**Implementation of Reinforcement Learning Algorithms**

1. **Q Learning , Sarsa, Temporal Difference**
2. **Mathematical modeling of above algorithms**

www.actiongeneration.googlecode.com

Consider the example market chain that has hundreds of stores all over a country selling thousands of goods to millions of customers. The point of sale terminals record the details of each transactions i.e. date, customer identification code, goods bought and their amount, total money spent and so forth. This typically amounts to gigabytes of data every day. What the market chain wants is to be able to predict who are the likely customers for a product. Again, the algorithm for this is not evident; it changes in time and by geographic location. The stored data becomes useful only when it is analyzed and turned into information that we can make use of for example to make predications.

We do not know exactly which people are likely to by this product, or another product. If we knew, we would not need any analysis of the data. But because we do not know, we can only collect data and hope to extract the answers to questions from data.

We do believe that there is a process that explains the data we observe. Though we do not know the details of the process underlying the generation of data – for example, customer behavior - we know that it is not completely random. People do not go to markets and by things at random. When they buy beer, they buy chips; they buy ice cream in summer and spices for Wine in winter. There are certain patterns in the data. We may not be able to identify the process completely, but we believe we can construct a *good and useful approximation*. That approximation may not explain everything, but may still be able to account for some part of the data. We believe that though identifying the complete process may not be possible, we can still detect patterns or regularities. Such patterns may help us understand the process, or make predictions. Assuming that the near future will not be much different from the past the future predictions can also be expected to be right.

In some applications, the output of the system is a sequence of actions. In such a case, a single action is not important; what is important is the policy that is the sequence of correct actions to reach the goal. There is no such thing as the best action in any intermediate state; an action is good if it is part of a good policy. In such a case, machine learning program should be able to assess the goodness of policies and learn from past good action sequences to be able to generate a policy. Such learning methods are called reinforcement learning algorithms.

**Algorithms:**

1. One step Q Learning (off policy)
2. One step Sarsa (on policy)
3. Watkins’s Q(**λ**)
4. Sarsa(**λ**)

**Model design:**

**Goal :** Maximize the sale of products that depends on price of product, customer age and period of sale. These are the information available to each agent i.e. shop. So it becomes the state of environment. Final result is to maximize profit by increasing total sale of products.

**State & Action set:**

We define the action set as the sale of possible product. i.e. A={p1,p2,p3…….p10}

Hence action a ****A. State of the system is queue of customer in the particular month for the given shop agent. So state can be described as

X(t) = { x1(t), x2(t),m }

where x1 🡪 customer queue with age ==> { Y, M, O } i.e. young, middle and Old age customer

x2 🡪 price of product queue ==> { H, M, L } i.e. High, Medium, Low

m 🡪 month of product sale ==> { 1,2,3,4…………..12 }

In the system minimum 108 states and actions are possible. Number of state-action increases as number of transactions increases. For simplicity it is assumed that single state for each transaction else the state space becomes infinitely large. Shop agent observes the queue and decides product i.e. action for each customer/state. After every sale reward is given to the agent. Table shows the snapshot of the dataset generated for single agent / shop.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Transaction ID | Age | Price | Month | Action Selected (Product) |
| 1 | Y | L | 1 | P1,P2,P4 |
| 2 | Y | M | 1 | P2,P3 |
| 3 | Y | H | 1 | P3,P4 |
| 4 | M | L | 1 | P1,P2 |
| 5 | M | M | 1 | P1,P2,P3 |
| 6 | M | H | 1 | P4,P2 |
| 7 | O | L | 1 | P1,P3 |

At time 0, the process X(t) is observed and classified into one of the states in the possible set of states (denoted by S). After identification of state the agent/shop chooses a product action from A.

If the process is in state i and agent chooses a ****A, then

1. The process transition into state j****S with probability Pij(a)
2. And further, conditional on the event that the next sate is j, the time until next transition is a random variable with probability distribution Fij(./a)

After the transition occurs, product sale action is chosen again by the agent and (i) and (ii) are repeated.

**Action selection:**

An important component of Q learning is the action selection mechanism. This mechanism is responsible for selecting the actions that the agent will perform during the learning process. Its purpose is to harmonize the trade-off between exploitation and exploration such that the agent can reinforce the evaluation of the actions it already knows to be good but also explore new actions.

It is common in Q learning to use a probabilistic approach for action selection. Actions with higher Q values are assigned higher probabilities, but every action is assigned a nonzero probability. **ε-**greedy exploration mechanism is considered for action selection. This mechanism selects a random action with probability **ε** and the best action i.e. the one that has the highest Q value at the moment, with probability 1- **ε**.

As such it can be seen as defining a probability vector over the action set of the agent for each state. Let x={x1,x2…xi} be one of these vectors, then the probability xi of selecting action i is given by

xi= (1- **ε**) + (**ε** / n)…………….if Q of i is the highest

= **ε** / n……………………….otherwise

where n is the number of actions in the set.

One way to assign such probabilities is

P(ai/s) = KQ’(s,ai) / ∑j.KQ’(s,aj)

P(ai/s)🡪 probability of selecting action ai

s🡪 current state

K🡪 constant >0. High value of K assign high probabilities to action i.e. maximum reward and small value of K assign higher probabilities to other action i.e. minimum reward.

Q learning is off-policy method as the value of the best next action is used without using the policy.

**Reward Model :**

Reward is goal in Q learning. Therefore profit maximization is depend upon increase in total sale of product which a result of more price earned by each product. So reward is calculated based on the frequency of product(s). Also frequency of product sold in association with each other is also helpful in giving the useful information.

Action set A = sale of products P1, P2…..P10. In particular state, product Pi is selected as action i.e. sale of product Pi has been done. So system changes its state from current state i to next state j upon action selected and reward is assigned depending on the action.

Each state comprises of set of age, price, and month. Frequency of particular product is calculated for particular state.

Let state i = {age, price, month} = { x1, x2, m} = {M, H, 7} and

state j = {age, price, month} = { x1’, x2’, m’}= {Y, H, 2}

Step 1: Frequency of product is total number of times particular products exist for given state in

the dataset. So product frequencies for state i and state j are defined as Pfi(0…9) andPfj(0…9) and calculated as

Pfi(0…9) =∑ ip0, ∑ ip1 ……

Pfj(0…9) =∑ jp0, ∑ jp1 ……

Step 2: Take union of two states and sum of each product frequencies in current state and next

state as Pfi(0…9) Pfj(0…9)

sum of product frequency is given as

Pfsum(0….9)= ∑ Pfi(0…9) Pfj(0…9) = Pi0+Pj0,Pi1+Pj1………

Step 3: If state transition is from state i to state j then relative frequency of each product is

calculated as

Prfj = Pfj(0…9) / Pfsum(0…..9)

If state transition is from state j to state i then relative frequency of each product is

calculated as

Prfi = Pfi(0…9) / Pfsum(0…..9)

Step 4: Weight is obtained by summation of relative frequency of products for the respective

state.

wrf = ∑Prfi = rfi1 + rfi2 + rfi3…….

//to be implemented ….

If price of current state i is less than price of next state j then price weight is considered as 1. If price amount in current state i is greater than price of state j then price weight is considered as -1.

wp = 1 ……for x2 < x2’

wp = -1 ……for x2 > x2’

wp= 0 ………otherwise

finally reward is calculated as

Reward r = wrf + wp

where = 0.5 = 0.5

This reward r is assigned for each state action pair and used for further learning.

**e. g.**

The frequencies of product Si = { P0:10, P1:12, P2:11} and in state j product frequencies are Sj = { P0:10,P2:9,P3:12}. To construct the global state, union of state (Su) for each product frequencies has been find out as

Su = Si U Sj = { P0:20, P1:12, P2:20,P3:12 }

Reward is calculated on conditional probability as

If it changes from state i to state j then the probability is

P(Si | Sj) = Sj / Su = { P0:10,P2:9,P3:12} / { P0:20, P1:12, P2:20,P3:12 }

= {P0:10 / P0:20}, {P1:0 / P1:12}, {P2:9 / P2:20}, {P3:12 / P3:12}



= {P0:0.5} + {P1:0} + {P2:0.45} + {P3:1.0}

=0.5+0.0+0.45+1.0

=1.95

Reward 1.95 assigned for state change from i to state j.

If it changes from state j to state i then the probability is

P(Si | Sj) = Sj / Su = { P0:10, P1:12, P2:11} / { P0:20, P1:12, P2:20,P3:12 }

= {P0:10 / P0:20}, {P1:12 / P1:12}, {P2:11 / P2:20}, {P3:0 / P3:12}



= {P0:0.5} + {P1:1} + {P2:0.55} + {P3:0.0}

=0.5+1.0+0.55+0.0

=2.05

Reward 2.05 assigned for state change from j to state i.

If the price in the next state is equal to the price in current state then reward is equal to action.

If price in the next state is more than price in current state then reward is two times the action and reward is 0 for all other cases.

\*\*\*\*\*\*\*\*\*\*\*\*

**Nondeterministic rewards and actions:**

The policy **π** defines the agent’s behavior and is a mapping from the states of the environment to actions. **π : s🡪a** . The policy defines the action to be taken in any state. The value of a policy **π**, V**π**(st) is the expected cumulative reward that will be received while the agent follows the policy, starting from state st.

The problem is modeled using a Markov decision process (MDP). The rewards and the result of actions are not deterministic so it has a probability distribution for the reward p(rt+1|st,at) from which rewards are sampled and there is a probability distribution for the next state P(st+1|st,at). These help to model the uncertainty in the system that may be due to forces we cannot control in the environment. Q learning algorithm is used for this purpose.

*initialize all Q(s, a) to 0 arbitrarily*

*for all episodes*

*initialize s*

*repeat*

*choose a using policy derived from Q e.g.* *******-greedy policy*

*take action a, observe r and s’*

*update Q(s, a) as*

*Q(s, a)🡨 Q(s, a) +α (r +* ***γ*** *maxa Q(s’, a’) – Q(s, a))*

*s🡨s’*

*until s is terminal state*

Discount rate parameter0 ≤ **γ** < 1. It is considered as **γ** = 0.9.

If **γ** =0, then only the immediate reward counts. As **γ** approaches 1, rewards further in the future count more, and it is said that agent becomes more farsighted. **γ** is less than 1 because there generally is a time limit to the sequence of actions needed to solve the task.

Value of learning rate parameter **α** is gradually decreased in time for convergence and it has been shown that this algorithm converges to the optimal Q values.

Initially all Q(s, a) are 0 and they are updated in time as a result of trial episodes. Let us say we have a sequence of moves and at each move, we use above equation to update the estimate of Q value of the previous state-action pair using the Q value the current state-action pair. In the intermediate states, all rewards and therefore values are 0, so no update is done. When we get to the goal state, we get the reward r and then we can update the Q value of the previous state-action pairs as **γ**r.

As for the preceding state-action pair, its immediate reward is 0 and the contribution from the next state-action pair is discounted by **γ** because it is one step later. Then in another episode, if we reach this state, we can update the one preceding that as **γ2r** and so on.

This way, after many episodes, this information is backed up to earlier state-action pairs. Q values increase until they reach their optimal values as we find paths with higher cumulative reward. We do not know the reward or next state functions here. They are part of the environment.

**Steps in Q Learning algorithm implemented:**

Below steps are followed as actual implementation of learning update rule involved in Q learning for continuous time MDP.

Let t0=0 and start with an initial arbitrary guess Q(s, a) = 0.

**Step 1:** At any nth transition epoch at time tn, observe the state s and select the product action a **є** argmaxa Q(s, a) with probability 1-**ε** and other product in A with probability **ε** for some ε>0.

**Step 2:** If X(tn)=s and the product action chosen is a then update its Q value as follows:

*Q(s, a)🡨 Q(s, a) +α (r + γ maxa Q(s’, a’) – Q(s, a))*

To explore, one possibility is to use *******-greedy policy* search where with probability ****** we choose one action *uniformly randomly* among all possible actions i.e. explore and with probability 1 -***,***

we choose the *best* action i.e. exploit. We do not want to continue exploring indefinitely but start exploiting once we do enough exploration. For this, we start with a high ****** value and gradually decrease it.

Repeat steps 1 & 2 infinitely. Convergence is slow as it is typical RL algorithm. The speed of convergence can be drastically improved using function approximations to Q-values based on some observed features.

We tabulate the Q function values for obtaining some insights. Table 1 shows the best action (that is optimal product) for different individual states. By knowing the Q function, the shop agent can compute best possible product for a given state that give maximum profit to it.

We have shown how a shop agent can effectively use reinforcement learning in setting products dynamically so as to maximize its profit matrix. We believe this is a promising approach for profit maximization in retail market environments with limited available information. The next model is to consider two shop agents and find out the results.

**Sarsa Algorithm**

Sarsa is an on policy version of Q learning where policy is used to determine also the next action. Instead of looking for all possible next actions and choosing the best, the on policy Sarsa uses the policy derived from Q values to choose one next action a and uses its Q value to calculate the temporal difference. On policy methods estimate the value of a policy while using it to take actions. They approximate Q value, the action values for current policy, then improve the policy gradually based on the approximate values for the current policy. The policy improvement can be done in a simplest way using **ε-**greedy policy with respect to current action value estimation.

Sarsa learning algorithm is used for this purpose.

*initialize all Q(s, a) to 0 arbitrarily*

*for all episodes*

*initialize s*

*repeat*

*choose a using policy derived from Q e.g.* *******-greedy policy*

*take action a, observe r and s’*

*update Q(s, a) as*

*Q(s, a)🡨 Q(s, a) +α (r +* ***γ****Q(s’, a’) – Q(s, a))*

*s🡨s’*

*a🡨a’*

*until s is terminal state*

**Eligibility Traces**

Eligibility traces are one of the basic mechanisms of reinforcement learning. Q learning or Sarsa can be combined with eligibility traces to obtain a more general method that learns more efficiently. Eligibility trace is a temporary record of the occurrence of an event, such as the visiting of a state or the taking of an action. The trace marks the memory parameters associated with the event as eligible for undergoing learning changes.

**Sarsa(λ)**

The eligibility trace version of Sarsa is called as Sarsa(λ). The idea in Sarsa(λ) is to apply the TD(λ) prediction method to state action pairs rather than to states. Let et(s, a) denote the trace for state action pair s, a; substituting state action variables for state variables the equation becomes

Qt+1 = Qt(s,a) + αδtet(s, a) for all s, a

where

δt = rt+1 + γQt(st+1, at+1) – Qt(st, at)

and

et(s, a) = γλet-1(s, a) + 1 if s=st and a=at

= γλet-1(s, a) otherwise

One step Sarsa and Sarsa(λ) are on policy algorithms. The one step method strengthens only the last action of the sequence of actions that led to the high reward, whereas the trace method strengths many actions of the sequence. The degree of strengthening falls off (according to γλ) with steps from the reward.

**Sarsa(λ) algorithm**

*initialize Q(s, a) arbitrarily and e(s, a) = 0 for all s, a*

*repeat for each episode*

*initialize s, a,*

*repeat for each step of episode*

*take action a, observe r, s’*

*choose a’ from s’ using policy derived from Q (e.g. ε greedy)*

*δ🡨 r + γQ(s’, a’) – Q(s, a)*

*e(s, a)🡨 e(s, a) + 1*

*for all s, a*

*Q(s, a)🡨 Q(s, a) + αδe(s, a)*

*E(s, a)🡨 γλe(s, a)*

*s🡨 s’; a🡨a’*

*until s is terminal*

**Q(λ)**

Q learning is an off policy method, meaning that the policy learned about need not be the same as the one used to select actions. In particular, Q learning learns about the greedy policy while it typically follows a policy involving exploratory actions. Because of this, special care is required when introducing eligibility traces.

Watkins’s Q(λ) does not look ahead all the way to the end of the episode in its backup. It only looks ahead as far as the next exploratory action. Watkins’s Q(λ) looks one action past the first exploration using its knowledge of the action values. Eligibility traces are used just as in Sarsa(λ), except that they are set to zero whenever an exploratory (non-greedy) action is taken. The trace update is thought of as occurring in two steps. First, the traces for all state action pairs are either decayed by γλ or if an exploratory action was taken, set to 0. Second, the trace corresponding to the current state and action is incremented by 1. The overall result is   
et(s, a) = sst. aat + γλet-1(s, a) if Qt-1(st, at) = maxaQt-1(st, at)

= 0 otherwise

whereas before xy is an identity indicator function, equal to 1 if x = y and 0 otherwise. The rest of the algorithm is defined by

Qt+1(s, a) = Qt(s, a) + αδtet(s, a)

where

δt = rt+1 + γmaxaQt(st+1, a’) – Qt(st, at)

Unfortunately, cutting off traces every time an exploratory action is taken loses much of the advantages of using eligibility traces. If exploratory actions are frequent, as they often are early in learning, then only rarely will backups of more than one or two steps be done, and learning may be little faster than one step Q learning.

**Q(λ) algorithm**

*initialize Q(s, a) arbitrarily and e(s, a) = 0 for all s, a*

*repeat for each episode*

*initialize s, a*

*repeat for each step of episode*

*take action a, observe r, s’*

*choose a’ from s’ using policy derived from Q (ε-greedy)*

*a\*🡨 argmaxb Q(s’, b)*

*δ🡨 r + γQ(s’, a\*) – Q(s, a)*

*e(s, a)🡨 e(s, a) + 1*

*for all s, a*

*Q(s, a) 🡨 Q(s, a) + αδe(s, a)*

*If a’ = a\* then e(s, a) 🡨 γλe(s, a)*

*else e(s, a) 🡨 0*

*s 🡨 s’; a🡨 a’*

*until s is terminal*

**1.** **Cooperative Q-learning: the knowledge sharing issue [1]**

**A** new cooperative learning method called weighted strategy sharing (WSS) with expertness measuring criteria was developed by Majid Nili Ahmadabadi, Masoud Asadpour and Eiji Nakano. Based on the amount of its teammate expertness, each agent assigns a weight to their knowledge and utilizes it accordingly in WSS method. Expertness criteria and WSS were tested on two simulated hunter–prey and object-pushing systems.

 **Algorithm 1 WSS algorithm for agent ai**

1. initialize
2. while not EndOfLearning do
3. begin
4. if InIndividualLearning Mode then
5. begin {Individual Learning}
6. xi := FindCurrentState()
7. ai := SelectAction()
8. DoAction(ai)
9. ri := GetReward()
10. yi := GoToNextState()
11. v(yi) := MaxbactionsQ(yi,b)
12. Qinew(xi,ai) := (1 - βi)QiOld(xi,ai)+ βi(ri +γiV(yi) )
13. ei := UpdateExpertness(ri)
14. end
15. else {Cooperative Learning}
16. begin
17. for j := 1 to n do
18. ej := GetExpertness(Aj)
19. Qinew := 0
20. for j := 1 to n do
21. begin
22. Wij := ComputerWeights(i,j,e1…...en)
23. Qjold := GetQ(Aj)
24. Qinew := Qinew  + Wij \* Qjold
25. end
26. end
27. end

**WSS METHOD**

It is assumed that members of a group of **n** homogeneous agents are learning in some environment in WSS method (Algorithm 1). Agents actions do not change the others learning environment (and do not produce the Hidden State Problem).

Two modes of learning are used i.e.: Individual Learning & Cooperative Learning Mode (lines 4 and 15 of Algorithm 1). All of the agents are in the Individual Learning Mode initially. Based on the one-step Q-learning, agent i executes ti learning trials. Each learning trial starts from a random state and ends when the agent reaches the goal. All agents stop the Individual Learning mode at the time when a specified number of individual trials are performed (which is called the cooperation time). Then agents switch to Cooperative Learning Mode.

Each learner assigns some weights to the other agents according to their expertness values **(Section 3.2)** in theCooperative Learning Mode**.** Then, it takes a weighted average of the others’ Q-tables and uses the resulted table as its new Q-table. (In Algorithm 1, multiplication (\*) and Summation (+) operators are specified according to the knowledge representation method.

**1. Individual learning based on Reinforcement Learning**

One-step Q-learning is used for the Individual Learning Mode. In the one-step Q-learning algorithm the external world is modeled as a Markov Decision Process with discrete time finite states. Next to each action, the agent receives a scalar ‘reward’ or ‘reinforcement’. The state-action value table, the Q-table, which estimates the long-term discounted reward for each state / action pair, determines the learned policy of the agent. Given the current state and the available actions ai, a Q-learning agent selects action ‘*a*’ with the probability .P / given by the Boltzmann distribution (line 7 of Algorithm 1):

P(ai|x) =

where t is the temperature parameter and adjusts the randomness of the decision. The agent executes the action (line 8), receives an immediate reward r (line 9), moves to the next state y (line 10) and updates Q as (line 12):

Qinew(xi,ai) := (1 - βi)QiOld(xi,ai)+ βi(ri +γiV(yi) )

where β is the learning rate, (γ< 0 γ < 1 ) is a discount parameter and V (x) is given by (line 11):

V(y)=maxbactionsQ(y,b)

Q is improved gradually and the agent learns when it searches the state space.

**2. Measuring the expertness values**

In human societies it is observed that, a learner evaluates the others’ knowledge with respect to their expertness. In other words, each learner tries to and the best evaluation method to and out how much the others’ knowledge is reliable. Different mechanisms for choosing the expert agents are used. In this WSS Method also weights of each agent’s knowledge is carefully calculated so that the group learning efficiency is maximized. The expertness measuring criterion significantly affects the learning efficiency.

***Expertness Measures****:* Expertise is defined as the “embodiment of knowledge and skills within individuals”. Expertise measure is performed at different levels. Expertness measures at the agent level (called as the *Q*-table) show the overall expertness level of an agent. The expertness measures at state level indicate how well the agent can find the optimal action in that state. As a result, proper expertness measures at the state level can be used to approximate agents’ area of expertise.

Two classes are used for expertness measures. In the first category, the computation of measure demands for extra information beyond the *Q*-table. This information needs to be gathered during the learning phase. Normal, absolute, positive, and negative expertness measures that belongs to this category. It is called as expertness measures based on learning history. In second category, expertness is measured merely based on the *Q*-values with no extra information. In the second case, there is no need for observations or *a priori* knowledge. Certainty and entropy, as expertness measures that belong to this category, are mentioned in this category. It is called as expertness measures based on Q-tables.

Six methods for measuring the agents’ expertness are used here. These methods are:

(i) Normal (Nrm): Sum of all rewards.

(ii) Absolute (Abs): Sum of absolute value of rewards.

(iii) Positive (P): Sum of the positive rewards.

(iv) Negative (N): Sum of absolute value of the negative rewards.

(v) Certainty (Cer): *P*robability of selecting action with the maximum *Q*-value.

(vi) Entropy (Ent): This term refers to the relative degree of randomness.

***1) Based on Learning History:***

***Norma (Nrm)****:* This criterion gives more credit to those who have more success and fewer failures. This is represented by an algebraic sum of the reinforcement signals

eiNrm =

where *ri*(*t*) is the amount of reinforcement signal that environment gives to agent *i* in step *t*.

***Absolute (Abs):***This measure considers both rewards and punishments as a sign of being experienced. It means that failures and successes, weighted by the value of reward and punishment signals, are both valuable for the agent. This is the sum of the absolute value of the reward signals.

eiAbs =

***Positive (P):***This measure disregards experiences that do not result in gaining rewards and considers only rewarding experiences. A sum of the positive reward signals is used for this formula

eiP = = 0 if 0

= otherwise

***Negative (N):***This formula looks at unsuccessful tries only and assigns a higher expertness value to those who experienced more failures. It means that the agent who has more punishments knows ways not leading to gaining rewards better and they should be considered as more expert, as the agent gets more punishments than rewards during learning. Negative is the sum of the absolute value of the negative reinforcement signals.

eiN = = 0 if > 0

= otherwise

***2) Expertness Measures Based on Q-Table:***

***Certainty (Cer):***This term is defined as the probability of selecting action with the maximum *Q*-value. In fact, certainty shows how much the agent is confident in the expected value of the selected action relative to the other actions. Therefore, provided that the action selection strategy is a Boltzmann one, the certainty is calculated as

(x)=

***Entropy (Ent):*** This term refers to the relative degree of randomness. The higher the entropy, the less is the difference in probability of selecting actions. Maximum entropy is attained when all actions are equally prone to be selected. A minimum entropy of zero is attained when one of the actions is a clear cut to be selected with probability one. Neumann–Shannon entropy formula is used to compute the entropy for state *x* as

eEnt(x) = -

where Pr(*a|x*) is the probability of selecting action *a* in state x. In spite of certainty that just considers the best action, entropy considers all the actions. To extend this measure to cases with more than one state, an average of the entropy on the states is computed.

**3. The weight-assigning mechanism**

Learner uses only the Q-tables of more expert agents to decrease the amount of communication required to exchange Q-tables. Therefore, partial weights of the less expert agents are assumed to be zero. Learner i assign the weight to the knowledge of agent j as:

1 - αi if i = j

Wij = if ej > ei

0 otherwise

where 0< α< 1 is the impressibility factor and shows how much agent i relies on the others knowledge. ei and ej are the expertness value of agents i and j respectively, and n is the total number of the agents.

A **pseudorandom number generator** (**PRNG**), also known as a **deterministic random bit generator** (**DRBG**) is an [algorithm](http://en.wikipedia.org/wiki/Algorithm) for generating a sequence of numbers whose properties approximate the properties of sequences of [random numbers](http://en.wikipedia.org/wiki/Random_number_generation). The PRNG-generated sequence is not truly [random](http://en.wikipedia.org/wiki/Random), because it is completely determined by a relatively small set of initial values, called the PRNG's [*seed*](http://en.wikipedia.org/wiki/Random_seed) (which may include truly random values). Good statistical properties are a central requirement for the output of a PRNG. In general, careful mathematical analysis is required to have any confidence that a PRNG generates numbers that are sufficiently close to random to suit the intended use.

**Periodicity**

A PRNG can be started from an arbitrary initial state using a [seed state](http://en.wikipedia.org/wiki/Random_seed). It will always produce the same sequence when initialized with that state. The *period* of a PRNG is defined thus: the maximum, over all starting states, of the length of the repetition-free prefix of the sequence. The period is bounded by the number of the states, usually measured in [bits](http://en.wikipedia.org/wiki/Bit). However, since the length of the period potentially doubles with each bit of "state" added, it is easy to build PRNGs with periods long enough for many practical applications.

If a PRNG's internal state contains *n* bits, its period can be no longer than 2*n* results, and may be much shorter. For some PRNGs, the period length can be calculated without walking through the whole period. [Linear Feedback Shift Registers (LFSRs)](http://en.wikipedia.org/wiki/Linear_feedback_shift_register) are usually chosen to have periods of exactly 2*n*−1. Although PRNGs will repeat their results after they reach the end of their period, a repeated result does not imply that the end of the period has been reached, since its internal state may be larger than its output; this is particularly obvious with PRNGs with a 1-bit output.

**Mathematical definition**

Given

* P- a probability distribution on ( ) (where is the standard [Borel field](http://en.wikipedia.org/wiki/Borel_field) on the real line)
* - a non-empty collection of Borel sets .

If is not specified, it may be depending on context.

* A  - a non-empty set (not necessarily a Borel set). Often A is a set between P’s [support](http://en.wikipedia.org/wiki/Support_%28mathematics%29) and its [interior](http://en.wikipedia.org/wiki/Interior_%28topology%29), for instance, if P is the uniform distribution on the interval (0,1], A might be (0,1]. If A is not specified, it is assumed to be some set contained in the support of P and containing its interior, depending on context.

we call a function : (where = {1,2,3….} is the set of positive integers) a **pseudo-random number generator for P given taking values in A** iff

* ()  **A**
* E < ε < ε

#s denotes the number of elements in the finite set S.)

It can be shown that if F is a pseudo-random number generator for the uniform distribution on (0,1)and if F is the [CDF](http://en.wikipedia.org/wiki/Cumulative_distribution_function) of some given probability distribution P, then F\* f is a pseudo-random number generator for P, where F\*:(0,1)🡪 is the percentile of P, i.e. F\*(x):=inf{t : xF(t)}. Intuitively, an arbitrary distribution can be simulated from a simulation of the standard uniform distribution.

**Generalization and Function Approximation**

We stored Q(s,a) values in a lookup table and the Q learning and Sarsa algorithms are called tabular algorithms. There are number of problems with this approach: 1) when the number of states and the number of actions is large, the size of the table may become quite large; 2) states and actions may be continuous, and to use a table, they should be discredited which may cause error and 3) when the search space is large, too many episodes may be needed to fill in all the entries of the table with acceptable accuracy.

The only way to learn anything at all on these tasks is to generalize from previously experienced states to ones that have never been seen. The kind of generalization required is called as function approximation because it takes examples from a value function and attempts to generalize from them to construct an approximation of the entire function. Function approximation is an instance of supervised learning.

Instead of storing the Q values as they are, we can consider this a regression problem. This is a supervised learning problem where we define a regressor (s,a | θ) taking s and a as inputs and parameterized by a vector parameters θ to learn Q values. A

**A liner, gradient-descent version of Watkins’s Q(λ) with binary features, ε greedy policy and accumulating traces.**

*Initialize θ arbitrarily and e = 0*

*Repeat for each episode*

*S🡨 initial state*

*For all a є A(s)*

*Fa 🡨 set of features present in s,a*

*Qa 🡨 ∑iєFaθ(i)*

*Repeat for each step of episode*

*With probability 1-ε;*

*a🡨 argmaxa Qa*

*e🡨γλe*

*else*

*a🡨 a random action є A(s)*

*e🡨 0*

*For all i є Fa: e(i) 🡨 e(i) +1*

*Take action a, observe reward r and next state s’*

*δ🡨 r - Qa*

*For all a є A(s’)*

*Fa🡨 set of features present in s’, a*

*Qa🡨 ∑iєFaθ(i)*

*a'🡨 argmaxa Qa*

*δ🡨 δ + γQa*

*θ🡨 θ + αδe*

*until s’ is terminal*

**8. Cooperative Reinforcement Learning Algorithm to Distributed Power System based on Multi-Agent[27]** La-mei GAO, Jun ZENG, Jie WU

With the development of renewable energy technology, the distributed wind-PV power system has a wider application. This paper proposes a distributed wind-PV power system based on Multi-Agent, whose main character is energy management, and describes the multiagent cooperative reinforcement learning process using the joint action learning pattern as the cooperative strategy. The experiment of a distributed wind-PV power system shows the efficiency.

Q learning is one of the main algorithms of reinforcement learning, and it is a form of model-free reinforcement learning. Q function is defined as the strengthened cumulative discount reward which is obtained through executing action at the state s, and after this executing the best action sequence. The object of Q learning is to look for a strategy which can maximize the reward in the future. The optimal Q value can be expressed as , defined as the reward summation that obtained through implementing correlative actions The distributed wind-PV power system consists of wind turbine, solar cell, and storage battery. Because of its small-scaled system and its disperse space, it is difficult to use concentrative providing energy. This paper takes every power subsystem for an intelligent Agent. Each subsystem consists of perception module, communication module, learning module, knowledge base, decision making module, executing module as in figure 1.

In a multi-Agent system, the environment is dynamically changing, and other Agents’ behaviors are unknown, so it is almost impossible to build a complete priori model. And many field knowledge is gradually obtained through interacting between Agent and other Agents. Multi-Agent cooperative reinforcement learning means that many Agents reciprocally communicate and cooperate in the process to pursuing a common object. Because the Agents change their own states and environment after obtaining information, every Agent gets the influence from other Agents’ knowledge, beliefs, intentions and so on during the learning process.

The distributed wind-PV power system is such a multiagent system that is in the dynamically changing environment. In order to overcome its disadvantages, for example, without complete priori model and knowledge, and single agent’s uncompleted learning, this paper proposes a **Joint Action Learning (JAL)** model. In this model, the current action that one Agent is executing is the optimal response to one of other Agents’ congregations of actions. Because this paper is discussing a distributed Multi-Agent system, each Agent in this system is indistinctive. Here, the JAL is a learning manner which is based on the forecast that each Agent toward other Agents’ actions. According to the system structure proposed before, the learning module is shown in Figure 2. The cooperative reinforcement learning algorithm this paper proposed is described as follows:

This system consists of six wind turbines and four photovoltaic cells (PV), with a total capacity of 70KW. The quaternion array in this paper is defined as

where Wspeed for wind speed, Wdir for wind direction, Isun for sunlight, Lneed for load requirement, Sequip (including four states, that is hot-standby, cold-standby, downtime, and network) for current state of wind turbine or PV. This paper only considers the wind turbines and PV at the hot standby state, so each Agent’s action set is A= {a1,a2},

This paper takes one decision making process for a learning process. Each decision making may be initiated by the user Agent or other Agent, so the learning process we discussed here is a decision making process initiated by different Agent asynchronously. Here the Q value will take no account of the impact of the future value. So, the discount factor γ=0, the reward is R decided by three factors together, that are: whether balance between supply and requirement (R1), power quality (R2), as well as the electrical price (R3).

The research of Multi-Agent system’s cooperative mechanism usually emphasizes the single learning of Agents and takes no account of other Agents’ actions, so that the MAS lacks the cooperative mechanism. This paper proposes a Multiagent cooperative reinforcement learning algorithm—the joint action reinforcement learning algorithm. In the algorithm, each Agent forecasts its own action strategy through observing the historical actions of other cooperative ones and makes the corresponding decision making to achieve the optimal joint action strategy. This paper carries out analyze and research with this algorithm to the distributed wind-PV power system, and shows the feasibility of the algorithm.

**5. Improvement with Joint Rewards on Multi-agent Cooperative Reinforcement Learning [33]** Pan Ying & Li Dehua

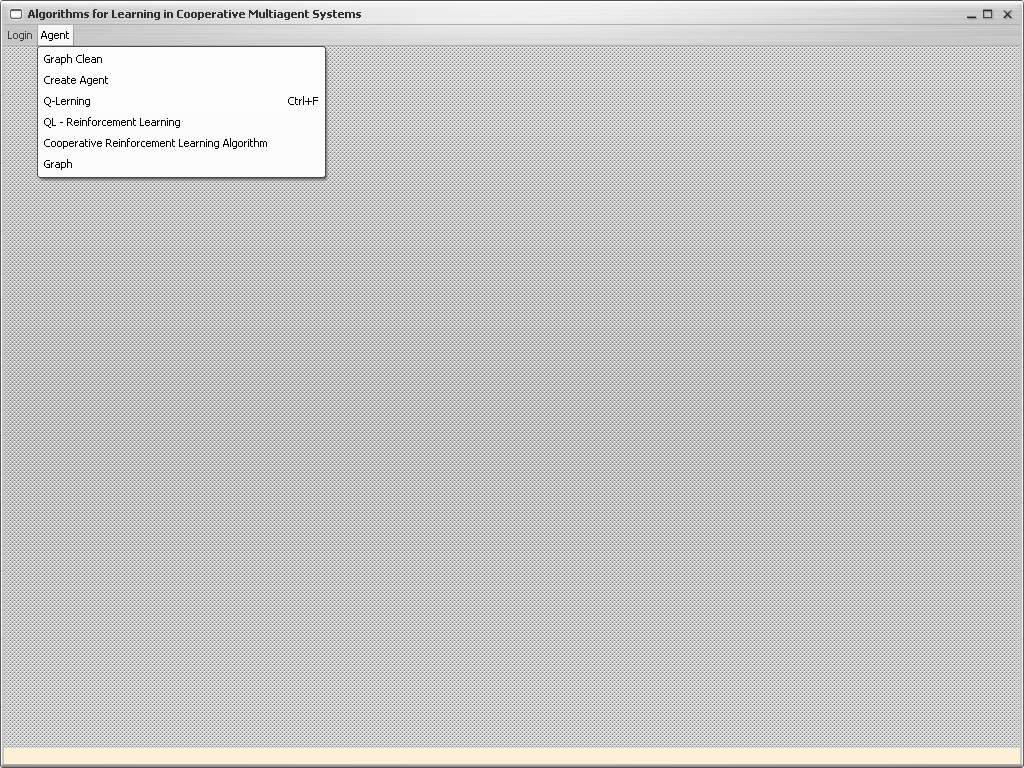
Cooperation among agents is important for multiagent system. In this paper, an improved cooperative reinforcement learning algorithm is proposed, which based on joint rewards to insure agents to learn cooperative behavior. Furthermore, a symmetry idea is included in the algorithm to reduce the states size of reinforcement learning. The experiment results show the efficiency and well convergence of the algorithm. In recent years, more and more interest has been attracted into multi-agent environment. In a multi-agent environment, it is not sufficient for each agent to act selfishly in order to arrive at a globally optimal strategy, and often, it is impossible to

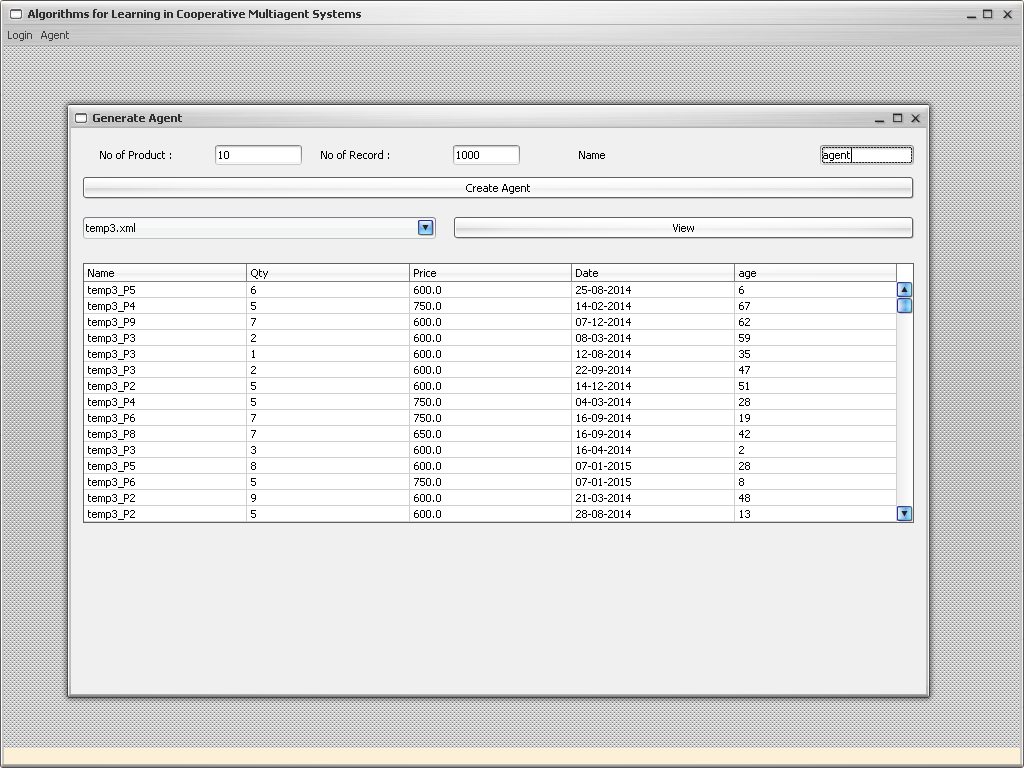
accomplish with only one agent in order to achieve a task. So cooperation among agents should be realized. Reinforcement learning, first proposed by Littman in 1994, is a learning method to get optimal policy through trialand-error and interaction with dynamic environment. It has properties of self-adaptive and online learning, and has been widely applied to the study on the cooperative behavior in multi-agent systems, such as pursuit problem, soccer, the prisoners’ dilemma game, and coordination games.

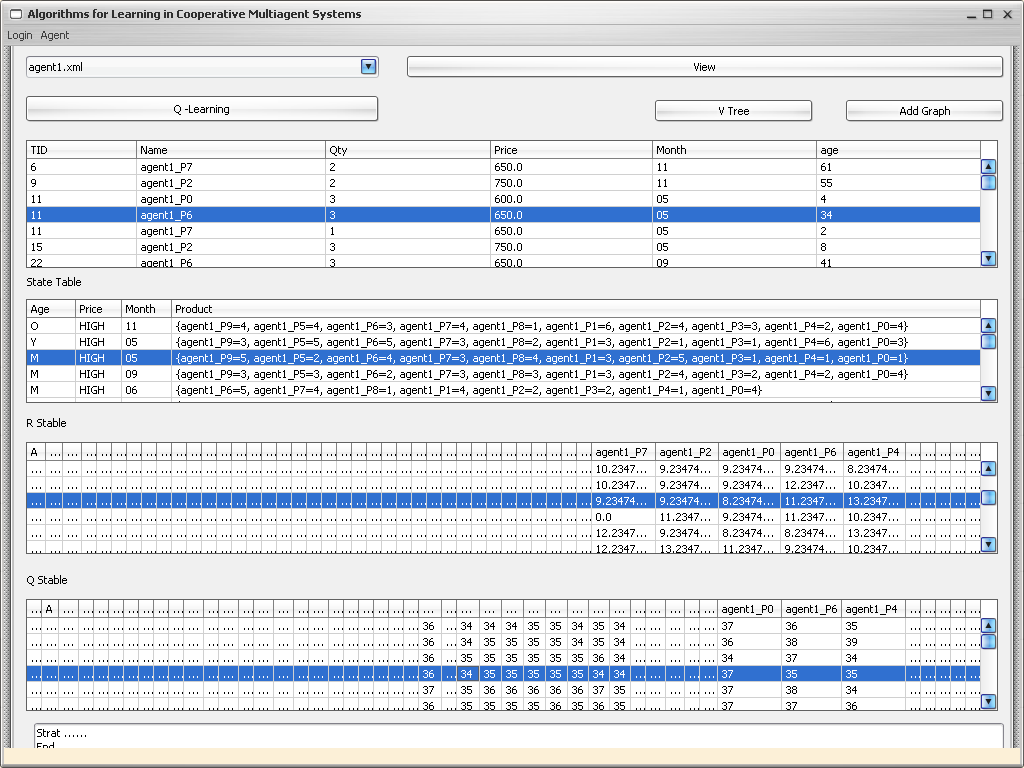
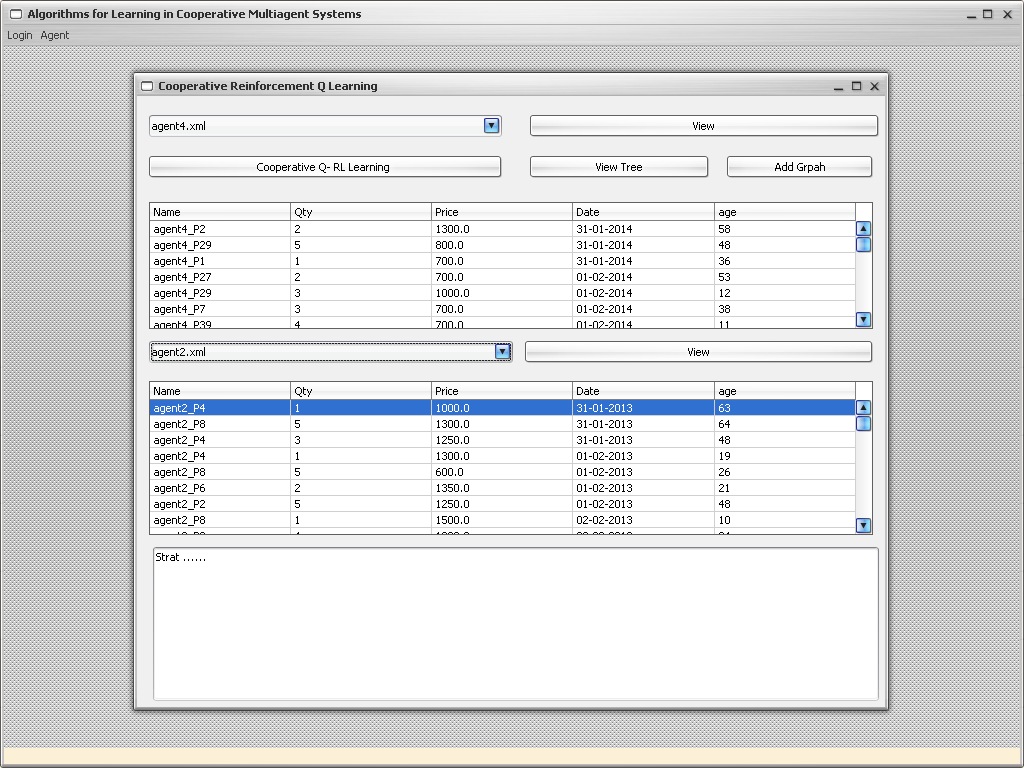
However, Reinforcement learning method cannot scale well if a problem has a large state space. Symmetry is utilized for state abstraction. Combining the vicarious reinforcement with the idea of symmetry, an improved learning algorithm with joint rewards on multi-agent cooperative reinforcement learning is proposed in this paper. The improved learning algorithm reduces the representation size of the state space with symmetrical information firstly, and realizes the cooperative behavior among agents through a joint rewards function. The improved learning algorithm is tested on the pursuit problem. The simulation results show the efficiency of the learning. In a multi-agent system, every agent needs to maintain a Q-table which contains the information about its and others agent’ states and actions, i.e. the situation of whole environment (assumed agent’s observation distance is infinite). However, it is found that in most cases parts of states of different actions are symmetrical, and thus the action value function can be shared which further reduces the reinforcement learning problem size. Kamal proposed a symmetrical restricted states reinforcement learning method that given the symmetrical restricted states is p(*s)* , then Q-learning algorithm can be rewrote as

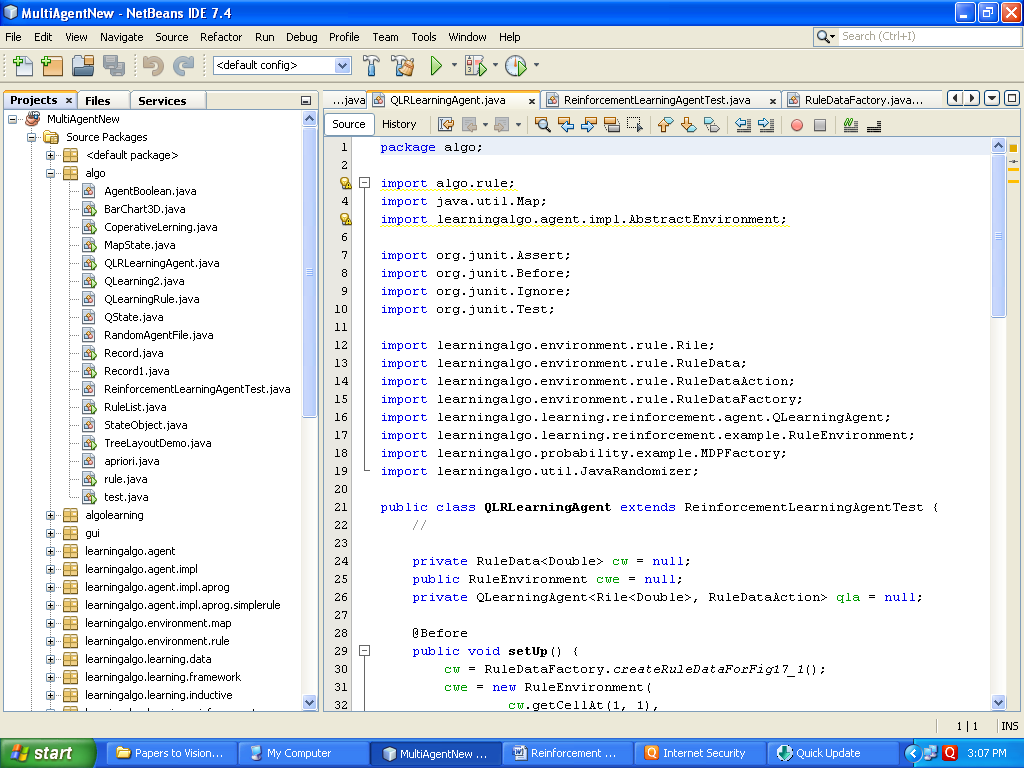
**Improved Multi-Agent Cooperative Reinforcement Learning Algorithm**











**Future work:**

Implementation of following algorithms

1. Q learning with function approximation
2. Two level Q learning etc.

**Publications:**

* D.A. Vidhate & Dr. Parag Kulkarni “Design of Multiagent System Architecture based on Association Mining for Cooperative Reinforcement Learning”, **Elsevier** **publication** by 2nd International Conference on Recent Trends in Engineering Sciences' on **15-16 Mar 2014.**
* D.A. Vidhate & Dr. Parag Kulkarni “New Technique using Multilevel Relationship Algorithm to improve Association Rule Mining : A step towards Cooperative Learning”, **Elsevier** **publication** by 3rd International Conference on Recent Trends in Engineering & Technology (ICRTET'2014) on **28-30 Mar 2014.**
* D.A. Vidhate & Dr. Parag Kulkarni “To improve Association Rule Mining using New Technique: Multilevel Relationship Algorithm towards Cooperative Learning”, **IEEE conference publication** by International Conference on Circuits, Systems, Communication and Information Technology Applications (CSCITA – 2014) on 04**-05 April 2014.**

