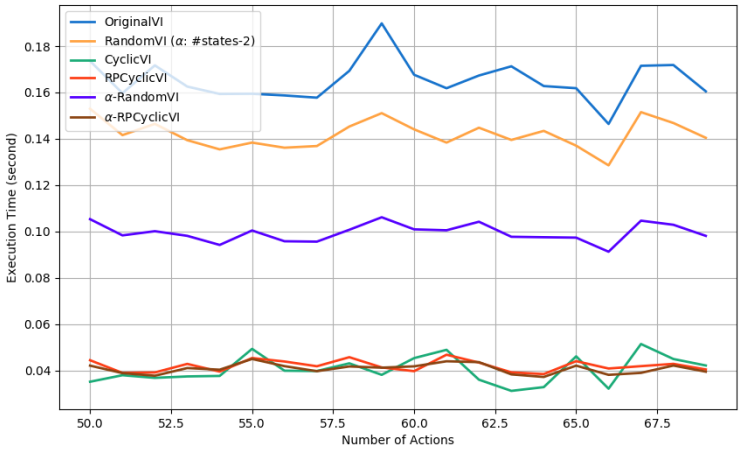

Figure 10: Execution Time vs. the Total Number of Actions

much impact on the execution time. Here, We can also notice that α-`RandomVI` does not achieve good performances as `CyclicVI` or `RPCyclicVI` but clearly shows much *improved performances* than RandomVI or the classic VI method as in Experiment 3 part i.

α-`RPCyclicVI` showed similar results with `RPCyclicVI`. The graph of α-`RPCyclicVI` looks almost same as `RPCyclicVI`.

iii) **Various discount factors**

Figure 11 shows the result of experimenting on different discount factor values (γ)

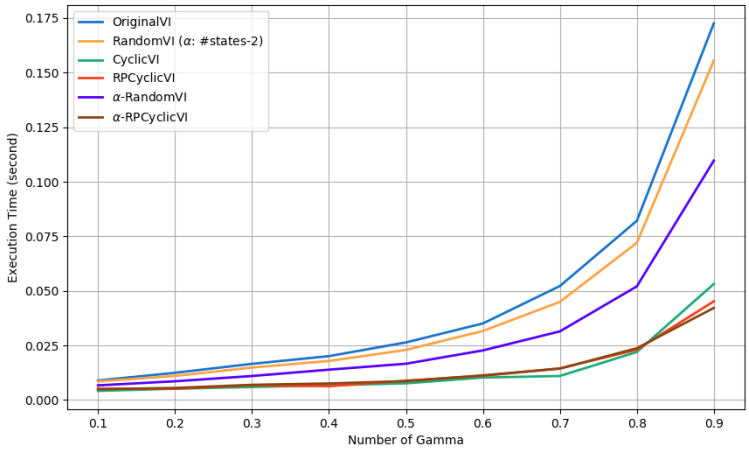

Figure 11: Execution Time vs. γ

on six algorithms. Here, we can observe that *increasing* discount factor *exponentially increases* the execution time. Here, We can also notice that α-`RandomVI` does not achieve good performances as `CyclicVI` or `RPCyclicVI` but clearly shows much *improved performances* than RandomVI or the classic VI method as in Experiment 3 part i and ii. In addition, α-`RPCyclicVI` showed similar results with `RPCyclicVI`. The graph of α-`RPCyclicVI` looks almost same as `RPCyclicVI`.

Here, we can conclude that α-`RandomVI` generally performs much better than `RandomVI` or `OriginalVI`. Also, using randomized subset size as in α-`RPCyclicVI` does not provide significant improvements than `RPCyclicVI` or `CyclicVI`

Appendix A

We include our `Python` code in this section.

Code for small 2D Grid World MDPs

Code for `mdp.py`

```
# MS&E310 Fall 2017
# Hyun Sik Kim (hsik@stanford.edu)
# Jongho Kim (jkim22@stanford.edu)
# Random VI implementation
import numpy as np
import random
import math
#import statistics

NUM_ROWS=5
NUM_COLS=5

def getNumberOfNeighbors(i, j, numCols=5, numRows=5):
    numberOfNeighbors = 4
    actions = {"left":1, "right":1, "down":1, "up":1}
    if i - 1 < 0:
        numberOfNeighbors -= 1
        actions["up"] = 0
    if i + 1 >= numRows:
        numberOfNeighbors -= 1
        actions["down"] = 0
    if j - 1 < 0:
        numberOfNeighbors -= 1
        actions["left"] = 0
    if j + 1 >= numCols:
        numberOfNeighbors -= 1
        actions["right"] = 0
    return numberOfNeighbors, actions

def updateTrapPoints(C, numCols, numRows):
    if numCols == 5 and numRows == 5:
        trapPoints = [(0,1), (1,1), (2,1), (4,1), (0,3), (2,3), (3,3), (4,3)]
        for trapPoint in trapPoints:
            C[trapPoint] = 100

def isInsideGrid(i, j, numRows, numCols):
    if i < 0:
        return False
    if j < 0:
        return False
    if j >= numCols:
        return False
    if i >= numRows:
```

```
            return False
        return True

class MDP:
    # We defined a 5 by 5 rectangular grid MDP for minimum cost path searching
    def __init__(self, numCols=5, numRows=5, gamma=0.8):
        # cost -1 for normal points
        # cost -10 for trap points
        self.C = {(i, j): -30 for i in range(numRows) for j in range(numCols)}
        updateTrapPoints(self.C, numCols, numRows)
        self.p = {(i, j): [] for i in range(numRows) for j in range(numCols)}
        # possible actions on each state
        self.stateActions = {}
        for i, j in self.p:
            numNeighbors, actions = getNumberOfNeighbors(i, j, numCols=5,
                numRows=5)
            self.stateActions[(i, j)] = [action for action in actions if
                actions[action] != 0]

        for i, j in self.p:
            self.p[(i, j)] = {action: {(row, col): 0 for row in range(numRows)
                for col in range(numCols)} for action in self.stateActions[(i
                ,j)]}
            for action in self.stateActions[(i,j)]:
                if action == "up":
                    self.p[(i,j)][action][(i-1,j)] = 1
                if action == "down":
                    self.p[(i,j)][action][(i+1,j)] = 1
                if action == "right":
                    self.p[(i,j)][action][(i,j+1)] = 1
                if action == "left":
                    self.p[(i,j)][action][(i,j-1)] = 1

        self.gamma = gamma
        self.numCols = numCols
        self.numRows = numRows
        # End state initialization
        self.C[(numRows-1, numCols-1)] = -100
        for action in self.stateActions[(numRows-1, numCols-1)]:
            self.p[(numRows-1, numCols-1)][action] = {state:0 for state in
                self.states()}
            self.p[(numRows-1, numCols-1)][action][(numRows-1, numCols-1)] = 1

    def states(self):
        states = [(i,j) for i in range(self.numRows) for j in range(self.
            numCols)]
        return states

    def IsEnd(self, state):
        if state == (self.numRows-1, self.numCols-1):
            return True
```

```
        return False

    def startState(self):
        # (0,0) upper left coner is the starting point
        return (0, 0)

    def actions(self, state):
        return self.p[state].keys()

    def succAndProbReward(self, state, action):
        if state == (self.numRows-1, self.numCols-1):
            return []

        next_states = []
        i, j = state
        actions = self.actions((i,j))
        for action in actions:
            if action == "up":
                if isInsideGrid(i-1, j, self.numRows, self.numCols):
                    next_states.append((i-1, j), self.p[i,j][action], self.
                        Grid[i-1][j])
            if action == "down":
                if isInsideGrid(i+1, j, self.numRows, self.numCols):
                    next_states.append((i+1, j), self.p[i,j][action], self.
                        Grid[i+1][j])
            if action == "left":
                if isInsideGrid(i, j-1, self.numRows, self.numCols):
                    next_states.append((i, j-1), self.p[i,j][action], self.
                        Grid[i+1][j-1])
            if action == "right":
                if isInsideGrid(i, j+1, self.numRows, self.numCols):
                    next_states.append((i, j+1), self.p[i,j][action], self.
                        Grid[i][j+1])

        return next_states
```

Code for `value.py`

```
import random
import math
import copy
import time
import mdp # MDP environment we've developed

NUM_ROWS=5
NUM_COLS=5
EPSILON=1e-5
NUM_STATES=25

# Bench Mark: Value Iteration
def vanillaVI(mdp):
    start_time = time.time()
    t = 0
    y = {state: 0 for state in mdp.states()}
    policy = {i:"None" for i in range(NUM_STATES)}
    while True:
        updated_y = copy.deepcopy(y)
        t += 1
        for state in mdp.states():
            candidates = []
            for action in mdp.p[state]:
                candidate = 0
                for next_state in mdp.p[state][action]:
                    candidate += mdp.gamma * mdp.p[state][action][next_state]
                        * y[next_state]

                row, col = state
                if action == "up":
                    next_state = (row-1, col)
                elif action == "down":
                    next_state = (row+1, col)
                elif action == "left":
                    next_state = (row, col-1)
                elif action == "right":
                    next_state = (row, col+1)

                # if it reaches the end state, then it doesn't have any next
                    state!
                if mdp.IsEnd(state):
                    candidates.append((0 + mdp.C[state], action))
                else:
                    candidates.append((candidate + mdp.C[next_state], action))

            updated_y[state] = min(candidates)[0]
            policy[state] = min(candidates)[1]
        checker = max(abs(y[state]-updated_y[state]) for state in mdp.states()
            )
```

```
        if checker <= EPSILON:
            execution_time = time.time() - start_time
            return updated_y, t, policy, execution_time

        y = updated_y


# Question 4: Random Value Iteration
def randomVI(mdp, subsetSize):
    start_time = time.time()
    t = 0
    y = {state: 0 for state in mdp.states()}
    policy = {i:"None" for i in range(NUM_STATES)}
    while True:
        updated_y = copy.deepcopy(y)
        t += 1
        B = random.sample(mdp.states(), subsetSize)
        for state in B:
            candidates = []
            for action in mdp.p[state]:
                candidate = 0
                for next_state in mdp.p[state][action]:
                    candidate += mdp.gamma * mdp.p[state][action][next_state]
                        * y[next_state]

                row, col = state
                if action == "up":
                    next_state = (row-1, col)
                elif action == "down":
                    next_state = (row+1, col)
                elif action == "left":
                    next_state = (row, col-1)
                elif action == "right":
                    next_state = (row, col+1)

                # if it reaches the end state, then it doesn't have any next
                    state!
                if mdp.IsEnd(state):
                    candidates.append((0 + mdp.C[next_state], action))
                else:
                    candidates.append((candidate + mdp.C[next_state], action))

            updated_y[state] = min(candidates)[0]
            policy[state] = min(candidates)[1]
        checker = max(abs(y[state]-updated_y[state]) for state in B)

        if checker <= EPSILON:
            execution_time = time.time() - start_time
            return updated_y, t, policy, execution_time

        y = updated_y
```

```
# Question 5 Cyclic Value Iteration
def cyclicVI(mdp):
    start_time = time.time()
    t = 0
    y = {state: 0 for state in mdp.states()}
    policy = {i:"None" for i in range(NUM_STATES)}
    while True:
        updated_y = copy.deepcopy(y)
        t += 1
        for state in mdp.states():
            candidates = []
            for action in mdp.p[state]:
                candidate = 0
                for next_state in mdp.p[state][action]:
                    candidate += mdp.gamma * mdp.p[state][action][next_state]
                        * y[next_state]

                row, col = state
                if action == "up":
                    next_state = (row-1, col)
                elif action == "down":
                    next_state = (row+1, col)
                elif action == "left":
                    next_state = (row, col-1)
                elif action == "right":
                    next_state = (row, col+1)

                # if it reaches the end state, then it doesn't have any next
                    state!
                if mdp.IsEnd(state):
                    candidates.append((0 + mdp.C[state], action))
                else:
                    candidates.append((candidate + mdp.C[next_state], action))

            y[state] = min(candidates)[0]
            policy[state] = min(candidates)[1]
        checker = max(abs(updated_y[state]-y[state]) for state in mdp.states()
            )

        if checker <= EPSILON:
            execution_time = time.time() - start_time
            return y, t, policy, execution_time


# Question 6: Random Permutation Cyclic Value Iteration
def RPcyclicVI(mdp):
    start_time = time.time()
    t = 0
    y = {state: 0 for state in mdp.states()}
```

```
    states = mdp.states()
    policy = {i:"None" for i in range(NUM_STATES)}
    while True:
        updated_y = copy.deepcopy(y)
        t += 1
        random.shuffle(states)
        for state in states:
            candidates = []
            for action in mdp.p[state]:
                candidate = 0
                for next_state in mdp.p[state][action]:
                    candidate += mdp.gamma * mdp.p[state][action][next_state]
                        * y[next_state]

                row, col = state
                if action == "up":
                    next_state = (row-1, col)
                elif action == "down":
                    next_state = (row+1, col)
                elif action == "left":
                    next_state = (row, col-1)
                elif action == "right":
                    next_state = (row, col+1)

                # if it reaches the end state, then it doesn't have any next
                    state!
                if mdp.IsEnd(state):
                    candidates.append((0 + mdp.C[state], action))
                else:
                    candidates.append((candidate + mdp.C[next_state], action))

            y[state] = min(candidates)[0]
            policy[state] = min(candidates)[1]
        checker = max(abs(updated_y[state]-y[state]) for state in mdp.states()
            )

        if checker <= EPSILON:
            execution_time = time.time() - start_time
            return y, t, policy, execution_time


# Print function for values and policy
def printer(grid, numCols=5, numRows=5):
    for row in range(numRows):
        line = []
        for col in range(numCols):
            line.append("("+str(row)+","+str(col)+")"+":")
            # for policy
            if (row, col) not in grid:
                line.append("None")
            else:
```

```
                # for policy
                if isinstance(grid[(row, col)], str):
                    line.append(grid[(row, col)])
                # for value
                else:
                    line.append("{0:.2f}".format(grid[(row, col)]))
        print line


if __name__=="__main__":
    mdp = mdp.MDP()

    # Value Iteration
    result_y, numIter, policy, execution_time = vanillaVI(mdp)
    print "Vanilla VI Number of iterations:", numIter
    print "Execution Time: {0:.3f}".format(execution_time)
    printer(result_y, mdp.numCols, mdp.numRows)
    printer(policy)
    print "=========================================================================="

    # Question 4
    for diff in range(24, 25): # when the subset size is too small, the
        algorithm does not work well!
        # Qustion 4
        subsetSize = NUM_STATES - diff
        result_y, numIter, policy, execution_time = randomVI(mdp, subsetSize)
        print "Random VI Number of iterations:", numIter, "SubsetSize:",
            subsetSize
        print "Execution Time: {0:.3f}".format(execution_time)
        printer(result_y, mdp.numCols, mdp.numRows)
        printer(policy)
        print "=========================================================================="

    # Qustion 5
    result_y, numIter, policy, execution_time = cyclicVI(mdp)
    print "Cyclic VI Number of iterations:", numIter
    print "Execution Time: {0:.3f}".format(execution_time)
    printer(result_y, mdp.numCols, mdp.numRows)
    printer(policy)
    print "=========================================================================="

    # Qustion 6
    result_y, numIter, policy, execution_time = RPcyclicVI(mdp)
    print "Random Permutation Cyclic VI Number of iterations:", numIter
    print "Execution Time: {0:.3f}".format(execution_time)
    printer(result_y, mdp.numCols, mdp.numRows)
    printer(policy)
    print "=========================================================================="

    '''
    # Qustion 4
```

```
#for i in range(24, 20, -1):
subsetSize = 24
result_y, numIter, policy = randomVI(mdp, subsetSize)
print "Random VI Number of iterations:", numIter, "Size of each subset:",
    subsetSize
#printer(result_y, mdp.numCols, mdp.numRows)
printer(policy)
'''
```

Code for General MDPs

Code for `kimMDP.py`

```
# Synthetic Experiment on MDP Value Iteration algorithms
import random
import numpy as np
import copy
import time
import statistics
import matplotlib.pyplot as plt

# Sparse probability: three vi algorithms are similar
# Large action space: reduce variation in num of iterations
class MDP:
    def __init__(self, NUM_STATES = 10, NUM_ACTIONS = 50, \
            lb = 30, ub = 50, GAMMA = 0.9, EPSILON = 1e-7):
        # Generates a random int vector with size = NUM_STATES
        ## [ |A1|, |A2|, ..., |Am| ]
        self.NUM_STATES=NUM_STATES
        self.NUM_ACTIONS=NUM_ACTIONS
        self.states = range(NUM_STATES)
        self.actions = range(NUM_ACTIONS)
        self.lb = lb
        self.up = ub
        self.GAMMA = GAMMA
        self.EPSILON = EPSILON

        self.stateActionSize = np.random.randint(lb, ub + 1,  size=NUM_STATES)
        self.stateActions = {}
        for i in range(NUM_STATES):
            self.stateActions[i] = sorted(random.sample(self.actions, self.
                stateActionSize[i]))

        self.stateActionCounts = {}
        for i in range(NUM_STATES):
            self.stateActionCounts[i] = {}
            for action in self.stateActions[i]:
                self.stateActionCounts[i][action] = 0

        # Generate probability matrix
        self.pmat = {}
        for i,actionSize in enumerate(self.stateActionSize):
            # for state i, pmatrix should have column size of actionSize, |Ai|
            self.pmat[i] = {}
            for action in self.stateActions[i]:
                # Generate probability for m states
                p = np.random.choice([0.1, 0.1, 0.1, 0.1, 200.0], NUM_STATES)
                p /= p.sum()
                self.pmat[i][action] = p

        self.c = {}
```

```
        for i in range(NUM_STATES):
            self.c[i] = {}
            for action in self.stateActions[i]:
                self.c[i][action] = np.random.randint(-1, 1 + 1, size=1)

# Value Iteration
def VanillaVI(mdp):
    start_time = time.time()
    t = 0
    # initialize y
    y = np.zeros(mdp.NUM_STATES)
    policy = {i:"None" for i in range(mdp.NUM_STATES)}
    states = range(mdp.NUM_STATES)
    while True:
        updated_y = copy.deepcopy(y)
        t += 1
        for i in states:
            candidates = []
            for action in mdp.stateActions[i]:
                candidate = mdp.c[i][action] + mdp.GAMMA * (np.dot(mdp.pmat[i
                    ][action], y))
                candidates.append((candidate, action))
            updated_y[i] = min(candidates)[0]
            policy[i] = min(candidates)[1]

        checker = max(abs(y[i]-updated_y[i]) for i in range(mdp.NUM_STATES))

        if checker <= mdp.EPSILON:
            execution_time = time.time() - start_time
            return updated_y, policy, t, execution_time
        y = updated_y

# Random Value Iteration
def RandomVI(mdp, subsetSize):
    start_time = time.time()
    t = 0
    y = np.zeros(mdp.NUM_STATES)
    policy = {i:"None" for i in range(mdp.NUM_STATES)}
    states = range(mdp.NUM_STATES)
    while True:
        subset = random.sample(states, subsetSize)
        updated_y = copy.deepcopy(y)
        t += 1
        for i in subset:
            candidates = []
            for action in mdp.stateActions[i]:
                candidate = mdp.c[i][action] + mdp.GAMMA * (np.dot(mdp.pmat[i
                    ][action], y))
                candidates.append((candidate, action))
            updated_y[i] = min(candidates)[0]
            policy[i] = min(candidates)[1]
```

```
        checker = max(abs(y[s]-updated_y[s]) for s in subset)

        if checker <= mdp.EPSILON:
            execution_time = time.time() - start_time
            return updated_y, policy, t, execution_time

        y = updated_y

# Random Value Iteration Empirical distribution
# Sampling Actions!!!
def Empirical_RandomVI(mdp, K):
    start_time = time.time()
    t = 0
    y = np.zeros(mdp.NUM_STATES)
    policy = {i:"None" for i in range(mdp.NUM_STATES)}

    states = range(mdp.NUM_STATES)

    weights = {}
    stateActionCounts = {}
    for i in range(mdp.NUM_STATES):
        weights[i] = {}
        stateActionCounts[i] = {}
        for action in mdp.stateActions[i]:
            weights[i][action] = 1.0/mdp.stateActionSize[i]
            stateActionCounts[i][action] = 1

    stateTotalActionCount = {i:mdp.stateActionSize[i] for i in range(mdp.
        NUM_STATES)}

    while True:
        updated_y = copy.deepcopy(y)
        t += 1
        for i in states:
            weight = [weights[i][action] for action in mdp.stateActions[i]]

            actions = np.random.choice(mdp.stateActions[i], size=K, replace=
                False, p=weight)

            candidates = []

            for action in actions:
                candidate = mdp.c[i][action] + mdp.GAMMA * (np.dot(mdp.pmat[i
                    ][action], y))
                candidates.append((candidate, action))
            updated_y[i] = min(candidates)[0]
            policy[i] = min(candidates)[1]
            stateActionCounts[i][policy[i]] += 1
            stateTotalActionCount[i] += 1
```

```
        checker = max(abs(y[s]-updated_y[s]) for s in states)

        if checker <= mdp.EPSILON:
            execution_time = time.time() - start_time
            return updated_y, policy, t, execution_time

        for i in range(mdp.NUM_STATES):
            for action in mdp.stateActions[i]:
                weights[i][action] = float(stateActionCounts[i][action]) /
                    stateTotalActionCount[i]

        y = updated_y

# Cyclic Value Iteration
def CyclicVI(mdp):
    start_time = time.time()
    t = 0
    y = np.zeros(mdp.NUM_STATES)
    policy = {i:"None" for i in range(mdp.NUM_STATES)}
    states = range(mdp.NUM_STATES)
    while True:
        updated_y = copy.deepcopy(y)
        t += 1
        for i in states:
            candidates = []
            for action in mdp.stateActions[i]:
                candidate = mdp.c[i][action] + mdp.GAMMA * (np.dot(mdp.pmat[i
                    ][action], y))
                candidates.append((candidate, action))
            y[i] = min(candidates)[0]
            policy[i] = min(candidates)[1]
        checker = max(abs(y[i]-updated_y[i]) for i in states)
        if checker <= mdp.EPSILON:
            execution_time = time.time() - start_time
            return y, policy, t, execution_time

# Random Permutation Value Iteration
def RPCyclicVI(mdp):
    start_time = time.time()
    t = 0
    y = np.zeros(mdp.NUM_STATES)
    policy = {i:"None" for i in range(mdp.NUM_STATES)}
    states = range(mdp.NUM_STATES)
    while True:
        updated_y = copy.deepcopy(y)
        t += 1

        random.shuffle(states)
        for i in states:
            candidates = []
            for action in mdp.stateActions[i]:
```

```
                candidate = mdp.c[i][action] + mdp.GAMMA * (np.dot(mdp.pmat[i
                    ][action], y))
                candidates.append((candidate, action))
            y[i] = min(candidates)[0]
            policy[i] = min(candidates)[1]
        checker = max(abs(y[i]-updated_y[i]) for i in states)

        if checker <= mdp.EPSILON:
            execution_time = time.time() - start_time
            return y, policy, t, execution_time

# Random Value Iteration Version 2
def RandomVI_Ver2(mdp):
    start_time = time.time()
    t = 0
    y = np.zeros(mdp.NUM_STATES)
    policy = {i:"None" for i in range(mdp.NUM_STATES)}
    states = range(mdp.NUM_STATES)
    while True:
        alpha = random.randint(1, mdp.NUM_STATES)
        subset = random.sample(states, alpha)
        updated_y = copy.deepcopy(y)
        t += 1
        for i in subset:
            candidates = []
            for action in mdp.stateActions[i]:
                candidate = mdp.c[i][action] + mdp.GAMMA * (np.dot(mdp.pmat[i
                    ][action], y))
                candidates.append((candidate, action))
            updated_y[i] = min(candidates)[0]
            policy[i] = min(candidates)[1]

        checker = max(abs(y[i]-updated_y[i]) for i in states)
        if checker <= mdp.EPSILON:
            execution_time = time.time() - start_time
            return updated_y, policy, t, execution_time

        y = updated_y


# Cyclic Value Iteration Version 2 (Basically with random permutation)
def CyclicVI_Ver2(mdp):
    start_time = time.time()
    t = 0
    y = np.zeros(mdp.NUM_STATES)
    policy = {i:"None" for i in range(mdp.NUM_STATES)}
    states = range(mdp.NUM_STATES)
    while True:
        updated_y = copy.deepcopy(y)
        t += 1
        alpha = random.randint(1, mdp.NUM_STATES)
```

```
        subset = random.sample(states, alpha)
        for i in subset:
            candidates = []
            for action in mdp.stateActions[i]:
                candidate = mdp.c[i][action] + mdp.GAMMA * (np.dot(mdp.pmat[i
                    ][action], y))
                candidates.append((candidate, action))
            y[i] = min(candidates)[0]
            policy[i] = min(candidates)[1]
        checker = max(abs(y[i]-updated_y[i]) for i in subset)
        if checker <= mdp.EPSILON:
            execution_time = time.time() - start_time
            return y, policy, t, execution_time


def policyPrinter(policy):
    for i in range(NUM_STATES):
        print "State", i, ":", "policy:", policy[i]

def yPrinter(y):
    for i in range(NUM_STATES):
        print "State", i, ":", "y_value:", y[i]

def freqPrinter(stateActionCounts):
    for i in range(NUM_STATES):
        line = []
        for action in stateActions[i]:
            line.append((action, stateActionCounts[i][action]))

        print "State", i, ":", "Action, Count:", line

def plot(mode, x, VI, RVI, CVI, RPCVI, RVI2, KVI, diff):
    plt.figure(figsize=(10,6), dpi=100)
    ax=plt.subplot(111)

    plt.plot(x, VI, color="#1874CD", linewidth=2.0, linestyle="-", label='
        OriginalVI')
    plt.plot(x, RVI, color="#FFA343", linewidth=2.0, linestyle="-", label='
        RandomVI (' + r"$\alpha$" + "):#states-" + str(diff) + ")")
    plt.plot(x, CVI, color="#1CAC78", linewidth=2.0, linestyle="-", label='
        CyclicVI')
    plt.plot(x, RPCVI, color="#FD3F17", linewidth=2.0, linestyle="-", label='
        RPCyclicVI')
    plt.plot(x, RVI2, color="#5500FF", linewidth=2.0, linestyle="-", label=r"$
        \alpha$"+ '-RandomVI')
    plt.plot(x, KVI, color="#8B4513", linewidth=2.0, linestyle="-", label=r"$\
        alpha$"+ '-RPCyclicVI')

    plt.grid()

    # mode = {"States", "Actions", "Gamma"}
```

```
    xlabel = "Number of " + mode
    file_name = "./" + mode + "_execution_time_extension.png"

    plt.xlabel(xlabel)
    plt.ylabel("Execution Time (second)")
    plt.legend(loc='upper left')
    plt.savefig(file_name)
    plt.show()

def plot4(x, EVI, RVI, diff):
    plt.figure(figsize=(10,6), dpi=100)
    ax=plt.subplot(111)

    plt.plot(x, EVI, color="#1874CD", linewidth=2.0, linestyle="-", label='
        EmpiricalVI')
    plt.plot(x, RVI, color="#FFA343", linewidth=2.0, linestyle="-", label='
        RandomVI (' + r"$\alpha$" + ": #states-" + str(diff) + ")")

    plt.grid()

    # mode = {"States", "Actions", "Gamma"}
    xlabel = "Number of Sampled Actions per iteration"
    file_name = "./Question4_empirical_execution_time.png"

    plt.xlabel(xlabel)
    plt.ylabel("Execution Time (second)")
    plt.legend(loc='upper left')
    plt.savefig(file_name)
    plt.show()

def plot4_subset(x, RVI, subsetSize):
    plt.figure(figsize=(10,6), dpi=100)
    ax=plt.subplot(111)

    plt.plot(x, RVI, color="#1CAC78", linewidth=3.0, linestyle="-", label='
        RandomVI (' + r"$\alpha$" + ")")

    plt.grid()

    xlabel = "Number of Sampled States per iteration"
    file_name = "./Question4_different_subset_size.png"

    plt.xlabel(xlabel)
    plt.ylabel("Execution Time (second)")
    plt.legend(loc='upper left')
    plt.savefig(file_name)
    plt.show()

# Main
if __name__=="__main__":
```

```
#### Setting
# m states
NUM_STATES = 10
NUM_ACTIONS = 50
states = range(NUM_STATES)
actions = range(NUM_ACTIONS)
lb = 30 # lower bound
ub = 50 # upper bound
GAMMA = 0.9
EPSILON = 1e-7
NUM_TRIALS=30
K = 5
NUM_ITER_4 = 30
# data for plot
x = []
VI=[]
RVI=[]
CVI=[]
RPCVI=[]
RVI2=[]
KVI=[]

diff = 2

# Choose mode depending on the plot we'd like to plot
mode = "States"
for numStates in range(10, 20):
    NUM_STATES = numStates
    x.append(numStates)

#mode = "Actions"
#for numActions in range(50, 70, 1):
#    NUM_ACTIONS = numActions
#    x.append(numActions)

#mode = "Gamma"
#NUM_GAMMA = 10
#Gamma = [i / float(NUM_GAMMA) for i in range(1, NUM_GAMMA + 1)]
#for gamma in Gamma[:-1]:
#    GAMMA = gamma
#    x.append(gamma)

    mdp = MDP(NUM_STATES=NUM_STATES, NUM_ACTIONS=NUM_ACTIONS,\
            lb=lb, ub=ub, GAMMA=GAMMA, EPSILON=EPSILON)

    # Bench Mark
    updated_y, policy, t, execution_time = VanillaVI(mdp)
    VI.append(execution_time)
    #print "Value Iteration:"
    #print "Iterations:", t
    #print "Execution Time: {0:.3f}".format(execution_time)
```

```
        #yPrinter(updated_y)
        #print "=========================================================="
        #policyPrinter(policy)

        RVI_memory = []
        RVI2_memory = []
        KVI_memory = []
        for _ in range(NUM_TRIALS):
            subsetSize = NUM_STATES - diff
            updated_y, policy, t, execution_time = RandomVI(mdp, subsetSize)
            RVI_memory.append(execution_time)

            updated_y, policy, t, execution_time = RandomVI_Ver2(mdp)
            RVI2_memory.append(execution_time)

            updated_y, policy, t, execution_time = CyclicVI_Ver2(mdp)
            KVI_memory.append(execution_time)

        RVI.append(statistics.median(RVI_memory))
        RVI2.append(statistics.median(RVI2_memory))
        KVI.append(statistics.median(KVI_memory))


        updated_y, policy, t, execution_time = CyclicVI(mdp)
        CVI.append(execution_time)


        updated_y, policy, t, execution_time = RPCyclicVI(mdp)
        RPCVI.append(execution_time)

        print mode, ":", x[-1], "Completed"

    plot(mode, x, VI, RVI, CVI, RPCVI, RVI2, KVI, diff)


    # Question 4: Different subset size of states
    x = []
    NUM_STATES=20
    for i in range(NUM_TRIALS):
        mdp = MDP(NUM_STATES=NUM_STATES, NUM_ACTIONS=NUM_ACTIONS,\
                    lb=lb, ub=ub, GAMMA=GAMMA, EPSILON=EPSILON)
        RVI_memory = {i:[] for i in range(10, NUM_STATES)}
        for subsetSize in range(10, NUM_STATES):
            if i == 0:
                x.append(subsetSize)
            updated_y, policy, t, execution_time = RandomVI(mdp, subsetSize)
            RVI_memory[subsetSize].append(execution_time)
        print i
    RVI = []
    for subsetSize in range(10, NUM_STATES):
        RVI.append(statistics.median(RVI_memory[subsetSize]))
```

```
plot4_subset(x, RVI, diff)


# Question 4 Advanced:
mdp = MDP(NUM_STATES=NUM_STATES, NUM_ACTIONS=NUM_ACTIONS,\
            lb=lb, ub=ub, GAMMA=GAMMA, EPSILON=EPSILON)

x = []
EVI = []
RVI = []
for K in range(3, lb+1):
    x.append(K)
    EVI_memory = []
    for _ in range(NUM_ITER_4):
        updated_y, policy, t, execution_time = Empirical_RandomVI(mdp, K=K
            )
        EVI_memory.append(execution_time)
        #print "Random Value Iteration Empirical distribution, K =", K
        #print "Iterations:", t
        #print "Execution Time: {0:.3f}".format(execution_time)
    EVI.append(statistics.median(EVI_memory))
    subsetSize = NUM_STATES - diff
    updated_y, policy, t, execution_time = RandomVI(mdp, subsetSize)
    RVI.append(execution_time)
    #print "Random Value Iteration,", "Subset Size:", subsetSize
    print "Iterations:", t
    #print "Execution Time: {0:.3f}".format(execution_time)
RVI_mean = statistics.median(RVI)
RVI = [RVI_mean for _ in range(3, lb+1)]

plot4(x, EVI, RVI, diff)
```

Appendix B

B.1 Small 2D Grid World

On terminal, execute following:

```
python value.py
```

```
Vanilla VI Number of iterations: 14
Execution Time: 0.014
['(0,0):', '-152.58', '(0,1):', '-157.86', '(0,2):', '-159.83', '(0,3):', '-162.29', '(0,4):', '-165.36']
['(1,0):', '-153.22', '(1,1):', '-159.83', '(1,2):', '-162.29', '(1,3):', '-165.36', '(1,4):', '-169.20']
['(2,0):', '-154.03', '(2,1):', '-157.86', '(2,2):', '-159.83', '(2,3):', '-169.20', '(2,4):', '-174.00']
['(3,0):', '-155.03', '(3,1):', '-156.29', '(3,2):', '-157.86', '(3,3):', '-174.00', '(3,4):', '-180.00']
['(4,0):', '-154.03', '(4,1):', '-155.03', '(4,2):', '-156.29', '(4,3):', '-180.00', '(4,4):', '-100.00']
['(0,0):', 'down', '(0,1):', 'right', '(0,2):', 'down', '(0,3):', 'down', '(0,4):', 'down']
['(1,0):', 'down', '(1,1):', 'right', '(1,2):', 'right', '(1,3):', 'right', '(1,4):', 'down']
['(2,0):', 'down', '(2,1):', 'right', '(2,2):', 'up', '(2,3):', 'right', '(2,4):', 'down']
['(3,0):', 'right', '(3,1):', 'right', '(3,2):', 'up', '(3,3):', 'right', '(3,4):', 'down']
['(4,0):', 'up', '(4,1):', 'right', '(4,2):', 'up', '(4,3):', 'right', '(4,4):', 'left']
====================================================
Random VI Number of iterations: 2 SubsetSize: 1
Execution Time: 0.000
['(0,0):', '0.00', '(0,1):', '0.00', '(0,2):', '0.00', '(0,3):', '0.00', '(0,4):', '0.00']
['(1,0):', '0.00', '(1,1):', '0.00', '(1,2):', '0.00', '(1,3):', '0.00', '(1,4):', '0.00']
['(2,0):', '0.00', '(2,1):', '0.00', '(2,2):', '0.00', '(2,3):', '-30.00', '(2,4):', '0.00']
['(3,0):', '0.00', '(3,1):', '0.00', '(3,2):', '0.00', '(3,3):', '0.00', '(3,4):', '0.00']
['(4,0):', '0.00', '(4,1):', '0.00', '(4,2):', '0.00', '(4,3):', '0.00', '(4,4):', '0.00']
['(0,0):', 'None', '(0,1):', 'None', '(0,2):', 'None', '(0,3):', 'None', '(0,4):', 'None']
['(1,0):', 'None', '(1,1):', 'None', '(1,2):', 'None', '(1,3):', 'None', '(1,4):', 'None']
['(2,0):', 'None', '(2,1):', 'None', '(2,2):', 'None', '(2,3):', 'left', '(2,4):', 'None']
['(3,0):', 'None', '(3,1):', 'None', '(3,2):', 'None', '(3,3):', 'None', '(3,4):', 'None']
['(4,0):', 'None', '(4,1):', 'None', '(4,2):', 'None', '(4,3):', 'None', '(4,4):', 'None']
====================================================
Cyclic VI Number of iterations: 12
Execution Time: 0.011
['(0,0):', '-152.58', '(0,1):', '-157.86', '(0,2):', '-159.83', '(0,3):', '-162.29', '(0,4):', '-165.36']
['(1,0):', '-153.22', '(1,1):', '-159.83', '(1,2):', '-162.29', '(1,3):', '-165.36', '(1,4):', '-169.20']
['(2,0):', '-154.03', '(2,1):', '-157.86', '(2,2):', '-159.83', '(2,3):', '-169.20', '(2,4):', '-174.00']
['(3,0):', '-155.03', '(3,1):', '-156.29', '(3,2):', '-157.86', '(3,3):', '-174.00', '(3,4):', '-180.00']
['(4,0):', '-154.03', '(4,1):', '-155.03', '(4,2):', '-156.29', '(4,3):', '-180.00', '(4,4):', '-100.00']
['(0,0):', 'down', '(0,1):', 'right', '(0,2):', 'down', '(0,3):', 'down', '(0,4):', 'down']
['(1,0):', 'down', '(1,1):', 'right', '(1,2):', 'right', '(1,3):', 'right', '(1,4):', 'down']
['(2,0):', 'down', '(2,1):', 'right', '(2,2):', 'up', '(2,3):', 'right', '(2,4):', 'down']
['(3,0):', 'right', '(3,1):', 'right', '(3,2):', 'up', '(3,3):', 'right', '(3,4):', 'down']
['(4,0):', 'up', '(4,1):', 'right', '(4,2):', 'up', '(4,3):', 'right', '(4,4):', 'left']
====================================================
Random Permutation Cyclic VI Number of iterations: 9
Execution Time: 0.009
['(0,0):', '-152.58', '(0,1):', '-157.86', '(0,2):', '-159.83', '(0,3):', '-162.29', '(0,4):', '-165.36']
['(1,0):', '-153.22', '(1,1):', '-159.83', '(1,2):', '-162.29', '(1,3):', '-165.36', '(1,4):', '-169.20']
['(2,0):', '-154.03', '(2,1):', '-157.86', '(2,2):', '-159.83', '(2,3):', '-169.20', '(2,4):', '-174.00']
['(3,0):', '-155.03', '(3,1):', '-156.29', '(3,2):', '-157.86', '(3,3):', '-174.00', '(3,4):', '-180.00']
['(4,0):', '-154.03', '(4,1):', '-155.03', '(4,2):', '-156.29', '(4,3):', '-180.00', '(4,4):', '-100.00']
['(0,0):', 'down', '(0,1):', 'right', '(0,2):', 'down', '(0,3):', 'down', '(0,4):', 'down']
['(1,0):', 'down', '(1,1):', 'right', '(1,2):', 'right', '(1,3):', 'right', '(1,4):', 'down']
['(2,0):', 'down', '(2,1):', 'right', '(2,2):', 'up', '(2,3):', 'right', '(2,4):', 'down']
['(3,0):', 'right', '(3,1):', 'right', '(3,2):', 'up', '(3,3):', 'right', '(3,4):', 'down']
['(4,0):', 'up', '(4,1):', 'right', '(4,2):', 'up', '(4,3):', 'right', '(4,4):', 'left']
====================================================
```

*Here, we can notice that the policy obtained by four algorithms as in **question 4 to 6** are identical as target policy in Figure 3

B.2 General MDPs

On terminal, execute following:

```
python kimMDP.py
```