CSC490

Reinforcement Learning

Report On Coursework 1

Submitted By:

Tanjina Proma

ID: 1320810

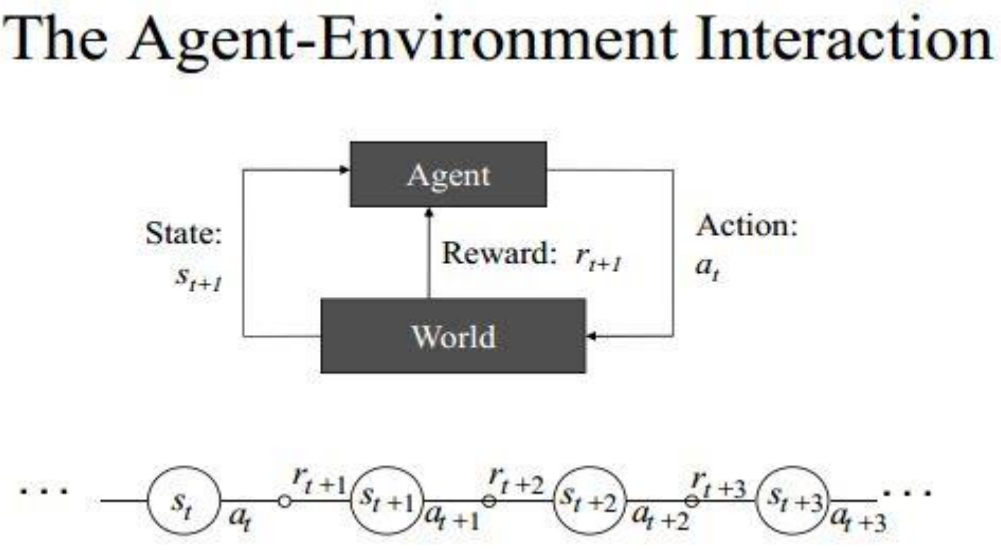
Abstract

Reinforcement learning offers to robotics a framework and set of tools for the design of sophisticated and hard-to-engineer behaviors. Conversely, the challenges of robotic problems provide both inspiration, impact, and validation for developments in reinforcement learning. The relationship between disciplines has sufficient promise to be likened to that between physics and mathematics. In this article, we attempt to strengthen the work in reinforcement learning for behavior generation. We highlight both key challenges in reinforcement learning as well as notable successes. We discuss how contributions tamed the complexity of the domain and study the role of algorithms, representations, and prior knowledge in achieving these successes. As a result, a particular focus of our paper lies on the choice between model-based and model-free as well as between value function-based and policy-based methods. By analyzing a simple problem in some detail we demonstrate how reinforcement learning approaches may be profitably applied, and we note throughout open quest.

Keyword : reinforcement learning, learning control.

1 Introduction

A remarkable variety of problems in robotics may be naturally phrased as ones of reinforcement learning. Reinforcement learning (RL) enables a robot to autonomously discover an optimal behavior through trial-and-error interactions with its environment. Instead of explicitly detailing the solution to a problem, in reinforcement learning the designer of a control task.



In reinforcement learning, an agent tries to maximize the accumulated reward over its life-time. In an episodic setting, where the task is restarted after each end of an episode, the objective is to maximize the total reward per episode. If the task is on-going without a clear beginning and end, either the average reward over the whole life-time or a discounted return (i.e., a weighted

average where distant rewards have less influence) can be optimized. In such reinforcement learning problems, the agent and its environment may be modeled being in a state s ∈ S and can perform actions a ∈ A, A state s contains all relevant information about the current situation to predict future states (or observables); an example would be the current position of a robot in a navigation task1 . An action a is used to control (or change) the state of the system.

For every step, the agent also gets a reward R, which is a scalar value and assumed to be a function of the state and observation. In the direction-finding task, a possible reward could be costs for taken actions and rewards for reaching targets. The goal of reinforcement learning is to find a mapping from states to actions, called policy π, that picks actions a in given states s maximizing the cumulative expected reward. The policy π is either deterministic or probabilistic. The former always uses the exact same action for a given state in the form a = π(s), the later draws a sample from a distribution over actions when it encounters a state, i.e., a ∼ π(s, a) = P (a|s). The reinforcement learning agent needs to discover the relations between states, actions, and rewards. Hence exploration is required which can either be directly embedded in the policy or performed separately and only as part of the learning process. Classical reinforcement learning approaches are based on the assumption that we have a Markov Decision Process (MDP) consisting of the set of states S, set of actions A, the rewards R and transition probabilities T that capture the dynamics of a system. Transition probabilities (or densities in the continuous state case) T(s ′, a, s) = P(s ′ |s, a) describe the effects of the actions on the state. Transition probabilities generalize the notion of deterministic dynamics to allow for modeling outcomes are uncertain even given full state. The Markov property requires that the next state s ′ and the reward only depend on the previous state s and action and not on additional information about the past states or actions. In a sense, the Markov property capitulates the idea of state – a state is a sufficient statistic for predicting the future, rendering previous observations irrelevant. Different types of reward functions are commonly used, including rewards depending only on the current state R = R(s), rewards depending on the current state and action R = R(s, a), and rewards including the transitions R = R(s ′, a, s). Most of the theoretical guarantees only hold if the problem adheres to a Markov structure, however in practice, many approaches work very well for many problems that do not fulfill this requirement.

The goal of reinforcement learning is to discover an optimal policy π ∗ that maps states (or observations) to actions so as to maximize the expected return G_t, which corresponds to the cumulative expected reward. There are different models of optimal behavior, which result in different definitions of the expected return.

$$v\pi(s) = E\pi[Gt|St = s]$$

This State,S may be arbitrarily large, as long as the expected reward over the episode can be guaranteed to converge. In difference to supervised learning, the learner must first discover its environment and is not told the optimal action it needs to take. To gain information about the rewards and the behavior of the system, the agent needs to explore by considering previously unused actions or actions it is uncertain about. It needs to decide whether to play it safe and stick to well known actions with (moderately) high rewards or to dare trying new things in order to discover new strategies with an even higher reward. This

problem is commonly known as the exploration-exploitation trade-off. Off-policy methods learn independent of the employed policy. On-policy methods collect sample information about the environment using the current policy. As a result, exploration must be built into the policy and determines the speed of the policy improvements. Such exploration and the performance of the policy can result in an exploration-exploitation trade-off between long- and short-term improvement of the policy.The agent needs to determine a correlation between actions and reward signals. An action taken does not have to have an immediate effect on the reward but can also influence a reward in the distant future. The difficulty in assigning credit for rewards is directly related to the states or mixing time of the problem. It also increases with the dimensionality of the actions as not all parts of the action may contribute equally.

2.1 Value function

Much of the reinforcement learning literature has focused on solving the optimization problem in Equations. Value functions satisfy recursive relationship (Dynamic Programming).

$$
\begin{aligned}
&v\pi(s) = \mathrm{E}[Gt|St = s] \\
&= \mathrm{E}[Rt{+}1 + \gamma Rt{+}2 + \gamma 2Rt{+}3 + ..|St = s] \\
&= \mathrm{E}[Rt{+}1 + \gamma(Rt{+}2 + \gamma Rt{+}3 + ..)|St = s] \\
&= \mathrm{E}[Rt{+}1 + \gamma Gt|St = s] \\
&= \mathrm{E}[Rt{+}1 + \gamma v(st{+}1)|St = s]
\end{aligned}
$$

Value functions can therefore be decomposed into immediate reward plus discounted value of successor state

$$v\pi(s) = \mathrm{E}\pi[Rt{+}1 + \gamma v\pi(St{+}1)|St = s]$$

Bellman Equation for Value Functions

$v\pi(s) = \mathrm{E}[Rt{+}1 + \gamma v(St{+}1)|St = s]$

Equivalently, we can write the value function as

$v\pi(s) = Rs + \gamma \mathrm{X}\ s0{\in}S\ Pss0v(s0)$.

This statement implies that there are as many equations as the number of states multiplied by the number of actions. For each state there can be one or several optimal actions a $*$ that result in the same maximal value, and, hence, can be written in terms of the optimal action a $*$ as V $\pi *$ (s) = R(s, a$*$) $-$ R^{-} + P s $'$ V $\pi *$ (s $'$)T(s, a$*$, s$'$). As a $*$ is generated by the same optimal policy $\pi *$, we know the condition for the multipliers at optimality is

$$V^*(s) = \max_{a^*}\left[R(s,a^*) - \bar{R} + \sum_{s'} V^*(s')T(s,a^*,s')\right]$$

where V ∗ (s) is a shorthand notation for V π ∗ (s). This statement is equivalent to the Bellman Principle of Optimality (Bellman, 1957)3 that states "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." Thus, we have to perform an optimal action a ∗ , and, subsequently, follow the optimal policy π ∗ in order to achieve a global optimum. When evaluating Equation, we realize that optimal value function V ∗ (s) corresponds to the long term additional reward, beyond the average reward $\bar{R}$, gained by starting in state s while taking optimal actions a ∗ (according to the optimal policy π ∗).long term additional reward, beyond the average reward $\bar{R}$, gained by starting in state s while taking optimal actions a ∗ (according to the optimal policy π ∗).

In contrast to the value function V π (s), the state action value function Q π (s, a) explicitly contains the information about the effects of a particular action. The optimal state-action value function is

$$Q^*(s,a) = R(s,a) - \bar{R} + \sum_{s'} V^*(s')T(s,a,s').$$
$$= R(s,a) - \bar{R} + \sum_{s'} \left(\max_{a'} Q^*(s',a') \right) T(s,a,s').$$

It can be shown that an optimal, deterministic policy π ∗ (s) can be reconstructed by always picking the action a ∗ in the current state that leads to the state s with the highest value V ∗ (s).

$$\pi^*(s) = \arg\max_a \left(R(s,a) - \bar{R} + \sum_{s'} V^*(s')T(s,a,s') \right)$$

If the optimal value function V ∗ (s ′) and the transition probabilities T(s, a, s′) for the following states are known, determining the optimal policy is straightforward in a setting with discrete actions as an exhaustive search is possible. For continuous spaces, determining the optimal action a ∗ is an optimization problem in itself. If both states and actions are discrete, the value function and the policy may, in principle, be represented by tables and picking the appropriate action is reduced to a look-up. For large or continuous spaces representing the value function as a table becomes intractable. Function approximation is employed to find a lower dimensional representation that matches the real value function as closely as possible. Using the state-action value function Q ∗ (s, a) instead of the value function V ∗ (s).

π ∗ (s) = arg max a Q ∗ (s, a) avoids having to calculate the weighted sum over the successor states, and hence no knowledge of the transition function is required. A wide variety of methods of value function based reinforcement learning algorithms that attempt to estimate V ∗ (s) or Q ∗ (s, a) have been developed and can be split mainly into three classes: (i) dynamic programming-based optimal control approaches such as policy iteration or value iteration, (ii) rollout-based Monte Carlo methods and (iii) temporal difference methods such as TD(λ) (Temporal Difference learning), Q-learning, and SARSA (State-Action-Reward-State-Action.)

3. Value iteration and Policy Iteration

Dynamic Programming-Based Methods require a model of the transition probabilities T(s ', a, s) and the reward function R(s, a) to calculate the value function. The model does not necessarily need to be predetermined but can also be learned from data, potentially incrementally. Such methods are called model-based. Typical methods include policy iteration and value iteration. Policy iteration alternates between the two phases of policy evaluation and policy improvement. The approach is initialized with an arbitrary policy. Policy evaluation determines the value function for the current policy. Each state is visited and its value is updated based on the current value estimates of its successor states, the associated transition probabilities, as well as the policy. This procedure is repeated until the value function converges to a fixed point, which corresponds to the true value function. Policy improvement greedily selects the best action in every state according to the value function as shown above. The two steps of policy evaluation and policy improvement are iterated until the policy does not change any longer. Policy iteration only updates the policy once the policy evaluation step has converged. In contrast, value iteration combines the steps of policy evaluation and policy improvement by directly updating the value function based on previous equations every time a state is up.

MATLAB CODE for Value Iteration:

```
function [v, pi] = valueIteration(model, maxit)

% initialize the value function
v = zeros(model.stateCount, 1);
theta = 1.0000e-22;

for i = 1:maxit,
    % initialize the policy and the new value function
    pi = ones(model.stateCount, 1);
    v_ = zeros(model.stateCount, 1);

    % perform the Bellman update for each state
    for s = 1:model.stateCount,
        % COMPUTE THE VALUE FUNCTION AND POLICY
        % YOU CAN ALSO COMPUTE THE POLICY ONLY AT THE END
        p1 = reshape(model.P(s,:,:), model.stateCount, 4);
        [v_(s,:), action] = max(model.R(s,:) + (model.gamma * p1' * v)');
        pi(s,:) = action;
    end
    v1 = v;
    v = v_;

    % exit early
    if v - v1 <= theta
        % CHANGE THE IF-STATEMENT
```

```
% %         fprintf('i = %d', i);
        disp(i);
        break;
    end
end
```

Observation:

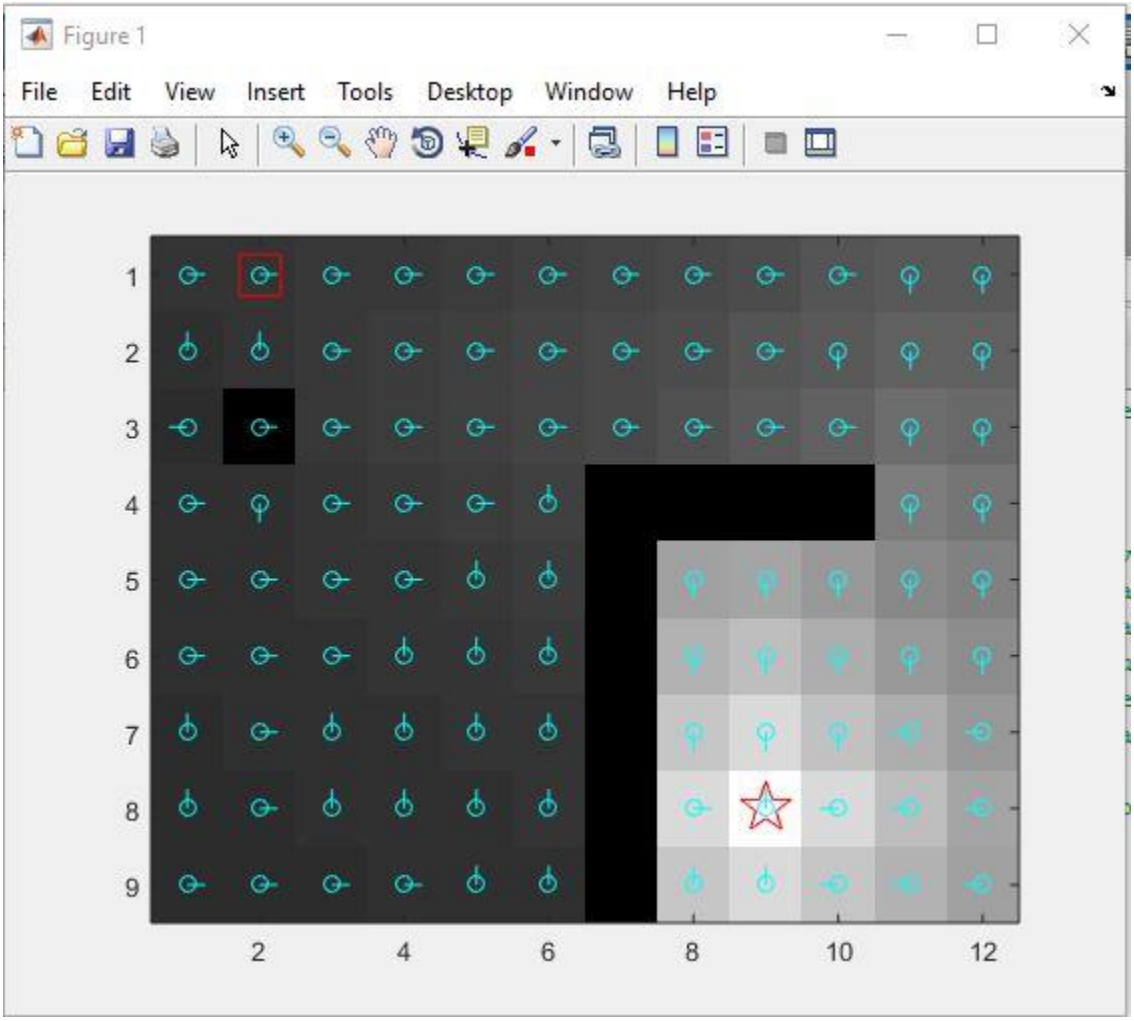


Fig: Value Iteration On Grid world

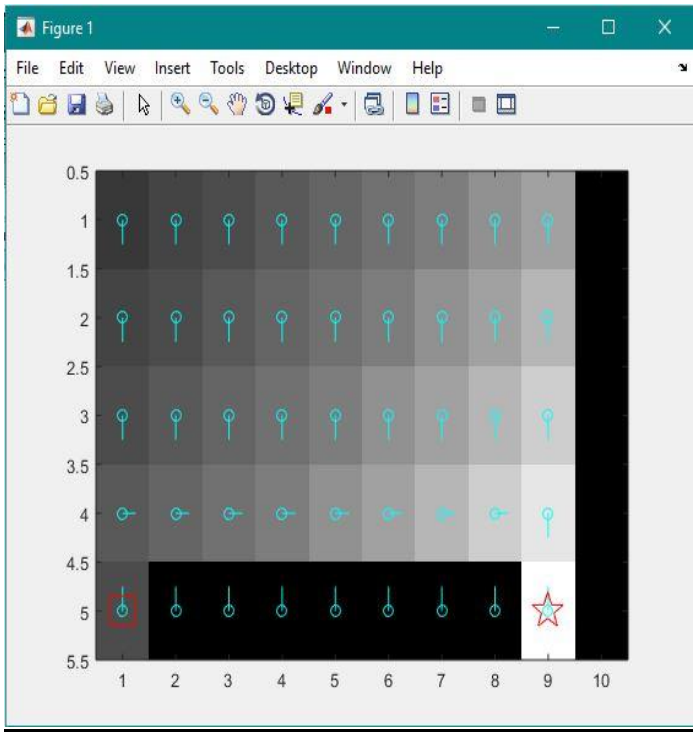


Fig: Value Iteration On cliff world

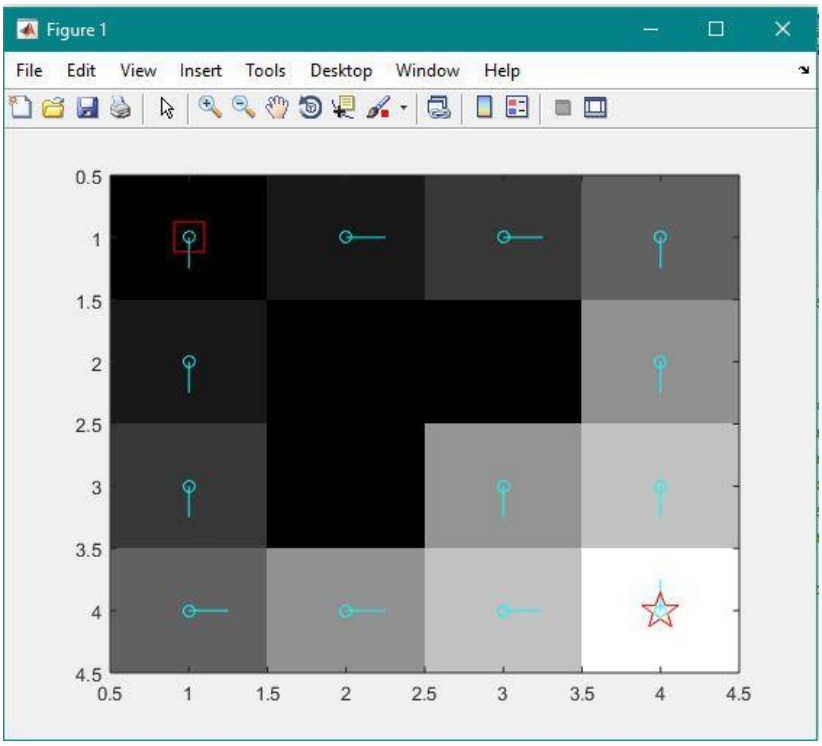


Fig: Value Iteration On small world

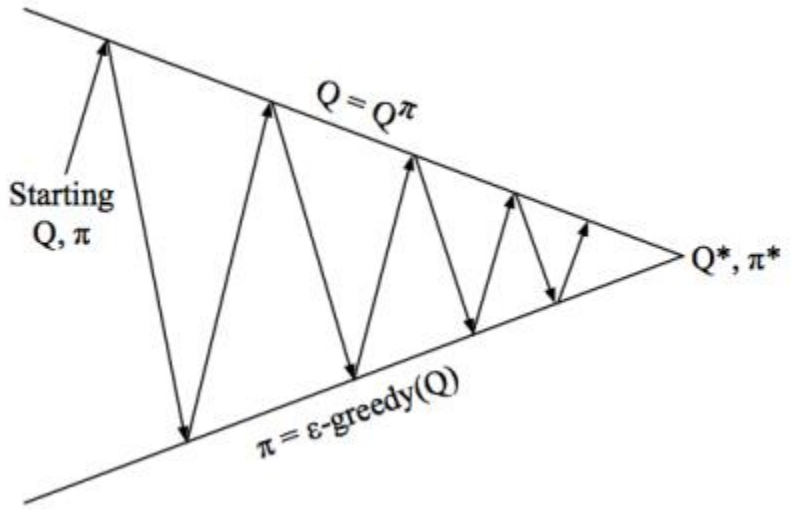
MATLAB CODE for Policy Iteration:

```
function [v, pi] = policyIteration(model, maxit)

% initialize the value function

v=zeros(model.stateCount, 1);
pi = ones(model.stateCount, 1);
old_v = zeros(model.stateCount, 1);
policy = ones(model.stateCount, 1);
theta = 1.0000e-22;

for iterations = 1:maxit,
   % Policy Evaluation Step
   %with same number of episodes
   for i = 1:maxit, v_ = zeros(model.stateCount, 1);
      % perform the Bellman update for each state
      for s=1:model.stateCount
         v_(s) = model.R(s, policy(s)) + (model.gamma*model.P(s,:,policy(s))*v)';
      end
      delta = norm(v - v_);
      v = v_;
      %check for convergence
      if delta <= theta
         fprintf('i = %d', i);
         break;
      end
   end
   for s=1:model.stateCount
      P=reshape(model.P(s,:,:),model.stateCount,4);
      [~, action] = max(model.R(s,:) + (model.gamma *P'*v)');
      policy(s) = action;
   end
end pi = policy; v = v_;
end
```

4. Monte Carlo

Monte Carlo Methods use sampling in order to estimate the value function. This procedure can be used to replace the policy evaluation step of the dynamic programming-based methods above.

Monte Carlo methods are model-free, i.e., they do not need an explicit transition function. They perform rollouts by executing the current policy on the system, hence operating on-policy. The frequencies of transitions and rewards are kept track of and are used to form estimates of the value function. For example, in an episodic setting the state-action value of a given state action pair can be estimated by averaging all the returns that were received when starting from them.



Monte-Carlo Policy Iteration.

Greedy In the Limit With Infinite Exploration Monte-Carlo Policy Control

- Sample kth episode using π: $\{s_1, a_1, r_2, ..., s_T\} \sim \pi$
- For each state s_t and action a_t in the episode,

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{1}{N(s_t, a_t)} (v_t - Q(s_t, a_t))$$

- Improve policy based on new action-value function

$$\epsilon \leftarrow 1/k$$

$$\pi \leftarrow \epsilon\text{-greedy}(Q)$$

5. **Temporal Difference**

Temporal Difference Methods, unlike Monte Carlo methods, do not have to wait until an estimate of the return is available (i.e., at the end of an episode) to update the value function. Rather, they use temporal errors and only have to wait until the next time 7 step. The temporal error is the difference between the old estimate and a new estimate of the value function, taking into account the reward received in the current sample. These updates are done iterativley and, in contrast to dynamic programming methods, only take into account the sampled successor states rather than the complete distributions over successor states. Like the Monte Carlo methods, these methods are modelfree, as they do not use a model of the transition function to determine the value function. In this setting, the value function cannot be calculated analytically but has to be estimated from sampled transitions in the MDP. For example, the value function could be updated iteratively by

$$V'(s) = V(s) + \alpha \left(R(s,a) - \bar{R} + V(s') - V(s) \right),$$

where V (s) is the old estimate of the value function, V ′ (s) the updated one, and α is a learning rate. This update step is called the TD(0)-algorithm in the discounted reward case. In order to perform action selection a model of the transition function is still required. The equivalent temporal difference learning algorithm for state-action value functions is the average reward case version of SARSA with

$$Q'(s,a) = Q(s,a) + \alpha \left(R(s,a) - \bar{R} + Q(s',a') - Q(s,a) \right),$$

where Q(s, a) is the old estimate of the state-action value function and Q ′ (s, a) the updated one. This algorithm is on-policy as both the current action a as well as the subsequent action a ′ are chosen according to the current policy π. The off-policy variant is called R-learning

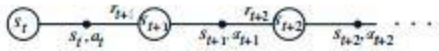
$$Q'(s,a) = Q(s,a) + \alpha \left(R(s,a) - \bar{R} + \max_{a'} Q(s',a') - Q(s,a) \right).$$

These methods do not require a model of the transition function for determining the deterministic optimal policy π ∗ (s). it is a related method that estimates a model of the transition probabilities and the reward function in order to perform updates that are reminiscent of value iteration. An overview of publications using value function based methods Here, model based methods refers to all methods that employ a predetermined or a learned model of system dynamics

6. Sarsa

• Transitions from non-terminal states update Q as follows:

$$Q(st, at) \leftarrow Q(st, at) + \alpha[rt+1 + \gamma Q(st+1, at+1) - Q(st, at)]$$



(for *terminal states* s_{t+1}, assign $Q(s_{t+1}, a_{t+1}) \leftarrow 0$)

Algorithm 2: Sarsa

```
Initialise Q(s,a) arbitrarily
for each episode
  initialise s
  choose a from s using π derived from Q /* e.g. ε-greedy */
  repeat (for each step of episode)
    perform a, observe r, s'
    choose a' from s' using π derived from Q /* e.g. ε-greedy */
    Q(s,a) ← Q(s,a) + α[r + γQ(s',a') − Q(s,a)]
    s ← s'
    a ← a'
  until s is terminal state
```

MATLAB Code for SARSA:

```
function [v, pi,accumulated_reward] = sarsa(model, maxit, maxeps)

% initialize the value function
Q = zeros(model.stateCount, 4);
pi = ones(model.stateCount, 1);
pi_changing = ones(model.stateCount, 1);
Alpha =0.1;
accumulated_reward = zeros(maxeps, 1);
for i = 1:maxeps,
   % every time we reset the episode, start at the given startState
   s = model.startState;
  % a = 1;
   a =  e_greedy(Q(s,:));
   for j = 1:maxit,
      % PICK AN ACTION
      %     a = 1;
      p = 0;
      r = rand;

      for s_ = 1:model.stateCount,
         p = p + model.P(s, s_, a);
         if r <= p,
            break;
         end
      end

      % s_ should now be the next sampled state.
      % IMPLEMENT THE UPDATE RULE FOR Q HERE.
      %..... Q(s,a) = Q(s,a) + Alpha[ R + Gamma * Q(s_, a_)  - Q(s,a)]
```

```
        reward = model.R(s,a);
            accumulated_reward(i) = accumulated_reward(i) + model.gamma * reward;
        %...greddy for a_.....................

        %[~, max_action_a_] = max(Q(s_,:));
        %pi_changing(s) = max_action;
        %  v_changing_s_ = Q( : ,max_action_a_);
        %.............................................
        a_ =  e_greedy(Q(s_,:));
        Q(s,a) = Q(s,a) + Alpha * [ reward + model.gamma  * Q(s_, a_)  - Q(s,a)];
        s = s_;
        a = a_;

        [~, max_action] = max(Q(s,:));
        pi_changing(s) = max_action;
        v_changing = Q( : ,max_action);

        % SHOULD WE BREAK OUT OF THE LOOP?

        if s == model.goalState
            break;
        end
    end
end

% REPLACE THESE
v = v_changing;
pi = pi_changing;

end
```

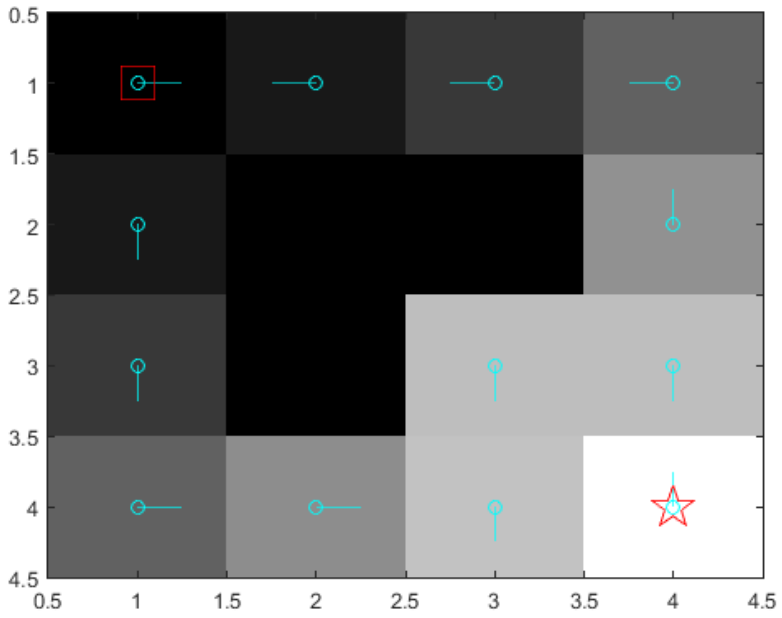
Function for Epsilon Greedy Policy:

```
function a = e_greedy(Q)
e = 0.1;
actions = [1 2 3 4];
p = rand();
    if p < (1 - e)
        [~, a] = max(Q);
    else a = actions(randi(length(actions)));
    end
end
```

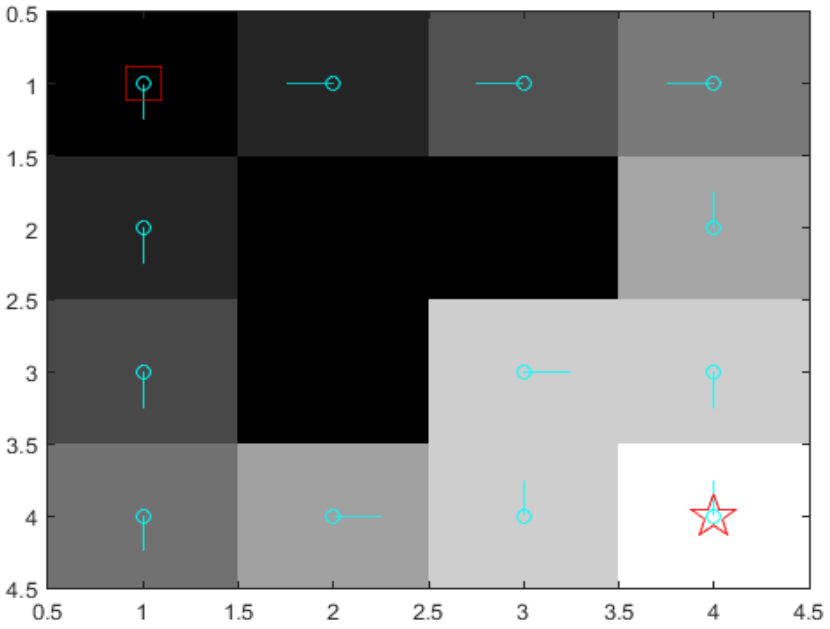
Observation:

The key observation for our purposes is that, since Sarsa is an on-policy method, it does not suffer from the problem of positive bias. Since updates are based on the actions that are actually taken, rather than on the best possible action, we expect Sarsa based modules to discover Q-values that are closer to the true expected return under the composite policy. Any of the action selection mechanisms could be recast to use Sarsa rather than Q-learning to train the modules. Recall that the goal is to maximize the summed reward across all of the component MDPs. Assuming that we have trustworthy utility estimates from each of the modules, it makes sense to choose the action that has the highest summed utility across all of the modules. By definition, this is the action that will lead to the greatest summed long term reward. This reasoning did not hold under Q-learning, because the utility estimates were inaccurate under the composite policy.

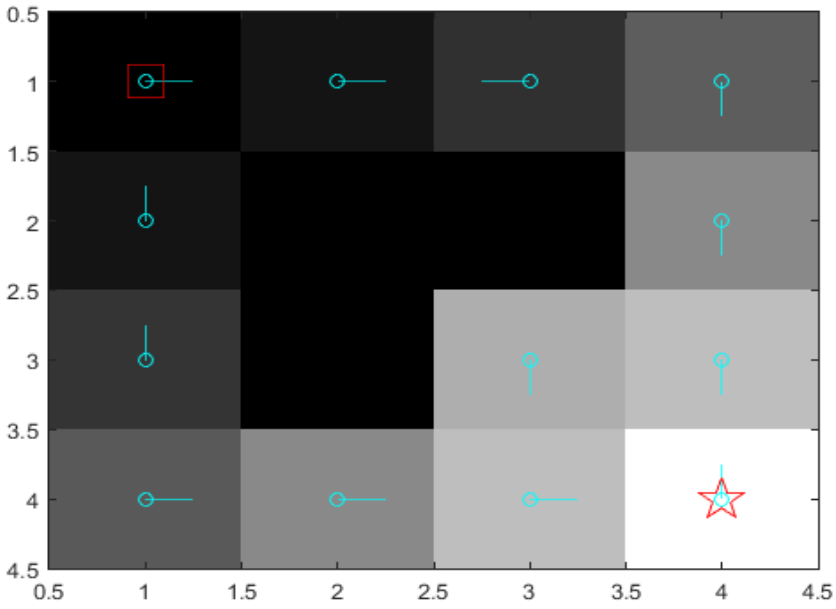
Alpha = 0.001 & Epsilon = 0.0001



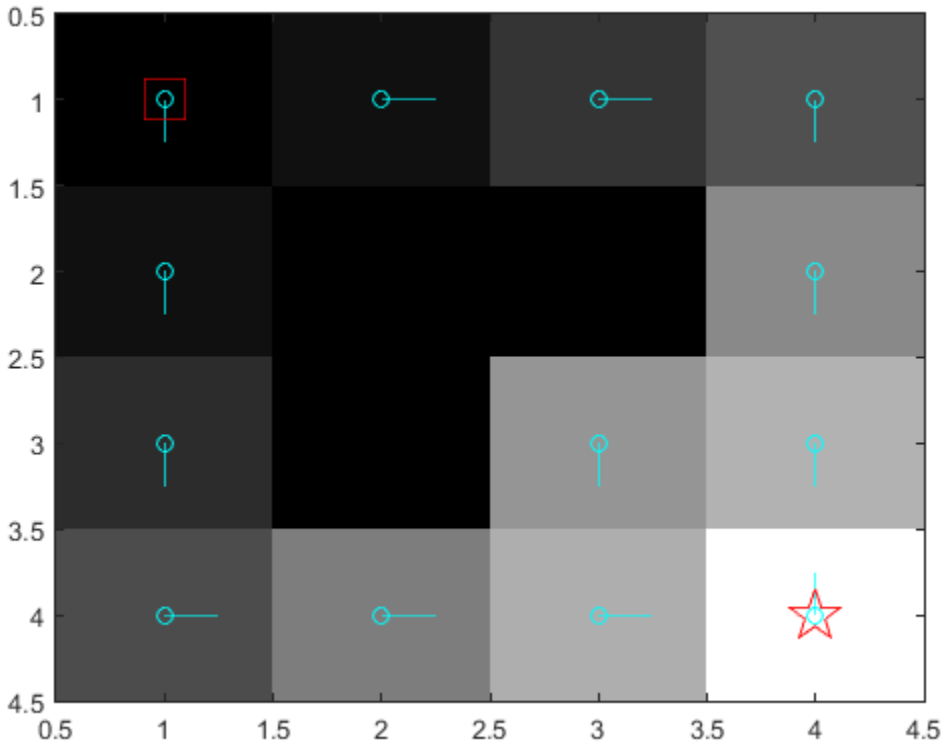
Alpha = 0.0001 & Epsilon = 0.0001



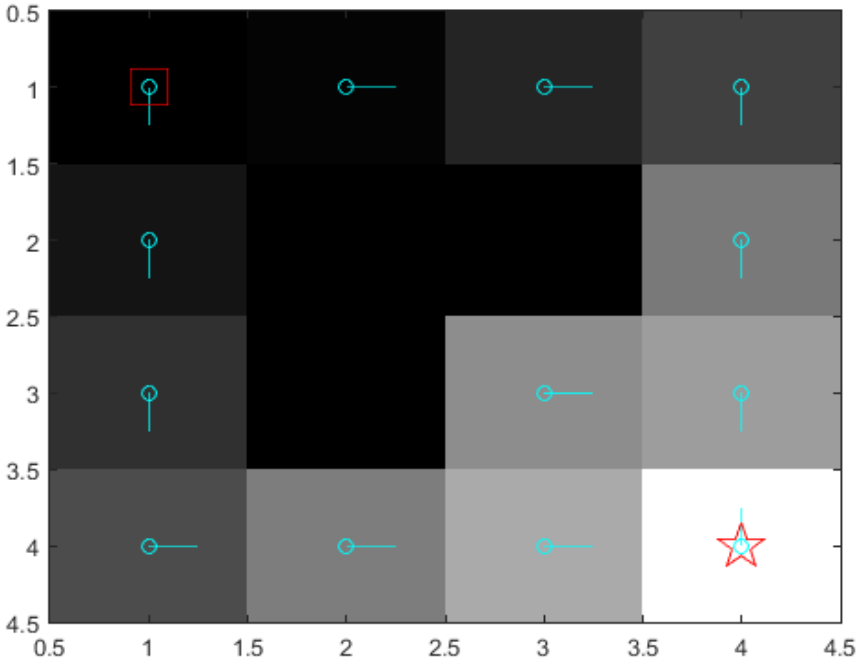
Alpha = 0.01 & Epsilon = 0.0001



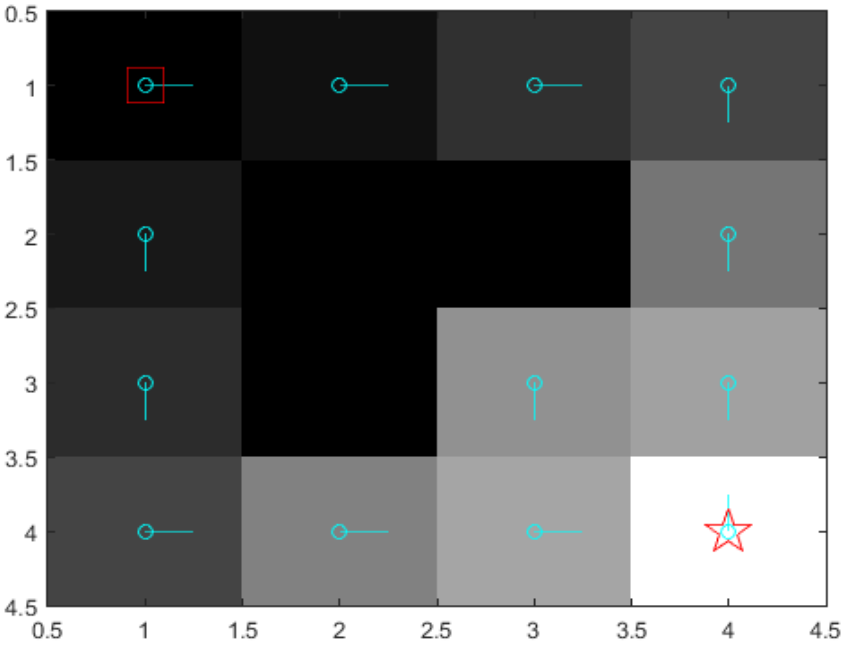
Alpha = 0.1 & Epsilon = 0.0001



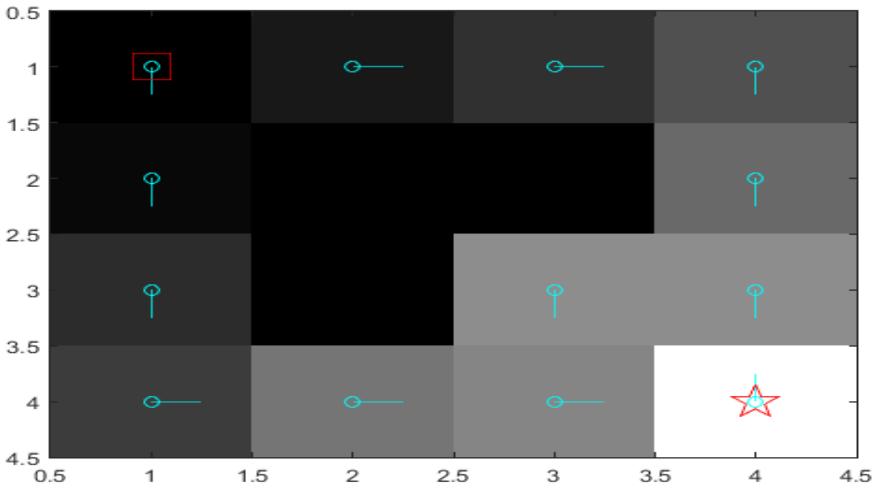
Alpha = 0.1 & Epsilon = 0.001



Alpha = 0.1 & Epsilon = 0.01



Alpha = 0.1 & Epsilon = 0.1



With the increasing value of Alpha and Epsilon greedy values, we get better results.

7. Q-learning

An off-policy method: approximate Q ∗ independently of the policy being followed:

Q(st, at) ← Q(st, at) + α[rt+1 + γ max a Q(st+1, a) − Q(st, at)]

Algorithm 3: Q-Learning

```
Initialise Q(s,a) arbitrarily
for each episode
  initialise s
  repeat (for each step of episode)
    choose a from s using π derived from Q /* e.g. ε-greedy */
    perform a, observe r,s'
    Q(s,a) ← Q(s,a) + α[r + γ max_a' Q(s',a') − Q(s,a)]
    s ← s'
  until s is terminal state
```

MATLAB Code for Q Learning:

```
function [v, pi,accumulated_reward] = qLearning(model, maxit, maxeps)

% initialize the value function
Q = zeros(model.stateCount, 4);
pi = ones(model.stateCount, 1);

pi_changing = ones(model.stateCount, 1);
accumulated_reward = zeros(maxeps, 1);
for i = 1:maxeps,
    % every time we reset the episode, start at the given startState
    s = model.startState;
```

```
    a = 1;
    for j = 1:maxit,
        % PICK AN ACTION

        p = 0;
        r = rand;

        for s_ = 1:model.stateCount,
            p = p + model.P(s, s_, a);
            if r <= p,
                break;
            end
        end

        % s_ should now be the next sampled state.
        % IMPLEMENT THE UPDATE RULE FOR Q HERE.
%       a_ = epsilon_greedy_policy(Q(s, :), j);
        a_ = e_greedy(Q(s, :));
        reward = model.R(s,a);
        accumulated_reward(i) = accumulated_reward(i) + model.gamma * reward;
        Alpha = 1/j ;

        Q(s,a) = Q(s,a) + Alpha * [ reward + model.gamma  * max( Q(s_, a_) ) - Q(s,a)];
        s = s_;
        a = a_;

        [~, max_action] = max(Q(s,:));
        pi_changing(s , : ) = max_action;
        v_changing = Q( : ,max_action);

        % SHOULD WE BREAK OUT OF THE LOOP?

        if s == model.goalState
            break;
        end
    end
end

% REPLACE THESE
v = v_changing;
pi = pi_changing;

end
```

Observation:

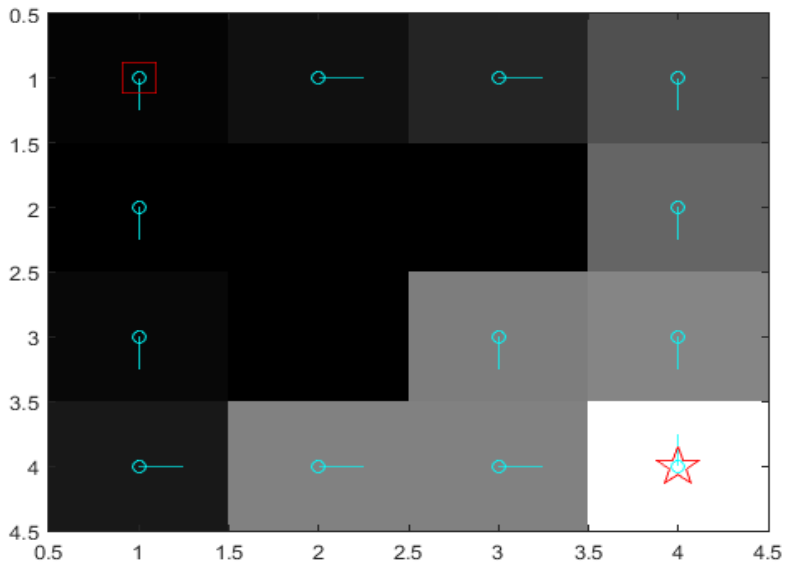


Fig: Q learning on Smallworld.

7.1 convergence

The drawback of switching from Q-Learning to Sarsa based modules is that it becomes more difficult to prove convergence results. The following four possibilities characterize the convergence guarantees that we might seek.

1) The Q-values converge to a bounded region.

2) The Q-values converge to a fixed point.

3) The Q-values converge to a local optimum.

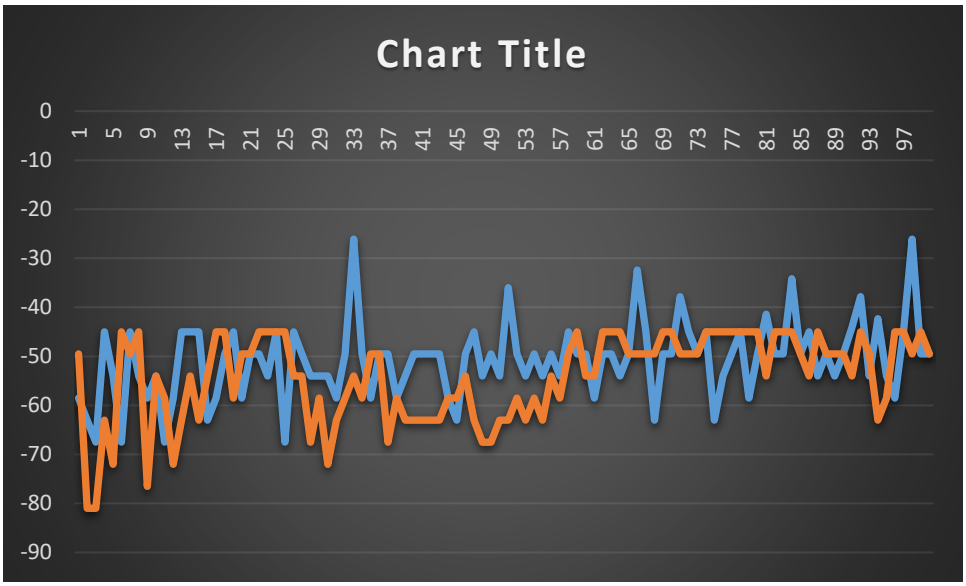
4) The Q-values converge to a global optimum. By local optimum we mean that no module can improve its policy given the current policies of the other modules. By global optimum we mean that the Q-values result in the optimum policy for the composite MDP.

At present,1) is known to be true, 2) and 3) are both unknown, and it is not difficult to contrive a counter-example to demonstrate that 4) is false.

The perception is that even though many policies may be explored during training, each Sarsa update moves the Q-values closer to the correct values for some policy. As long as rewards are bounded, the correct Q-values for all policies must also be bounded and lie within some region. This is a very weak result. It provides no guarantees concerning the performance of the policies that Sarsa discovers.

Thus far, we have not been able to prove that Sarsa necessarily converges to a locally optimal policy, or even that it is guaranteed to converge to any fixed point at all. However, as the examples in the next section demonstrate, Sarsa does appear to converge to good policies for some problems.

Accumulative Reward:



Orange line is presenting Q Learning Values. And Blue Line is presenting the SARSA values. We used 50 episodes and 100 iterations or steps within each episode for both SARSA and Q-Learning for all the 50 iterations. The epsilon value 0.1 ensures a balanced relation between exploration and exploitation. We used 100 iterations since the cumulative reward scale depends on the number of iterations within each episode (reward of 1 for every transition). And the reward for SARSA is better than Q learning.

References

[Dietterich, 2000] T. G. Dietterich, "Hierarchical reinforcement learning with the MAXQ value function decomposition," Journal of Artificial Intelligence Research, 13, 2000.

[Gordon, 2000] G. J. Gordon, "Reinforcement Learning with Function Approximation Converges to a Region," In Advances in Neural Information Processing Systems, volume 13, 2000.

[Singh and Sutton, 1996] S. Singh and R. Sutton, "Reinforcement Learning with Replacing Eligibility Traces," Machine Learning, 22(1-3), 1996.