Single Agent Planning

Duy Chuan Ha, Said Al Faraby, Christos Louizos, Oana Munteanu

September 20, 2013

Abstract

The aim of this assignment is to make a research according to the planning scenario in which the main goal of the predator is to capture the prey. The report comprises the methods that have been implemented by using the Dynamic Programming paradigm, namely policy evaluation, policy iteration and value iteration. Our contribution on Single Agent Planning reveals the comparison between the implemented methods in terms of convergence time and results.
+Conclusion

1 INTRODUCTION

The general idea of reinforcement learning is to acquire knowledge from experience, specifically interacting with the environment, getting feedback from it in the form of rewards, in order to achieve a goal. To perform this kind of task there is a need of a policy, which basically summarizes the course of action an agent should take at any specific state of the environment. The agent always seeks to maximize the amount of reward it receives, either immediately or in the long run.

In order to be able to get an optimal policy given the environment, the agent should evaluate the current policy and then improve it if it's not the optimal one. The whole loop can be summed up pretty well in figure 1.1. If we already a complete model of the environment, and the environment satisfies the Markov property, namely it is an MDP, we can solve this problem by using a paradigm called dynamic programming(DP), which evaluates and improves the policy at hand. There is a variety of methods in DP that are guaranteed to find the optimal policy for us, namely policy evaluation(2.2), policy iteration(2.3) and value iteration(2.4).

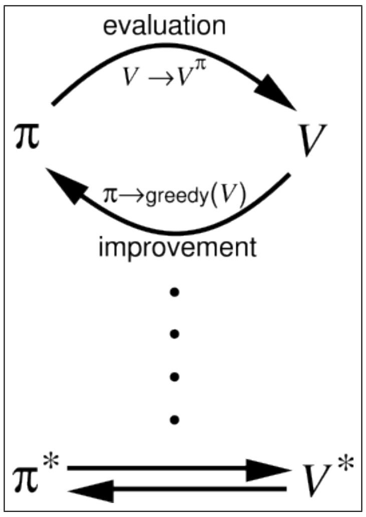


Figure 1.1: Generalized Policy Iteration

We begin the following report by first explaining the predator-prey environment and defining it's goal in section 1.1. Afterwards we present in section 2 the various methods that were implemented in the aforementioned environment. The results that derived from each of these methods are displayed in detail in section 3 and finally there is a discussion as well as a comparison between them in section 4

1.1 ENVIRONMENT

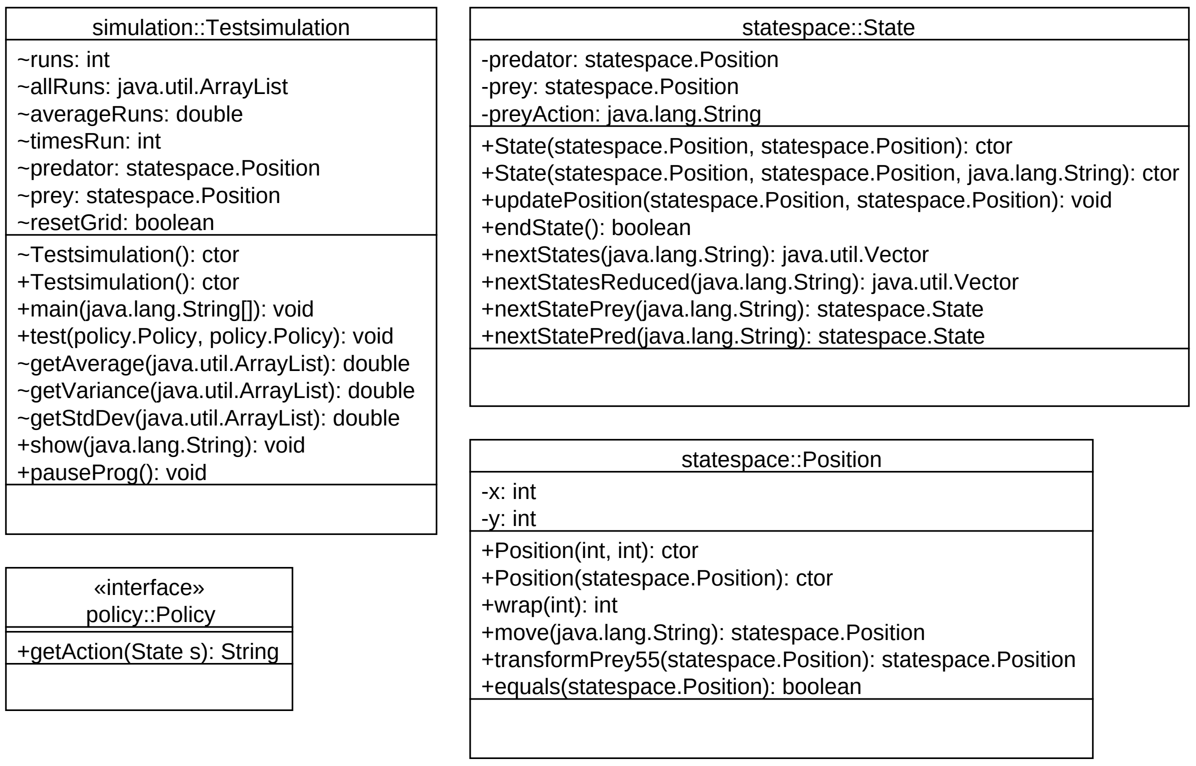
The environment is a 11x11 grid which is toroidal, and each state is encoded as the positions of the two agents, predator and prey. The simulator is defined through a while-loop of a hundred runs where the initial position of the predator is (0,0) and that of the prey (5,5). In each transition, the predator will make a move first and then the prey, according to their respective policies. Each run is composed of an episode where the goal of the predator is to capture the prey, which can be interpreted as the end state of this particular episode.

Both the predator and the prey are initialized with a random policy in the beginning where for the predator it is equiprobable to choose any of the possible actions in any state. The prey has a steady policy and hence it can be modelled as part of the environment. It has 0.8 probability of waiting and 0.2 probability of moving to any of the adjacent squares.

2 Methods and Procedures

2.1 Implementation overview

In our implementation the agents are encoded as positions by the position class: each entity consists of an x,y coordinate denoting it's position in the 11x11 grid. A wrapper method herein takes care of the grid being toroidal. A move in the grid of an agent is implemented. A state consists of two positions that of the prey and of the predator. Herein we have encoded the end state and methods for giving the conditional next state. Policies are encapsulated by the policy interface which ensures each implemented policy to have a getAction() method. In each implemented policy the statespace is defined by a hashtable of all possible states; visualized as statespace[predator(x,y), prey(x,y)].



```
simulation::Testsimulation
~runs: int
~allRuns: java.util.ArrayList
~averageRuns: double
~timesRun: int
~predator: statespace.Position
~prey: statespace.Position
~resetGrid: boolean
~Testsimulation(): ctor
+Testsimulation(): ctor
+main(java.lang.String[]): void
+test(policy.Policy, policy.Policy): void
~getAverage(java.util.ArrayList): double
~getVariance(java.util.ArrayList): double
~getStdDev(java.util.ArrayList): double
+show(java.lang.String): void
+pauseProg(): void
```

```
statespace::State
-predator: statespace.Position
-prey: statespace.Position
-preyAction: java.lang.String
+State(statespace.Position, statespace.Position): ctor
+State(statespace.Position, statespace.Position, java.lang.String): ctor
+updatePosition(statespace.Position, statespace.Position): void
+endState(): boolean
+nextStates(java.lang.String): java.util.Vector
+nextStatesReduced(java.lang.String): java.util.Vector
+nextStatePrey(java.lang.String): statespace.State
+nextStatePred(java.lang.String): statespace.State
```

```
«interface»
policy::Policy
+getAction(State s): String
```

```
statespace::Position
-x: int
-y: int
+Position(int, int): ctor
+Position(statespace.Position): ctor
+wrap(int): int
+move(java.lang.String): statespace.Position
+transformPrey55(statespace.Position): statespace.Position
+equals(statespace.Position): boolean
```

Figure 2.1: Predator-prey domain framework

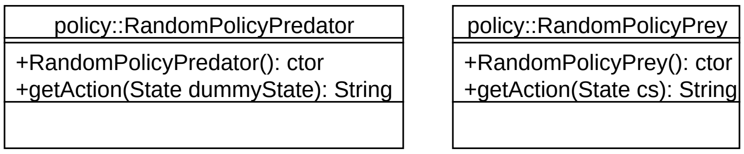

policy::RandomPolicyPredator
+RandomPolicyPredator(): ctor
+getAction(State dummyState): String
policy::RandomPolicyPrey
+RandomPolicyPrey(): ctor
+getAction(State cs): String

Figure 2.2: Random policies implemented

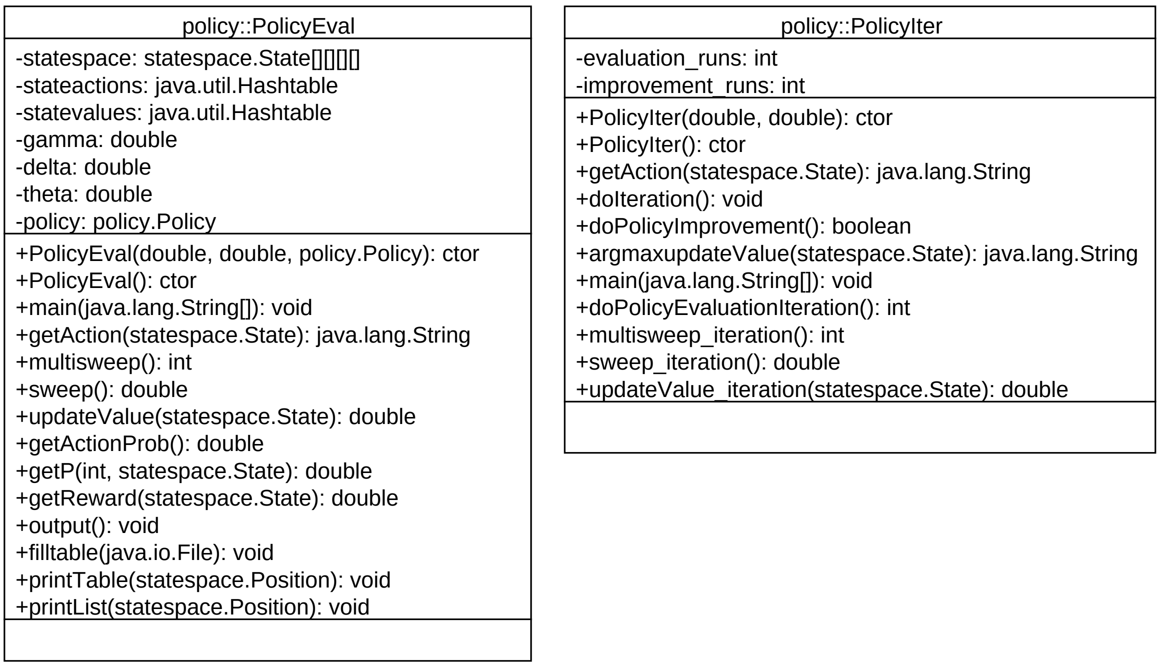
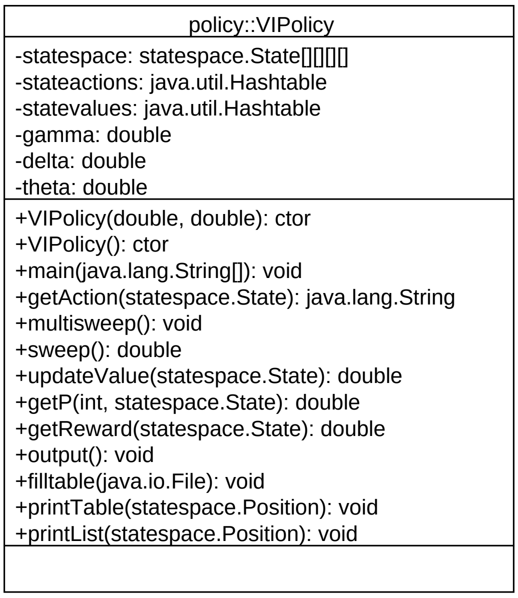

policy::PolicyEval
-statespace: statespace.State[][][][]
-stateactions: java.util.Hashtable
-statevalues: java.util.Hashtable
-gamma: double
-delta: double
-theta: double
-policy: policy.Policy
+PolicyEval(double, double, policy.Policy): ctor
+PolicyEval(): ctor
+main(java.lang.String[]): void
+getAction(statespace.State): java.lang.String
+multisweep(): int
+sweep(): double
+updateValue(statespace.State): double
+getActionProb(): double
+getP(int, statespace.State): double
+getReward(statespace.State): double
+output(): void
+filltable(java.io.File): void
+printTable(statespace.Position): void
+printList(statespace.Position): void
policy::PolicyIter
-evaluation_runs: int
-improvement_runs: int
+PolicyIter(double, double): ctor
+PolicyIter(): ctor
+getAction(statespace.State): java.lang.String
+doIteration(): void
+doPolicyImprovement(): boolean
+argmaxupdateValue(statespace.State): java.lang.String
+main(java.lang.String[]): void
+doPolicyEvaluationIteration(): int
+multisweep_iteration(): int
+sweep_iteration(): double
+updateValue_iteration(statespace.State): double

Figure 2.3: Policy evaluation and iteration implemented



```
policy::VIPolicy
-statespace: statespace.State[][][][]
-stateactions: java.util.Hashtable
-statevalues: java.util.Hashtable
-gamma: double
-delta: double
-theta: double
+VIPolicy(double, double): ctor
+VIPolicy(): ctor
+main(java.lang.String[]): void
+getAction(statespace.State): java.lang.String
+multisweep(): void
+sweep(): double
+updateValue(statespace.State): double
+getP(int, statespace.State): double
+getReward(statespace.State): double
+output(): void
+filltable(java.io.File): void
+printTable(statespace.Position): void
+printList(statespace.Position): void
```

Figure 2.4: Policy value iteration implemented

2.2 Iterative Policy Evaluation

In Markov Decision Processes, one of the important steps is evaluation step, which is to find the state-value functions V^{π} for an arbitrary policy, π. The state-value functions estimate value of a state in terms of expected return that can be obtained from that state. Below is the equation for calculating V^{π}.

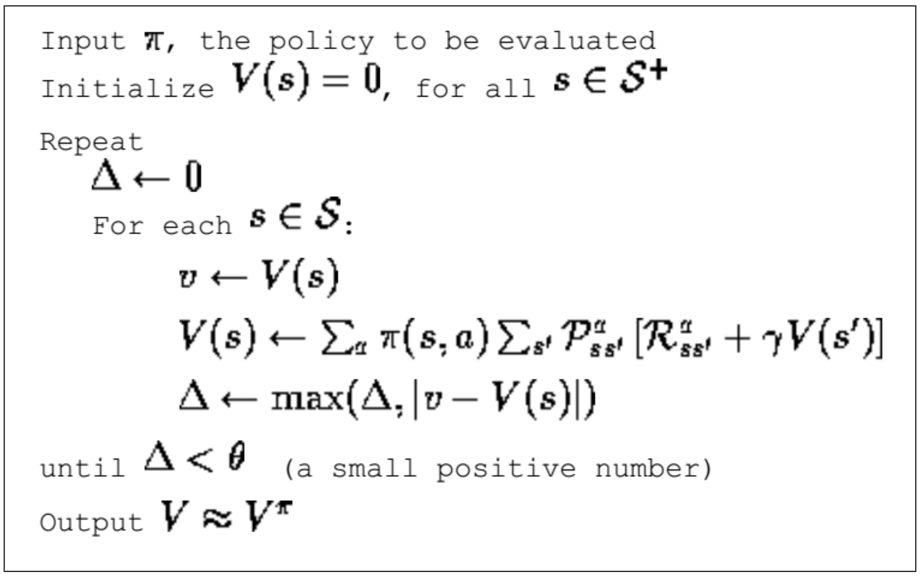
$$\begin{aligned} V^{\pi}(s) &= E_{\pi}\left\{r_{t+1} + \gamma V^{\pi}(s_{s+1}) \mid s_t = s\right\} \\ &= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V^{\pi}(s')] \end{aligned} \tag{2.1}$$

where $\pi(\mathbf{s},\mathbf{a})$ is probability of taking action **a** in state **s** under policy π, $P^a_{ss'}$ is probability of end up at state $\mathbf{s}'$ after taking action **a** in state **s**, $R^a_{ss'}$ is expected immediate reward on transition from **s** to $\mathbf{s}'$ under action **a**, and γ is a discount factor.

Since in our test environment the dinamics are completely known, then equation (2.1) can be solved using iterative solution methods. By iterative solution methods, we can consider there is a sequence of V, $V_0, V_1, V_2, \ldots$, each mapping all states in S^+ to $\Re$. The initial values of V^{π} are chosen arbitrarily, and the update rule for each successive state-values are gained by following the Bellman equation :

$$\begin{aligned} V_{k+1}(s) &= E_{\pi}\left\{r_{t+1} + \gamma V_k(s_{t+1} \mid s_t = s)\right\} \\ &= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V_k(s')] \end{aligned} \tag{2.2}$$

For the technical implementation, we used one array to save state-values and updated the values in place. This implementation is slightly different from (2.2), because there is a possibility that $V_k(s')$ uses new values instead of old ones, but this implementation still guarantees to converge to V^π. Another technical implementation to be considered is the stopping criteria. In formal, iterative policy iteration will converge only in the limit, but for practical purposes stopping condition is necessary. One of common criteria to stop the sweep is when there is no any single state managed to make a significant changes to their values. A complete pseudo code of the whole processes of the iterative policy evaluation is shown in figure 2.5.



```
Input $\pi$, the policy to be evaluated
Initialize $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx V^\pi$
```

Figure 2.5: Iterative Policy Evaluation [1]

2.3 Policy Iteration

Policy iteration is composed of two steps, the policy evaluation, which was explained above, and the policy improvement. The main reason for calculating the value function for a policy is to use it in order to find better policies. Consequently if we have determined the value function V^π for a deterministic policy π we can then select a different action α for one state s and afterwards keep on using that specific action for that state. Therefore we can determine if the change improved the policy or not by calculating the following formula:

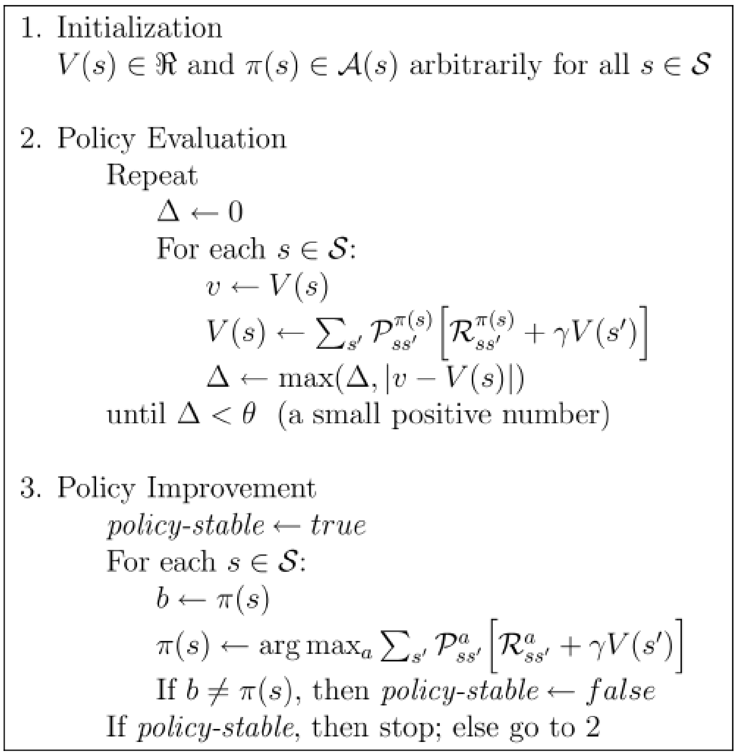
$$\begin{aligned} Q^\pi(s,\alpha) &= E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s, \alpha_t = \alpha\} \\ &= \sum_{s'} P^\alpha_{ss'}[R^\alpha_{ss'} + \gamma V^\pi(s')] \end{aligned} \tag{2.3}$$

if this is greater than the value of the existing policy in that state $V^\pi(s)$ then it would be better to continue taking action α every time we encounter state s, which results in a new policy π'. This is true because it derives from the policy improvement theorem where if we have two deterministic policies π and π' such that for all states $s \in \mathbf{S}$ the $Q^\pi(s, \pi'(s)) >= V^\pi(s)$ then the policy π' must be equal to or better than policy π meaning that it will yield equal or higher values for all state s. Thus if $Q^{\pi}(s,\alpha) > V^{\pi}(s)$ then the changed policy π' is indeed better than π. There is one more step needed in order to finish the policy improvement. Since we can acquire a better reward by doing a different action than the one the policy provide at a given state, then it makes sense to choose the action that maximizes that reward and incorporate it to the new policy π'. Then the formula resulting from this step is:

$$\begin{aligned}\pi'(s) &= arg\max_{\alpha} Q^{\pi}(s,\alpha)\\ &= arg\max_{\alpha} Er_{t+1} + \gamma V^{\pi}(s_{t+1})|s_t = s, \alpha_t = \alpha\\ &= arg\max_{\alpha} \sum_{s'} P_{ss'}\alpha[R^{\alpha}_{ss'} + \gamma V^{\pi}(s')]\end{aligned} \tag{2.4}$$

where $arg\max_{\alpha}$ denotes the action that maximizes the term that is following. Taking all the above into consideration we can say that the process of making a new policy that improves the existing one by being greedy on the value function of the original policy, is called policy improvement.

The whole pseudo code loop of iterative evaluation and improvement of the current policy in order to produce a better policy π' until we converge to the optimal policy is illustrated in figure 2.6.



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

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

3. Policy Improvement
       *policy-stable* $\leftarrow$ *true*
       For each $s \in \mathcal{S}$:
           $b \leftarrow \pi(s)$
           $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma V(s')\right]$
           If $b \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
       If *policy-stable*, then stop; else go to 2
```

Figure 2.6: Policy Iteration [1]

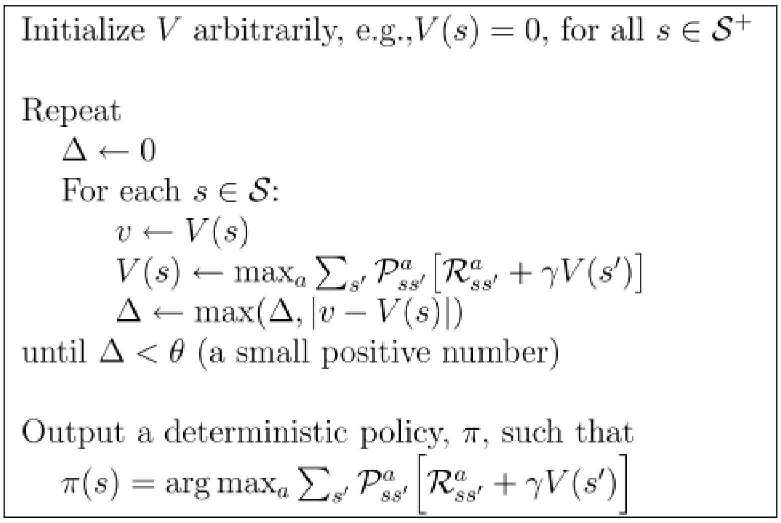
We begin by initializing the values and actions for the policy arbitrarily and afterwards we perform policy evaluation until the difference in the values are smaller than a certain small threshold θ. After this step we begin the policy improvement step where we find the best possible action for each possible state, with respect to maximizing the value. If the resulting actions are the same as the current policy then we have reached the optimal policy, otherwise we perform policy evaluation again on the new policy and keep on the same procedure until we converge to the optimal one.

2.4 Value Iteration

Value iteration extends on Policy iteration in which the policy evaluation step of policy evaluation is stopped after just one sweep, ie there's one backup of each state. It consists of a backup operation that combines the policy improvement and truncated policy evaluation steps:

$$\begin{aligned} V_{k+1}(s) &= \max_a E r_{t+1} + \gamma V_k(s_{t+1}) | s_t = s, a_t = a \\ &= \max_a \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V_k(s')] \end{aligned} \tag{2.5}$$

From this we can see it's identical to the Policy evaluation except that it requires the maximum to be taken over all actions. The algorithm, as previous algorithms converge to an optimal policy for discounted finite MPDs. Formally, convergent to V^* is only reached when the number of iterations is taken to infinity. In practice we consider θ as the termination condition of the loop, which keeps track of the change in value funtion. The pseudocode for value iteration is shown in the following figure:



```
Initialize $V$ arbitrarily, e.g.,$V(s) = 0$, for all $s \in \mathcal{S}^+$

Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi$, such that
    $\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
```

Figure 2.7: Value Iteration [1]

As in our implementation of policy evaluation and policy iteration, the backup operation is done "in-place", thus faster convergent is reached. The rate of convergent hereby is very much dependant on the order in which states are backed up during the sweep. In our implementation the order is straightforward; it's aligned with the array[i][j], starting with the x-coordinate x_i. From the second row forward the benefit hereby is valued at most $3V_k$, leaving at minimum $2V_{k-1}$ still containing the old values, where k denotes the current sweep.

2.5 Reducing the state space

From the values of the table of policy and value iteration when the prey was located at prey(5,5) and the tables from when the prey was at prey(2,3), we can see that those are the same values only proportional to the distance of the predator and the prey, translated to a different position. From this observation we could reduce the state space from 11^4 to 11^2 by creating a transform function which translates the actual positions of the agents to the positions displayed in table 3.5. The transformation taking place can be illustrated in formula **??**, which also takes into consideration that the grid is toroidal. projectedPredatorPosition = projectedPreyPosition - (actualPreyPosition + actualPredatorPosition) The reduced state space results in a lot less memory requirements as well as less computation, which leads us to a faster runtime.

3 Results

3.1 Random Policy

For 100 runs	Summary
Average time to capture the prey	336.78
Standard deviation	285.66324859876534

Table 3.1: Random policy results

3.2 Iterative Policy Evaluation

The goal of the experiment in this algorithm was to evaluate the arbitrary policy, and to measure the number of iteration it took to converge. There were two parameters should be defined, discount factor γ and a threshold value for stopping condition θ. The values of the parameters along with some results are shown in the following table.

	Summary
γ	0.8
θ	1.0E-20
runtime	12248712457 ns
Number of iterations	106

Table 3.2: Policy evaluation results

Runtime is time that the algorithm took to converge, and it was measured in nanosecond. While the number of iterations is the number of sweep (sweep is backup operation for all states) until it converged.

Predator	Prey	$V(s)$
(0,0)	(5,5)	0.005724141401102873
(2,3)	(5,4)	0.18195076385152237
(2,10)	(10,10)	0.18195076385152237
(10,10)	(0,0)	1.1945854778368172

Table 3.3: State values for the following states

In our test environment, the goal of the planning is to make the predator catch the prey as fast as possible. So state-value functions in this case represent how close the predator from the prey in a given state. The closer the predator from the prey, the higher the state-value function should be for that state. Table 3.3 shows the state-value functions for four different states after performing iterative policy evaluation with parameters given in table 3.2.

The results make sense because the state value is proportional to the Manhattan distance between the predator and the prey. Intuitively the predator has a higher chance of catching the prey when it's closer.

3.3 Policy Iteration

For all the experiments with policy iteration theta was defined as $\theta = \exp(-20)$, in order to perform a thorough sanity check, as it was suggested. As far as convergence is concerned it is presented in table 3.4, and as we can see for larger values of γ convergence is slower since more iterations are needed because Δ is smaller and it takes longer to reach θ.

γ	Evaluation runs	Improvement runs
0.1	100	8
0.5	227	7
0.7	323	8
0.9	763	10

Table 3.4: Convergence in iterations for different γ

Following a similar notion as in policy evaluation we present the state-value table in 3.5 for the states that are composed from the prey being at position prey(5,5). As in the case of policy evaluation the state values are proportional to the distance between the predator and the prey. The closer to the prey the higher the value. As we can also see the position of the prey is defined as the goal to be reached, hence the value will be 0 since there aren't successor states and the episode ends.

3.883	4.291	4.742	5.237	5.792	6.251	5.792	5.237	4.742	4.291	3.883
4.291	4.712	5.228	5.802	6.436	6.997	6.436	5.802	5.228	4.712	4.291
4.742	5.228	5.802	6.440	7.148	7.839	7.148	6.440	5.802	5.228	4.742
5.237	5.802	6.440	7.148	7.936	8.780	7.936	7.148	6.440	5.802	5.237
5.792	6.436	7.148	7.936	8.780	10.000	8.780	7.936	7.148	6.436	5.792
6.251	6.997	7.839	8.780	10.000	0.000	10.000	8.780	7.839	6.997	6.251
5.792	6.436	7.148	7.936	8.780	10.000	8.780	7.936	7.148	6.436	5.792
5.237	5.802	6.440	7.148	7.936	8.780	7.936	7.148	6.440	5.802	5.237
4.742	5.228	5.802	6.440	7.148	7.839	7.148	6.440	5.802	5.228	4.742
4.291	4.712	5.228	5.802	6.436	6.997	6.436	5.802	5.228	4.712	4.291
3.883	4.291	4.742	5.237	5.792	6.251	5.792	5.237	4.742	4.291	3.883

Table 3.5: Values from policy iteration when the prey is at [5][5]

3.4 Value Iteration

As in policy iteration, similar tables, namely 3.6, 3.7, for value iteration are presented.

γ	Nr. of iterations
0.1	18
0.5	25
0.7	27
0.9	29

Table 3.6: Convergence in iterations for different γ

3.883	4.291	4.742	5.237	5.792	6.251	5.792	5.237	4.742	4.291	3.883
4.291	4.712	5.228	5.802	6.436	6.997	6.436	5.802	5.228	4.712	4.291
4.742	5.228	5.802	6.440	7.148	7.839	7.148	6.440	5.802	5.228	4.742
5.237	5.802	6.440	7.148	7.936	8.780	7.936	7.148	6.440	5.802	5.237
5.792	6.436	7.148	7.936	8.780	10.000	8.780	7.936	7.148	6.436	5.792
6.251	6.997	7.839	8.780	10.000	0.000	10.000	8.780	7.839	6.997	6.251
5.792	6.436	7.148	7.936	8.780	10.000	8.780	7.936	7.148	6.436	5.792
5.237	5.802	6.440	7.148	7.936	8.780	7.936	7.148	6.440	5.802	5.237
4.742	5.228	5.802	6.440	7.148	7.839	7.148	6.440	5.802	5.228	4.742
4.291	4.712	5.228	5.802	6.436	6.997	6.436	5.802	5.228	4.712	4.291
3.883	4.291	4.742	5.237	5.792	6.251	5.792	5.237	4.742	4.291	3.883

Table 3.7: Values from value iteration when the prey is at [5][5]

3.5 Comparison of Policy and Value iteration

We can see that tables 3.5 and 3.7 have the same values, and that can be explained because both methods converge to the optimal policy. The only difference can be the number of iterations it takes to converge to that optimal policy. Value iteration has only one backup operation for each state and reduces the policy evaluation to one step, whereas policy iteration performs multiple evaluations until it reaches the improvement part. That's why it will converge much faster than policy iteration which can be shown in graph 3.1.

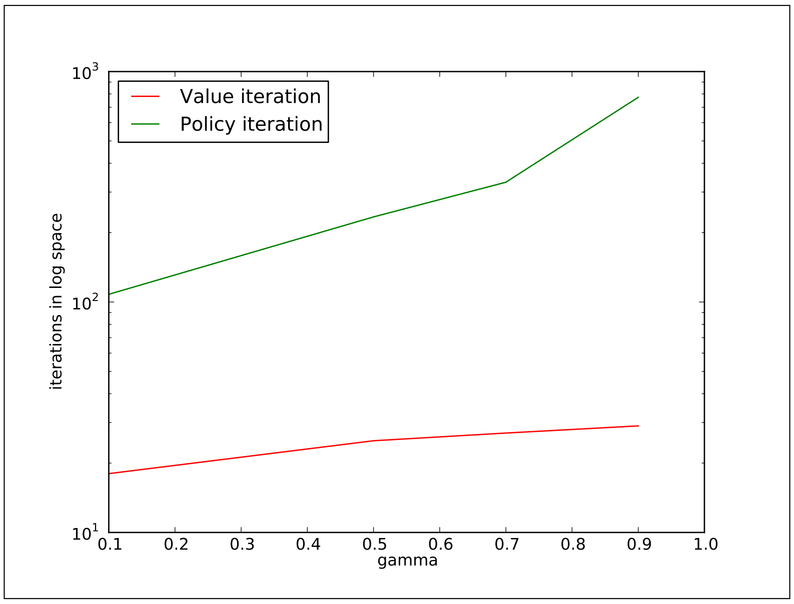


Figure 3.1: Comparative graph of convergence for policy and value iteration

4 CONCLUSION

To conclude we presented the algorithms of policy evaluation, policy iteration, value iteration as well as their results on a predator-prey environment which consists of a toroidal grid. Through the presentation of those results we gained better insights on the advantages and disadvantages of using a specific algorithm. In our assignment value iteration converges faster to the optimal policy than policy iteration, and seems to be better in that regard. However maybe in some specific problems we need a decoupling of policy evaluation from improvement, because further evaluation is not needed. This can only be done with policy iteration.

Additionally, we reduced the statespace from 11^4 states to 11^2 states and we can actually reduce it even more, ie a quarter of that same statespace by employing mirroring. The idea came from the visualization of the distribution of values of the states around the projected prey. It is not as straightforward as the transformation used in the initial reduction because state actions need to be mirrored as well.

we are relied to a complete and finite MDP see the big picture of GPI loop

REFERENCES

[1] Richard S. Sutton, Andrew G. Barto, *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, A Bradford Book, 1998.