Fall 12-2014

Reinforcement Learning of Distributed Surveillance Plans

Madhavi Chittireddy
University of Southern Mississippi

The University of Southern Mississippi

REINFORCEMENT LEARNING OF DISTRIBUTED SURVEILLANCE PLANS

by

Madhavi Chittireddy

A Thesis
Submitted to the Graduate School
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Master of Science

Approved:

Dr. Bikramjit Banerjee
Committee Chair

Dr. Beddhu Murali

Dr. Dia Ali

Dr. Karen Coats
Dean of the Graduate School

December 2014

ABSTRACT

REINFORCEMENT LEARNING OF

DISTRIBUTED SURVEILLANCE PLANS

by Madhavi Chittireddy

December 2014

This thesis describes the design and implementation of a Reinforcement Learning algorithm on a camera surveillance model which is used to know the stackelberg strategies of attacker and defender. This reinforcement learning algorithm is compared with the uniform policy and hill climbing algorithms by executing them on a common set of different data files, generated programmatically with various combinations of problem size, location, and orientation transitions as well as rewards of attacker and defender. The comparison includes the time taken to obtain better stackelberg policy and the resulted final pay-off of the defender. This thesis shows that the reinforcement learning algorithm developed in Java performs better than the uniform policy and proves to be chosen for large problem size as it produces acceptable results in less time when compared to that of the hill climbing algorithm.

ACKNOWLEDGMENTS

I would like to thank my thesis director, Dr. Bikramjit Banerjee, and my other committee members, Dr. Beddhu Murali and Dr. Dia Ali, for their advice and support throughout the duration of this thesis. I would especially like to thank Dr. Bikramjit Banerjee for his enormous patience and the time he spent in putting this work together.

I would also like to thank Dr. George Glover, Adjunct Instructor/Academic System Administrator at The University of Southern Mississippi (USM), for his guidance in dealing with Albacore cluster. Appreciation must also be expressed to Landon Kraemer, PhD student in Computer Science at The University of Southern Mississippi (USM), for his help with AMPL modeling and hill climbing algorithm.

This work was supported by the U.S. Army under grant #W911NF-11-1-0124, but reflects the opinions solely of this author.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF ILLUSTRATIONS

Figure

LIST OF ABBREVIATIONS

AMPL	A Mathematical Programming Language
LA	Learning Agent
RL	Reinforcement Learning

CHAPTER I

INTRODUCTION

Background

Stackelberg games have been used in several deployed applications of game theory to make recommendations for allocating limited resources for protecting critical infrastructure. A Stackelberg security game models an interaction between an attacker and a defender [5]. The defender first commits to a security policy (which may be randomized), and the attacker is able to use surveillance to learn about the defender's policy before launching an attack. A solution to the game yields an optimal randomized strategy for the defender, based on the assumption that the attacker will observe this strategy and respond optimally. Software decision aids based on Stackelberg games have been implemented in several real-world domains, including LAX (Los Angeles International Airport) [6], FAMS (United States Federal Air Marshals Service) [7], TSA (United States Transportation Security Agency) [8], and the United States Coast Guard [9].

Stackelberg Games

In some multiagent settings, one agent must commit to a strategy before the other agents choose their own strategies. These scenarios are known as Stackelberg games [1, 2]. In a Stackelberg game, a leader commits to a strategy first, and then a follower selfishly optimizes its own reward, considering the action chosen by the leader. Stackelberg games are commonly used to model attacker-defender scenarios in security domains [3] as well as in patrolling and could potentially be used in many other situations such as network routing, pricing in transportation systems, setting up security

checkpoints, and other adversarial domains. For example, consider a domain where a single security agent is responsible for patrolling a region, let us suppose he is searching for a robber in a particular region. Since the security agent (the leader) cannot be in all areas of the region at once, it must instead choose some strategy of patrolling various areas within the region one at a time. This strategy could be a mixed strategy in order to be unpredictable to the robber (follower). The robber, after observing the pattern of patrols over time, can then choose its own strategy for selecting a location to rob. Although the follower in a Stackelberg game is allowed to observe the leader's strategy before picking its own strategy, there is often an advantage for the leader in case of simultaneous games where both players must choose their moves at the same time. To see the advantage of being the leader in a Stackelberg game, consider a simple game with the payoff table (see Table 1), adapted from [4]. The leader is the row player, and the follower is the column player.

Table 1

Payoff table for example Normal Form game

	C	**D**
A	2, 1	4, 0
B	1, 0	3, 2

The only pure-strategy Nash equilibrium relevant in the simultaneous move game where the leader and follower select moves simultaneously is when the leader plays a, and the follower plays c which gives the leader a payoff of 2; in fact, for the leader,

playing b is strictly dominated. However, in the leader-follower version of this game, if the leader can commit to playing b before the follower chooses its strategy, then the leader will obtain a payoff of 3, since the follower would then play d to ensure a higher payoff for itself. If the leader commits to a uniform mixed strategy of playing a and b with equal (0.5) probability, then the follower will play d, giving the leader an expected payoff of 3.5. Therefore, the leader can extract a better expected payoff in the leader-follower Stackelberg version of a game, and can convey an advantage to it in real world situations where it can indeed move first.

CHAPTER II

DESIGN

Surveillance Domain Description

The author's scenario consists of a camera acting as defender, and a person acting as attacker planning to navigate through the view-field of the camera unnoticed. The camera's viewfield is divided into several orientation segments, only one of which it can monitor at any time. The attacker can see the camera's current orientation, but the camera cannot see the movements of the attacker. The attacker can also observe the temporal pattern of orientations of the camera and determine a movement plan that is most likely to pass unnoticed.

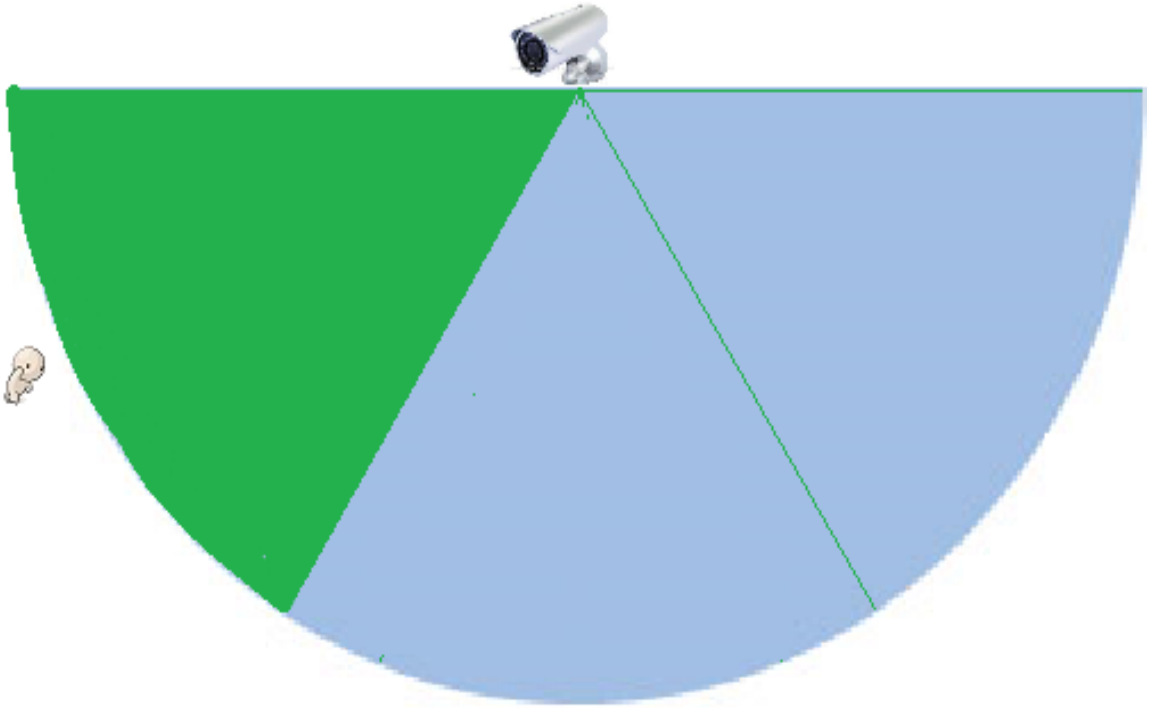

Figure 1. Domain Scenario. In this figure, the image animated as a small person is the attacker, and the blue colored inverted semicircle indicates the coverage area of the camera with green colored cone showing the current direction of the camera and the area that the camera can cover at a particular direction.

However, as discussed before, the camera has an advantage in that it is the first mover; it can select an orientation plan in such a way that even when the attacker maximally evades it, it still gets the highest expected payoff possible. The camera can receive a reward if its current orientation succeeds in capturing the attacker, otherwise the attacker receives a reward. This can also be applied to a real world scenario (see Figure 1). The goal of the attacker is to move through the view-field of the camera trying his best not to get caught by it (if possible) or to minimize the number of times he will get caught by the camera. The goal of the camera (defender) is to track the attacker as many times as possible.

Orientation and Location Transitions

Orientation corresponds to the direction of the camera, and Location refers to the position of the attacker. The camera has an orientation size (OSize) of at least 2 and at most 3 orientations, and the attacker has a location size (LSize) of 2 to 10 locations. Locations of the attacker are within the orientations of the camera. Each location is definitely covered by the camera in one or the other orientation; also the same location can be covered in 1 or more orientations. An Orientation Map (OMap) gives the locations that are covered in a particular orientation. The orientation transitions (OT) are the action set that can be performed by the camera corresponding to the orientations where the camera can move in the next step from the current orientation. Similarly, location transitions (LT) are the action set of the attacker corresponding to the locations where the attacker can move in the next step from the current location. These orientation and location transitions are generated at random for each experiment file.

Actions

The author have assumed that all the players have a finite set of actions which they can perform depending on the domain scenario. At any point a player can perform only one action from the given set of actions. There are different sets of these actions for the attacker and the defender, generated randomly for each experiment. Mostly, the camera's action set consists of three actions namely 'turn left' (from the current orientation), 'no turn' (stay at the same orientation without turning to left/right) and 'turn right' (from the current orientation). These action sets at a particular orientation and location can be known from OT and LT of the camera and the attacker, respectively.

Reward Structure

In general, the reward structure can be modeled in different ways. The author opted a reward structure where we have modeled the game in such a way that the attacker gets either a positive or a negative reward where as the defender gets a reward of 0 (zero) or a positive reward which is explained in following cases:

case-i: if the attacker is caught by the camera (defender) then,

Camera gets a positive reward and

Attacker gets negative

case-ii: if attacker escapes from being caught by the camera then,

Attacker gets a positive reward and

Camera gets a reward of 0 (zero)

CHAPTER III

CONCEPTS/ALGORITHMS USED

Linear Programming

Linear programming (LP) is a method used to get the best outcome in a mathematical model whose requirements are represented by linear relationships. The author has used this concept to formulate a Linear program (LP) in such a way that it computes the maximum payoff the defender can get for any given policy against the best response of the attacker. See the below canonical form of LP, where σ is a binary variable, x is a parameter, p(l,o) is a variable that keeps the track of the frequency of l and o (joint state occurrence), and β is a variable that is computed as x * σ * p(l, o).

#Linear Program to compute defender's maximum payoff against attacker's best response

Maximize: $\sum_{o,la} \mathrm{p(la,o)} * \mathrm{R^{attacker}}$

Subject to

$$defender = \left(\sum_{o,la} \mathrm{p(la,o)} * \mathrm{R^{defender}}\right)$$

$$\forall l' \in \mathrm{L}, \forall \mathrm{o}' \in \mathrm{O},\ p(l', o') = \sum_{l,o,l\prime,o\prime} \beta_{l\prime,\mathrm{o}\prime}^{l,\mathrm{o}}$$

$$\forall l \in \mathrm{L}, \forall \mathrm{o} \in \mathrm{O},\quad p(l, o) = \sum_{l\prime,o\prime} \beta_{l\prime,\mathrm{o}\prime}^{l,\mathrm{o}}$$

$$\sum_{l,o,l\prime,o\prime} \beta_{l\prime,\mathrm{o}\prime}^{l,\mathrm{o}} = 1$$

$$\forall l \in \mathrm{L}, \forall \mathrm{o} \in \mathrm{O}, \forall l' \in \mathrm{LT}[l], \forall \mathrm{o}' \in \mathrm{OT}[\mathrm{o}],\quad \beta_{l\prime,\mathrm{o}\prime}^{l,\mathrm{o}} \leq \sigma_{l\prime}^{l,\mathrm{o}}$$

$$\forall l \in \mathrm{L}, \forall \mathrm{o} \in \mathrm{O}, \forall l' \in \mathrm{LT}[l], \forall \mathrm{o}' \in \mathrm{OT}[\mathrm{o}],\quad \beta_{l\prime,\mathrm{o}\prime}^{l,\mathrm{o}} \leq \mathrm{x(o,o')} * \mathrm{p}(l,\mathrm{o})$$

$$\forall l \in \mathrm{L}, \forall \mathrm{o} \in \mathrm{O}, \forall l' \in \mathrm{LT}[l], \forall \mathrm{o}' \in \mathrm{OT}[\mathrm{o}],\quad \beta_{l\prime,\mathrm{o}\prime}^{l,\mathrm{o}} + \left(1 - \sigma_{l\prime}^{l,\mathrm{o}}\right) * 100000 \geq \mathrm{x(o,o')} * \mathrm{p}(l,\mathrm{o})$$

$$\sum_{l,o} \mathrm{p}(l,\mathrm{o}) = 1$$

$$\forall l \in \mathrm{L}, \forall \mathrm{o} \in \mathrm{O}, \quad \sum_{l'} \sigma_{l'}^{l,o} = 1$$

This LP is provided as a model file to run on AMPL shell with a CPLEX optimizer for various data files on USM's Albacore cluster to get the required payoff values for the defender.

Uniform Policy

Uniform Policy is a simple algorithm that gives the payoff value for the defender playing a fixed uniform policy where the defender probabilities are distributed uniformly over every possible action of the defender at a given defender state against the attacker's best response. This algorithm calls AMPL using runAMPL() function with the above described LP as model file and data file having uniform defender policies. The Pseudo code of the algorithm is shown below.

```
//Uniform Policy Algorithm
        for i = 2..10
            for j = 1..20
                runAMPL(LP.mod, security_i_j.dat);
```

The Uniform Policy algorithm is run on 180 (9*20) different data files which contain uniform policies for defender corresponding to the OT of each data file. The average of every 20 files with the same Node Size (OSize and LSize) is considered to generate the plot (see Uniform Policy bars in Figure 2).

Hill Climbing Algorithm

Hill Climbing algorithm can be described as a concept where we have hills with different heights, and our goal is to find the highest hill and to reach the peak of that highest hill. In the hill plots, the abscissa represent the defender's policies, and the

ordinate represent the defender's values of those policies. This algorithm has few random starts indicating random defender policies, where at every start the defender payoff against the attacker's best response is calculated by considering all of its neighbors. Among them, the neighbor or the point that gives the highest defender payoff is chosen, indicating that the hill with maximum height is found and that the peak of that hill is reached.

This algorithm also calls AMPL with the above described LP model file and data file having random starts as defender policies, exploring all its neighbors to reach the highest peak. These random starts also include a start with uniform defender policies. The Pseudo code of the algorithm is shown below.

```
//Hill Climbing Algorithm
                for i = 2..10
                 for j = 1..20
                  loop  random starts(security_i_j.dat)
                        // with random defender policies including uniform policy
                        //each start representing the starting point in exploring  a hill
                        runAMPL(LP.mod, security_i_j.dat);
                        exploreNeighbors(security_i_j.dat);

        function exploreNeighbors(security_x_y.dat)
                loop //to reach the peak of the hill
                        runAMPL(LP.mod, security_x_y.dat);
```

This algorithm is run on the same 180 different data file as that of the Uniform Policy Algorithm, where the data files have random defender policies for each start, and the policies are being modified while all the neighbors of every start are been explored. The average of every 20 files with the same Node Size (OSize and LSize) is considered to generate the plot (see hill climbing algorithm bars in Figure 2).

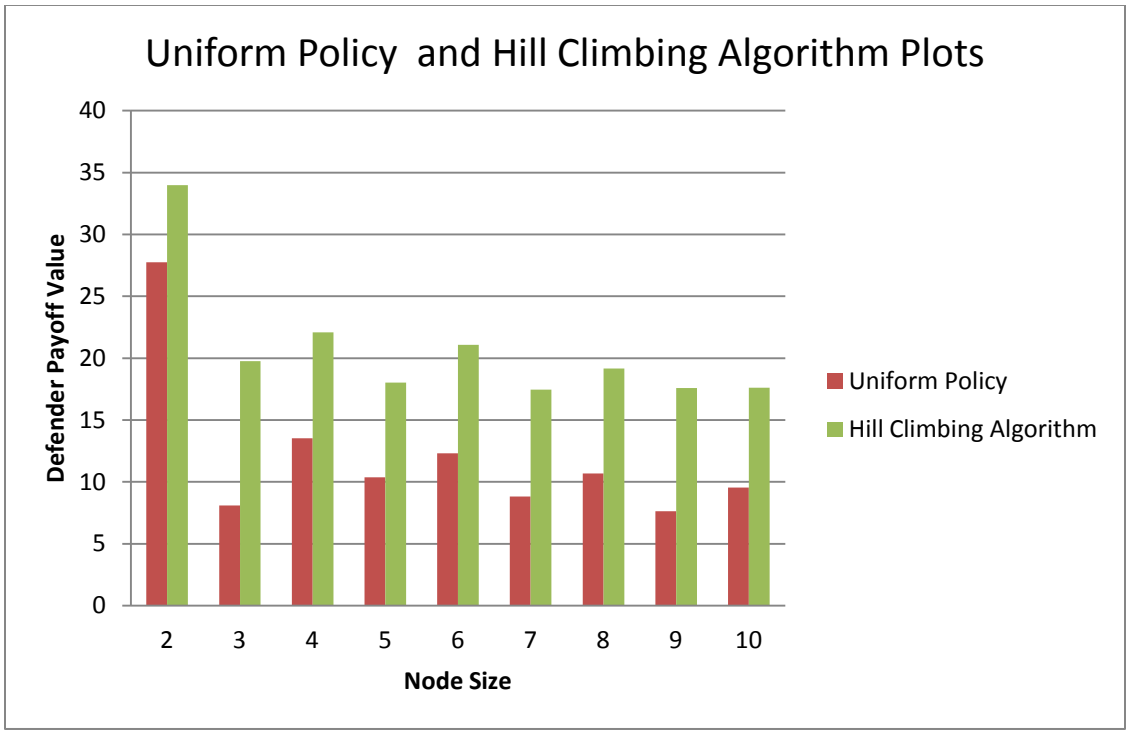

Figure 2. Uniform Policy and Hill Climbing Algorithm Plots. This shows the average values of the defender payoff resulted from the uniform policy and the hill climbing algorithms, which are run independently over the same set of different data files with node size varying from 2 to 10.

Learning Agent Concept

"Learning Agent" is an algorithm we used to make the agents, either attacker or defender or both, to learn a policy. A general Learning Agent Model is shown in Figure 3.

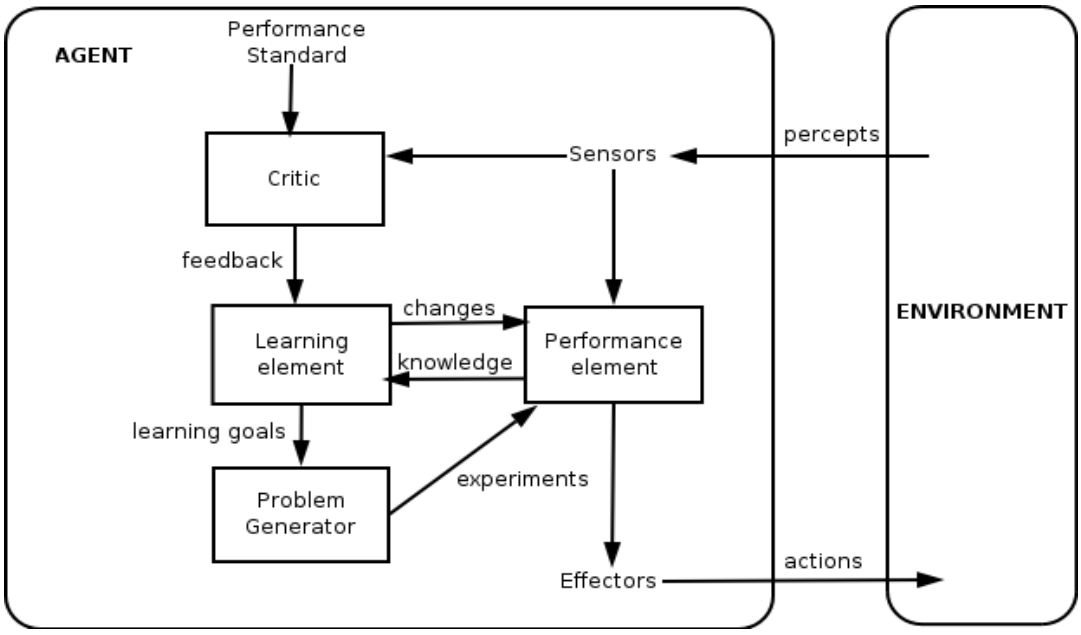

Figure 3. A General Learning Agent Model, describing the elements in a learning agent.

Learning allows the agents to initially operate in unknown environments and to become more competent than its initial knowledge alone might allow. The most important distinction is between the "learning element," which is responsible for making improvements, and the "performance element," which is responsible for selecting external actions. The learning element uses feedback from the "critic" on how the agent is doing and determines how the performance element should be modified to do better in the future. The performance element is considered to be the entire agent; it takes in percepts and decides on actions. The last component of the learning agent is the "problem generator." It is responsible for suggesting actions that will lead to new and informative experiences.

The case of the author's Learning Agent algorithm, exploration using 'epsilon-greedy selection' serves as the "problem generator." The Learning Agent algorithm comprises of the following important methods namely:

startEpisode() - called only once for an agent during the start of each episode with initial start-state as input, which returns an action by calling selectAction().

step() - follows startEpisode() and is called more than once per an episode during which the agent learns a policy, where Q-Update is done which is used to maintain and update learning policy, and the next action is generated using selectAction().

endEpisode() - called only once indicating end of an episode.

selectAction() - performs Epsilon - greedy selection with a exploring rate (ε) of 0.01 and calls argmaxQ()

argmaxQ() - used to select the best action with the best learnt Q-value

Here, the rewards of the attacker and the defender, which they get for moving from one state to another, serves as the feedback from the "critic." The pseudo code of Learning Agent algorithm is shown below:

```
//Learning Agent Algorithm
function startEpisode(state) returns an action
        //takes initial-start state and returns an action
        //resulting from selectAction function.
        action = selectAction(state)
        return action
```

```
function step(reward, state) returns an action
        //takes reward of previous action with its resulting state as input
        //and returns an action.
        δ = reward - Q[lastState][lastAction]
        action = selectAction(state)
        if agent is learning Q-Update of previous state and  action is done
                Q[lastState][lastAction] + = α * δ
        return action


function endEpisode(reward)
        //takes reward of previous action
        if agent is learning
                //then Q-Update of previous state and action is done
                δ = reward - Q[lastState][lastAction]
                Q[lastState][lastAction] + = α * δ


function selectAction(state) returns an action
        //Epsilon-greedy
        if random < explorationRate than
                action = generate random action
         else
                action = argmaxQ(state)
        return action
```

```
function argmaxQ(state) returns an action
    loop over number of actions
        select the best action for given state with highest Q value
    return action.
```

CHAPTER IV

REINFORCEMENT LEARNING ALGORITHM

Initial RL Algorithm

Description

Initially, the author has developed an RL algorithm that makes a function call to the previous learning agent with little addition of code and concepts.

Q-Update is done as:

$$\mathrm{Q(s,a)} \leftarrow \mathrm{Q(s,a)} + (\alpha * \delta)$$

and Pi-Update where the value of π, which is used to maintain the probability of occurrence of a particular state and action, is updated as:

$$\pi(\mathrm{s,a}) \leftarrow \pi(s,a) + \left[\beta * \pi(s,a) * \left(Q(s,a) - \sum_{b} \pi(s,b)Q(s,b)\right)\right]$$

and then the π-values are normalized.

In addition to argmaxQ(), as previously described in learning agent algorithm, a piSample() method is included, where argmaxQ is used to select the best action for the attacker based on Q-values, and piSample is used to select an action for the defender based on the sampling of π-values.

Initial-RL algorithm calls the learning agent with the above described enhancements for both the attacker and the defender. The functions startEpisode(), step(), and endEpisode() are called with number of episodes up to 100,000 and step-size up to 1000. This is done twice, once for training and a second time for testing where during testing the agent-learn flag is set to false so that the agent will not learn but only executes the previously learned policy. Initial start states are generated randomly each time when

the startEpisode() is called. In this thesis, the attacker and the defender learn alternatively by initializing the learning rate of the defender to 0.0001 and the attacker as 100 times faster than the defender, alternating the learning rates after every 10,000 episodes. During the testing phase, the rewards of the attacker and the defender are individually calculated for each episode as:

$$\gamma\_Reward += reward * \gamma^{stepNum}$$

then the average of γ_Rewards over total number of episodes is calculated and is resulted as the final reward.

//Pseudo-code of Initial-RL algorithm

readDatFile(datFileName)

//to read required data like OSize, LSize, LT, OT, OMap, RA and RD

//from given data file

//Learning agents for attacker and defender

attackerAgent = initializeLA( )

defenderAgent = initializeLA( )

Q_{at} , $Q_{def} \leftarrow 0$

π_{at} , π_{def} ← Uniform Policies

//Training - to train the attacker and defender learning agents

callLA( )

//Testing - to test the learned policies of attacker and defender learning agents

callLA( )

defFinalReward = defTotalGammaReward/100,000

```
function callLA( )
//makes frequent calls to Learning Agent
loop up to 100,000 // Episode count
        defState = random(OSize)
        attState = random(LSize)
        //start episodes
        attAction = attackerAgent.startEpisode(jointState)
        defAction = defenderAgent.startEpisode(defState)
        getRewards( ) //for corresponding state-actions
        updateStates( ) //to get resulting state for the action performed
        //step
        loop up to 1000 // Step size
                attAction = attackerAgent.step(jointState, attReward)
                defAction = defenderAgent.step(defState, defReward)
                getRewards( ) //for corresponding state-actions
                updateStates( ) //to get resulting state for the action performed
                if testing //not for training
                        calculateGammaRewards( ) //as described above
        //end of inner loop - step size
        //end episodes
        attackerAgent.endEpisode (attReward)
        defenderAgent. endEpisode (defReward)
//end of outer loop - episode count
```

Output Analysis

The above described initial-RL algorithm was run on 180 different data files with different combinations of OSize, LSize, OT, LT, OMAP as well as different attacker and defender rewards. The average of resulted defender payoff of every 20 files, having same OSize and LSize, are plotted and compared with the results of hill climbing and uniform policy algorithms (see Figure 4).

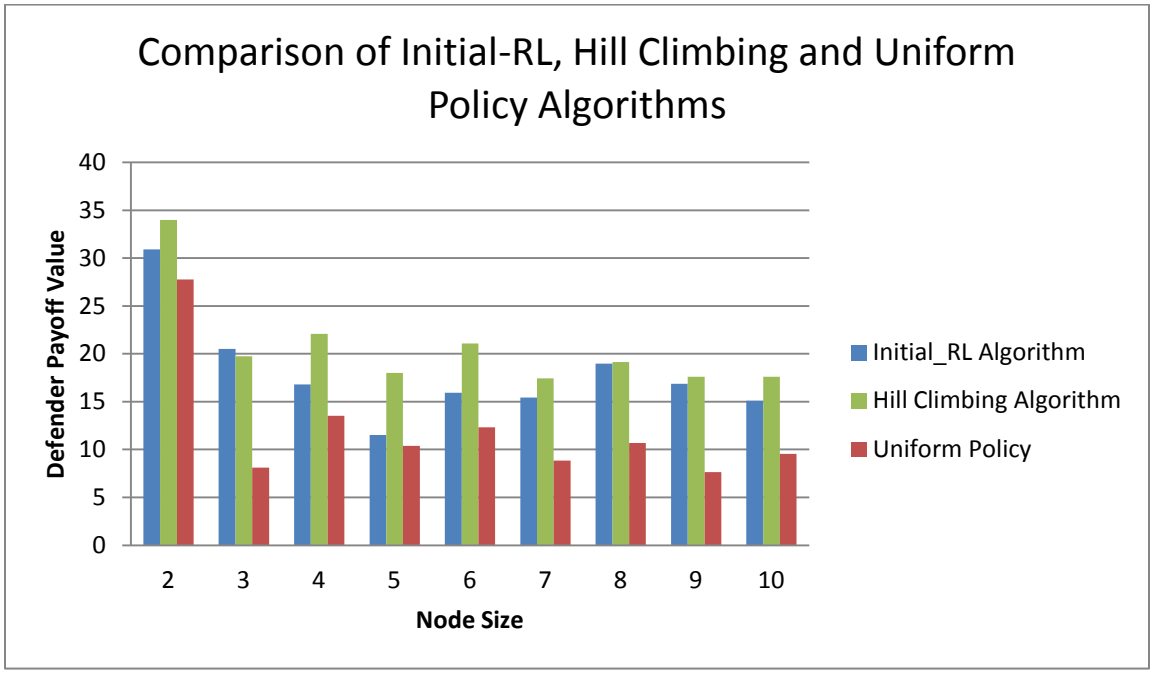

Figure 4. Comparison of Initial-RL, Hill Climbing, and Uniform Policy Algorithms. This shows that initial-RL algorithm close results to that of hill climbing algorithm, saying that defender gets good payoffs, in some cases almost equal or equal to hill climbing algorithm and higher payoffs than uniform policy in all cases, which implies that defender have learned well.

On closer inspection, the author found that the joint convergence policies of the defender and the attacker have a characteristic where neither is a best response to the other. If the defender's policies are extracted from these results and run through AMPL optimizing the attacker's response to these policies, then the defender's true values can be obtained. See Figure 5 for these values, where the defender-policies of initial-RL

algorithm, which are leading to deceiving high payoffs, are taken and run on AMPL to get the true payoff of the defender when the attacker plays the true best response to its policy and validate the initial-RL algorithm.

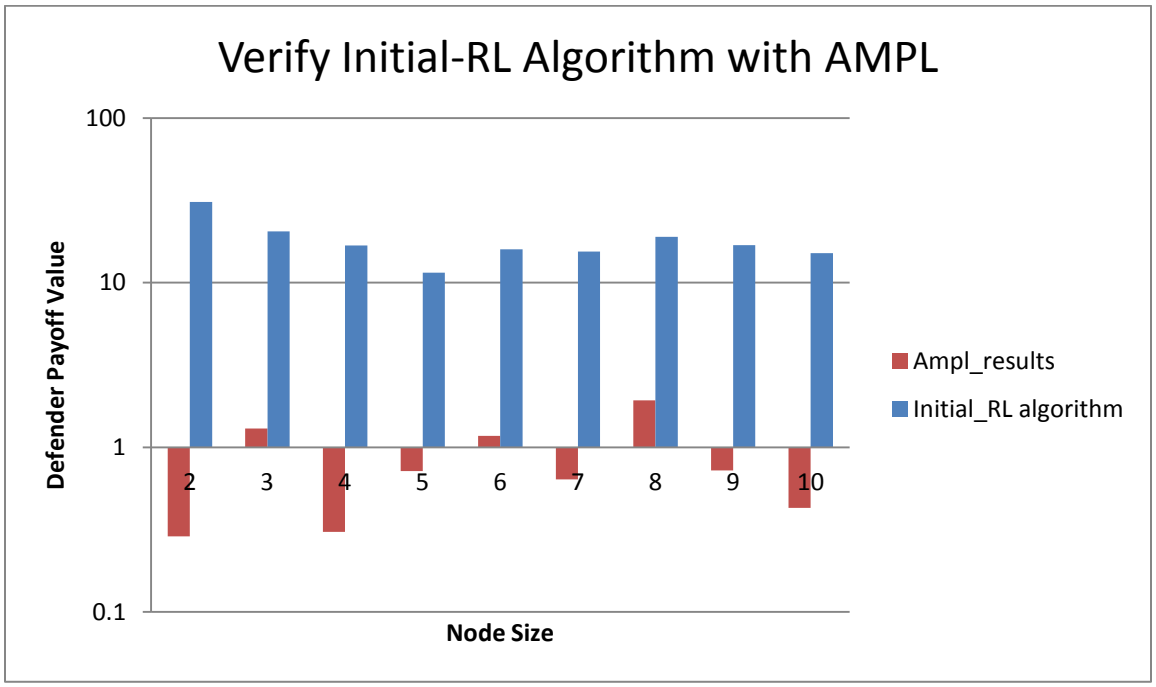

Figure 5. Verify Initial-RL algorithm with AMPL. Here, we can see that the actual defender payoffs resulted from AMPL are very low when compared to the payoffs given by initial-RL algorithm.

This confirms that the results of the initial-RL algorithm are deceiving as the defender policies of the initial-RL algorithm lead to poor defender payoffs, implying that the defender has not learned well and the high-payoffs of the initial-RL algorithm are only because the attacker have not learned a good response. This leads to an important conclusion that an ordinary concurrent RL algorithm fails to reach the Stackelberg equilibrium. In particular, the best response tendency of Q-learning is ill-suited for the problem of farsighted best response of the defender. So, to make the algorithm work the RL algorithm is modified by changing the way the defender policy is modeled and the way the defender payoffs are calculated, which lead to New RL Algorithm.

New RL Algorithm

Description

The key reasons for the failure of ordinary reinforcement learning are:
(a) myopic best response tends to forget (the defender's) better policies seen in the past.
(b) as the learning rate tapers down, policies change slowly which can sometimes make it hard for the learner to escape poor policies.

A standard technique for addressing (b) is perturbation [10]. A learner's policy is shaken out of a narrow neighborhood in order to create the opportunities for exploring better policies. This strategy is reminiscent of the random restart strategy of hill climbing, except that while random restart can move a policy arbitrarily, perturbation usually limits the shift to a bounded neighborhood of the current policy. The size of this bounded neighborhood is further reduced with time, in order to allow our RL algorithm to converge. Perturbation also bears resemblance to neighbor evaluation of hill climbing. The important distinction is that hill climbing evaluates many neighbors of a policy before a shift, while an online RL agent can only evaluate its current policy (and none other) and can only shift to (and hence evaluate) one neighboring policy.

In order to address (a), our RL defender agent simply maintains a reference policy, π_{ref}. Before a perturbation, the learner's current policy is compared to the current reference policy, and saved in the latter if it is better. This way the reference policy always maintains the best policy that the RL defender has seen so far.

New RL algorithm, same as of initial RL algorithm, also uses previously described learning agent algorithm and have the same functions as that of the initial RL algorithm with piSample() in addition to those of learning agent algorithm functions like

argmaxQ(), where argmaxQ() is used to select the best action for the attacker based on Q-Update values and piSample() returns the defender's best action depending on Pi-Updates. The learning rates of the new RL algorithm, unlike to that of the initial-RL algorithm, are updated after every episode as:

$$\alpha(t) = \frac{\alpha_0}{(1 + t/100)}$$

where α_0 is the initial learning rate and t is the episode number. β is updated same as α.

A major enhancement made to the new RL algorithm is that it calls AMPL to get the payoff for the defender, instead of relying on γ_Rewards. AMPL calls are made to get payoff for the defender policy modeled by RL algorithm after every episode. Initially the new RL algorithm starts with a uniform defender policy and then all of its neighbors are explored, each during an episode. The resulted defender policy by RL algorithm after every episode is modified by applying perturbation to that policy. The resulting defender policy payoffs are then compared and the defender policy leading to the highest payoff is chosen.

The new RL algorithm, with the above described enhancements, calls the learning agent for both attacker and defender. Episode count is up to 100,000 and step-size up to 1000. There is no training and testing as that in the initial-RL algorithm, but only learning where we simply let the agent learn, allowing it to gain a better policy. Calls to the learning agent is made in the same way as that of the initial-RL algorithm with α and β rates being gradually decreased as we reach the maximum episode count, making the rates zero by the end of final episode. After 100,000 episodes AMPL is called with the resulted defender policy to get defender payoff and is saved for later comparison. The

policy resulting better payoff than previous is saved as π_{ref} with initial π_{ref} as uniform policy.

Now perturbation is done to the defender policy and call to the learning agent is made again for 100,000 episodes. This continues with an initial perturbation rate as 1, until the rate becomes <= 0.01 with gradual decrease by a factor of 0.9. At every point of perturbation, π_{ref} is updated only if the current payoff π_{cur} of defender is greater than the payoff of previous π_{ref}. At the end, after all the neighbors of the uniform defender policy are explored, the policy of the neighbor leading to a better payoff than the other neighbors is resulted. Pseudo code of this new RL algorithm is given below:

```
//Pseudo-code of New RL algorithm
    readDatFile(datFileName)
    //to read required data like OSize, LSize, LT, OT, OMap, RA and RD
    //from given data file
    Q_at , Q_def ← 0
    π_cur, π_ref ← Uniform Policy
    r =1 //perturbation radius
    //getting the defender payoff value for uniform defender policy
    v(π_ref) = callAMPL(π_cur)
    while  r ≥ 0.01
        Q_at , Q_def ← 0
        attackerAgent = initializeLA( ) //Learning agent for attacker
        defenderAgent = initializeLA( ) //Learning agent for defender
        callLA( )
```

//same as in Initial RL algorithm with loop up to 100,000

//but no γ- rewards, also callLA( ) method modifies π_{cur}

$v(\pi_{cur})$ = callAMPL(π_{cur})

if $v(\pi_{cur}) > v(\pi_{ref})$//updating the better defender policy into π_{ref}

$\pi_{ref} \leftarrow \pi_{cur}$

$v(\pi_{ref}) \leftarrow v(\pi_{cur})$

//perturbing $\pi_{cur}$with r and normalizing π_{cur}as well

$\pi_{cur} \leftarrow perturbation(\pi_{cur}, r)$

$r \leftarrow r * 0.9$

// end of while loop

// output the better defender policy among explored points

print π_{ref}

Output Analysis

The new RL algorithm is expected to give the defender a better payoff than that of the fixed uniform policy as the new RL algorithm is initialized with the uniform defender policies. The new RL algorithm was run on the same set of 180 different data files that are used to analyze the output of the initial-RL algorithm and the average of the resulted defender payoff for every 20 files, having the same OSize and LSize are plotted (see Figure 6).

The plot in Figure 6 shows that the results of the new RL algorithm are good and also the use of AMPL calls assure that the algorithm is valid in producing better defender policies leading to better defender payoffs. This confirms that the new RL algorithm is a successful enhancement of the initial-RL algorithm. The performance analysis of the new

RL algorithm is done by comparing it with the uniform policy and the hill climbing algorithm in terms of the defender's payoff value and time, which is described in Chapter V.

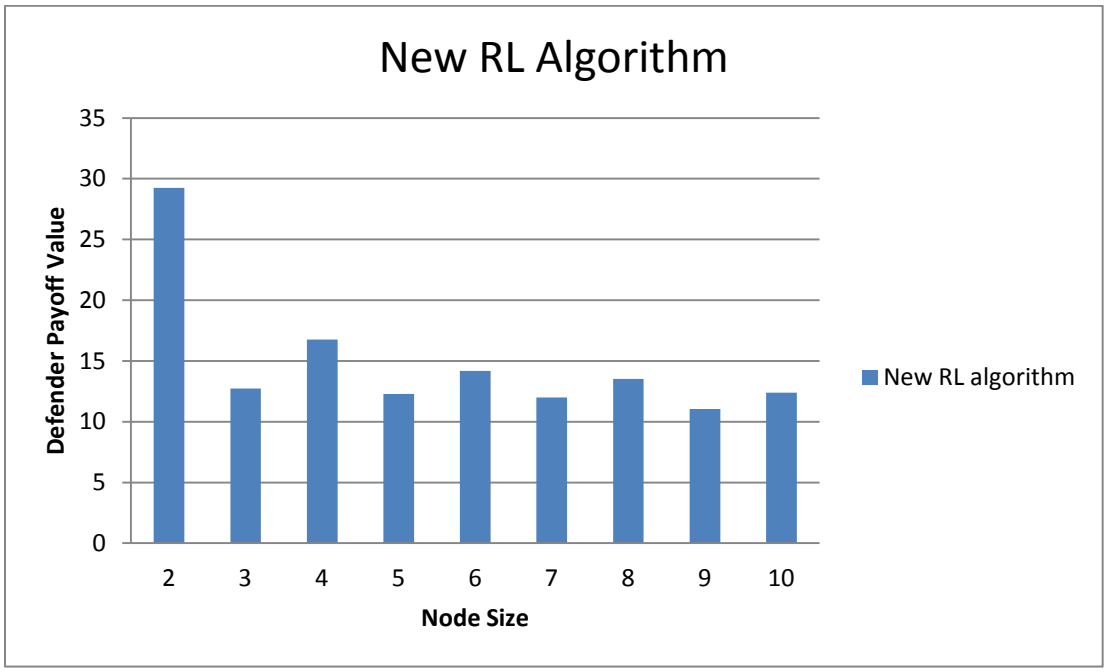

Figure 6. New RL Algorithm. The plot gives the resulted payoff values of the defender by the new RL algorithm, averaged over every 20 data files having the same node size.

CHAPTER V

COMPARISONS FOR PERFORMANCE ANALYSIS

Value

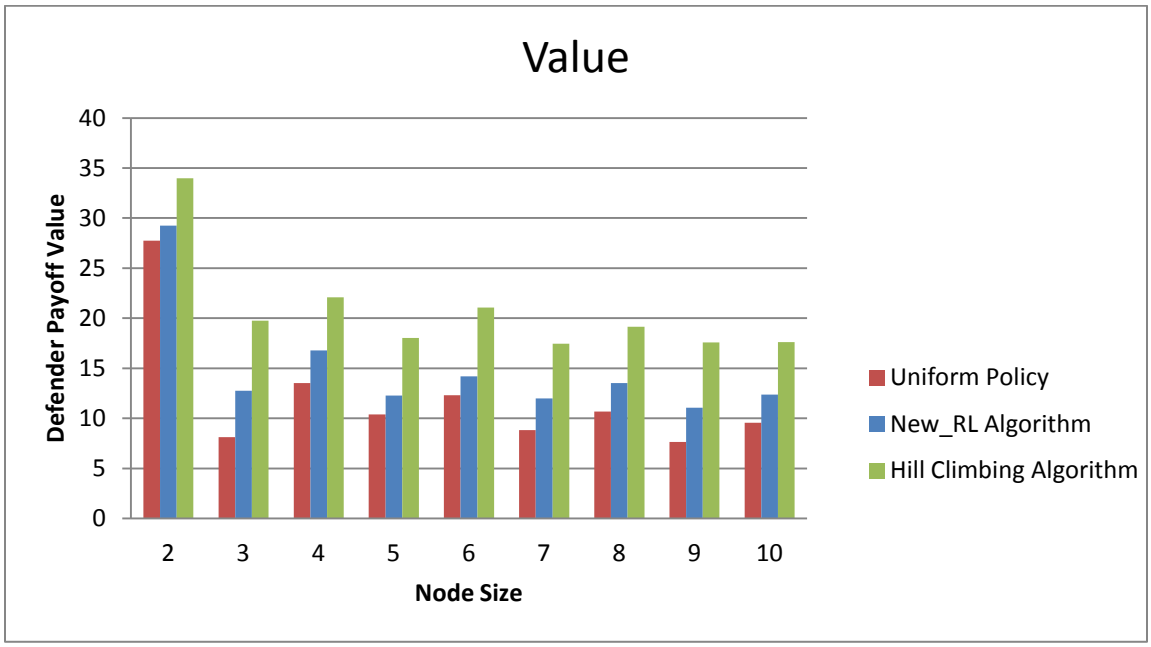

Figure 7. Comparison of Algorithms in terms of values. We compare the defender payoff values resulted by new RL algorithm with those of uniform policy and hill climbing algorithms. All the three algorithms run on the same set of different data files and are averaged over every 20 files for each node size, 2 to 10.

This comparison plot shows that the new RL algorithm gives the defender a better payoff than that of the fixed uniform policy in all cases. Also, the new RL algorithm is comparable to that of the hill climbing algorithm with the new RL algorithm generating the defender's payoff values which seem to shadow the values of the hill climbing algorithm. This value difference does not grow exponentially with problem size. Hence it can be said that the new RL algorithm produces values that lies between the uniform policy and the hill climbing algorithms. Furthermore, RL's domination of uniform policy means that learning is indeed a useful approach for this problem.

Time

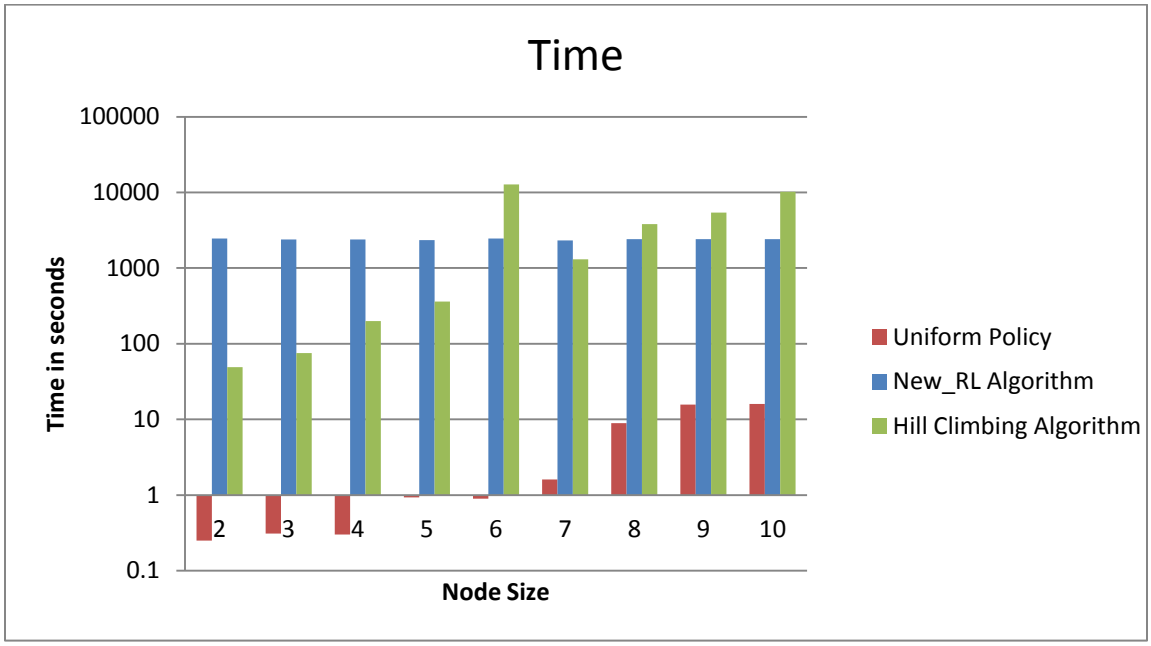

Figure 8. Comparison of Algorithms in terms of time. We compare the time taken by the three algorithms to produce the resulting defender payoff values. The algorithms are run on the same set of different data files with node size varying from 2 to 10.

There is an important observation that can be noticed in the above plot that the new RL algorithm takes almost same amount of time for any node size, which implies that it maintains a constant time for given episode count and step size. It means that the RL algorithm's time can be set independently of the problem size without affecting the its performance. The times of both uniform policy and hill climbing algorithms, on the other hand, increase with the node size and the increase is exponential. For larger (>10) problems, the hill climbing algorithm will take impractically large time and even may not be completed successfully. Whereas the new RL algorithm is reliable in producing results within the same amount of time even for larger problems.

CHAPTER VI

CONCLUSION AND FUTURE SCOPE FOR RESEARCH

The thesis can be concluded saying both uniform policy and hill climbing algorithm's time grow exponentially, whereas the RL algorithm with a fixed time produces values that lies between the uniform policy and the hill climbing algorithms. The net benefit of our approach is that we can get epsilon-close to the hill climbing algorithm in values without exponential growth in time.

In future the RL algorithm can still be improvised in terms of time as well as value to make the RL algorithm results reach or may even cross those of hill climbing algorithm with less time and being time independent of the problem size.

APPENDIX A

SOURCE CODE OF RL ALGORITHM

```
//NewLearningAgent.java

package securitygames.NewRL.RLwithAMPL;

public class NewLearningAgent {

     boolean m_learning,jointStateVisible;
     private int m_lastAction, m_numActions, m_lastState,
m_lastIndv_numActions ;
     private int m_episode_cnt, tot_numEpisodes;
     private double m_alpha, m_beta, m_gamma, m_explor;
     public double alpha, beta;
     private NewHashKeyClass htable,pitable;

     int range_rand(int maxRange) {
          return (int) (Math.random() * maxRange);
     }

     public NewLearningAgent(int numActions, int
numEpisodes, boolean learning,
               double epsilon, double alp, double
bet,double gamma, NewHashKeyClass htab, boolean visibility)
{//Initialize Attacker
          m_numActions = numActions;
          tot_numEpisodes = numEpisodes;
          m_learning = learning;
          alpha=alp;
          beta=bet;
          m_gamma=gamma;
          m_explor = epsilon;
          m_episode_cnt = 0;
          htable=htab;
          jointStateVisible=visibility;
     }

     public NewLearningAgent(int numActions, int
numEpisodes, boolean learning,
               double epsilon, double alp, double
bet,double gamma, NewHashKeyClass htab, NewHashKeyClass
pitab, boolean visibility) {//Initialize Defender
          m_numActions = numActions;
          tot_numEpisodes = numEpisodes;
```

```
          m_learning = learning;
          alpha=alp;
          beta=bet;
          m_gamma=gamma;
          m_explor = epsilon;
          m_episode_cnt = 0;
          htable=htab;
          pitable=pitab;
          jointStateVisible=visibility;
     }

     public int startEpisode(int state, int
indv_numActions) {
          m_episode_cnt++;
          setAlphaBeta();
          m_lastAction =
selectAction(state,indv_numActions);
          m_lastState = state;
          m_lastIndv_numActions=indv_numActions;
          return m_lastAction;
     }

     public int step(double reward, int state, int
indv_numActions) {
          double Q1[] = new double[m_numActions];
          double Q2[] = new double[m_numActions];
          double Q3[] = new double[m_numActions];

          Q1 = (double[]) htable.get(m_lastState);
          double delta = reward - Q1[m_lastAction];

          int save_last_action = m_lastAction;
          m_lastAction =
selectAction(state,indv_numActions);

          Q2 = (double[]) htable.get(state);
          delta += (m_gamma*Q2[m_lastAction]);

          if (m_learning) {
               //Q Update
               for(int i=0;i<m_numActions;i++){
                    if(i!=save_last_action)
                         Q3[i]=Q1[i];
               }

     Q3[save_last_action]=Q1[save_last_action]+((m_alpha *
delta));
```

```
                htable.put(m_lastState, Q3);
                if(!jointStateVisible)
                    piUpdate(save_last_action,Q3);
            }
            m_lastState = state;
            m_lastIndv_numActions=indv_numActions;
            return m_lastAction;
        }

        public void endEpisode(double reward) {
            double Q[] = new double[m_numActions];
            if (m_learning) {
                //Q Update
                Q = (double[]) htable.get(m_lastState);
                double delta = reward - Q[m_lastAction];
                Q[m_lastAction] += (m_alpha * delta);
                htable.put(m_lastState, Q);
                if(!jointStateVisible)
                    piUpdate(m_lastAction,Q);
            }
            if(m_episode_cnt==tot_numEpisodes){
                m_episode_cnt=0;
            }
        }

        private void piUpdate(int save_last_action, double[]
Q){
            double P1[] = new double[m_numActions];
            double P2[] = new double[m_numActions];
            double P3[] = new double[m_numActions];
            P1 = (double[]) pitable.get(m_lastState);

            //Pi Update
            double incP=0,sumPQ=0,sumP=0;
            for(int i=0;i<m_lastIndv_numActions;i++){
                sumPQ+=(P1[i]*Q[i]);
            }
            for(int i=0;i<m_numActions;i++){
                if(i!=save_last_action){
                    P3[i]=P1[i];
                    sumP+=P3[i];
                }
            }

        incP=(m_beta*P1[save_last_action]*(Q[save_last_action]
-sumPQ));
```

```
            if((P1[save_last_action]+incP)<0){
                P3[save_last_action]=0;
            }else if((P1[save_last_action]+incP)>1){
                P3[save_last_action]=1;
            }else{

      P3[save_last_action]=(P1[save_last_action]+incP);
            }
            sumP+=P3[save_last_action];
            //Normalize
            for(int i=0;i<m_numActions;i++){
                P2[i]=P3[i]/(double)(sumP);
            }
            pitable.put(m_lastState, P2);
      }

      private int selectAction(int state,int
indv_numActions) {
            int action;
            if(m_learning){
                // Epsilon-greedy
                if (Math.random() < m_explor) { /*explore*/
                    action = range_rand(indv_numActions);
                    //System.out.println("PICKED RANDOM");
                }else if(jointStateVisible){
                    action =
argmaxQ(state,indv_numActions);
                }else{
                    action = prob(state,indv_numActions);
                }
            }else if(jointStateVisible){
                action = argmaxQ(state,indv_numActions);
            }else{
                action = prob(state,indv_numActions);
            }
            return action;
      }

      private int argmaxQ(int state,int indv_numActions) {
            int bestAction = 0;
            double Q[] = new double[m_numActions];
            Q = (double[]) htable.get(state);
            double bestValue = Q[bestAction];
            int numTies = 0;
            for (int a = bestAction + 1; a < indv_numActions;
a++)            {
                double value = Q[a];
```

```
                if (value > bestValue) {
                     bestValue = value;
                     bestAction = a;
                } else if (Math.abs(value-
bestValue)<0.00001){
                     numTies++;
                     if (range_rand(numTies + 1) == 0) {
                          bestValue = value;
                          bestAction = a;
                     }
                }
           }
           return bestAction;
     }

     private int prob(int state,int indv_numActions) {
           int bestAction = 0;
           double randFraction;
           double P[] = new double[m_numActions];
           P = (double[]) pitable.get(state);

           randFraction=Math.random();
           for(int i=0;i<indv_numActions;i++){
                double sum=0;
                for(int j=i;j>=0;j--){
                     sum+=P[j];
                }
                if(randFraction<sum){
                     bestAction=i;
                     break;
                }
           }
           return bestAction;
     }

     private void setAlphaBeta() {
           double t_100=tot_numEpisodes/(double)100;

     m_alpha=alpha/(double)(1+(m_episode_cnt/(double)t_100)
);

     m_beta=beta/(double)(1+(m_episode_cnt/(double)t_100));
     }

}//end of NewLearningAgent class
```

```
//NewHashKeyClass.java

package securitygames.NewRL.RLwithAMPL;

import java.util.Hashtable;

public class NewHashKeyClass extends Hashtable<Integer,
double[]> {

     private static final long serialVersionUID = 1L;
     public double[] Q;

     public NewHashKeyClass(int initialCapacity, int LSize,
int OSize) {//For Attacker hash table
          // TODO Auto-generated constructor stub
          Q = new
double[initialCapacity];//initialCapacity=max no.of actions
          for(int i=0;i<initialCapacity;i++){
               Q[i]=0.0;
          }
          for (int n = 1; n <= LSize; n++) {//
               for (int i = 1; i <= OSize; i++) {
                    String state = Integer.toString(n) +
Integer.toString(i);
                    this.put(Integer.parseInt(state), Q);
               }
          }
     }

     public NewHashKeyClass(int initialCapacity, int OSize)
{//For Defender hash table
          // TODO Auto-generated constructor stub
          Q = new
double[initialCapacity];//initialCapacity=max no.of actions
          for(int i=0;i<initialCapacity;i++){
               Q[i]=0.0;
          }
          for (int i = 1; i <= OSize; i++) {
               this.put(i, Q);
          }
     }

     public NewHashKeyClass(int maxNumActions, int OSize,
int[]actNum) {//For Defender Pi table
          // TODO Auto-generated constructor stub
          for (int i = 1; i <= OSize; i++) {
```

```
                double[] Q1 = new
double[maxNumActions];//initialCapacity=max no.of actions
                for(int n=0;n<maxNumActions;n++){
                     Q1[n]=0.0;
                }
                for(int j=0;j<actNum[i];j++){
                     Q1[j]=(1/(double)actNum[i]);

     //System.out.println("actNum["+i+"]="+actNum[i]);

     //System.out.println("Q1["+j+"]="+Q1[j]);
                }
                put(i, Q1);
          }
     }

     public void display(NewHashKeyClass htable,int
maxNumActions, int LSize, int OSize){//Attacker display
          double[] R=new double[maxNumActions];
          for (int n = 1; n <= LSize; n++) {
                for (int i = 1; i <= OSize; i++) {
                     String state = Integer.toString(n) +
Integer.toString(i);
                     R=htable.get(Integer.parseInt(state));
                     for(int j=0;j<maxNumActions;j++){

     System.out.println(Integer.parseInt(state)+"\t"+j+"\t"
+R[j]);
                     }
                }
          }
     }

     public void display(NewHashKeyClass htable,int
maxNumActions, int OSize){//Defender display
          double[] R=new double[maxNumActions];
          for(int i=1;i<=OSize;i++){
                R=htable.get(i);
                for(int j=0;j<maxNumActions;j++){
                     System.out.println(i+"\t"+j+"\t"+R[j]);
                }
          }
     }

}//end of NewHashKeyClass
```

```
//NewRLClasss.java

package securitygames.NewRL.RLwithAMPL;

import java.io.IOException;
class NewRLClass {
     public static void main(String args[]) throws
IOException{
          NewManager newMngr=new NewManager();
          newMngr.managerMainFun(args);
     }
}//end of NewRLClass


//NewManager.java

package securitygames.NewRL.RLwithAMPL;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.io.RandomAccessFile;
import java.util.Random;
import java.util.Scanner;
import java.util.regex.MatchResult;

public class NewManager {
     int sessionNumber=0, maxSteps=0, att_alpha_rate,
att_beta_rate;
     double epsilon=0.01, alpha, beta, gamma=0.9;
     int LSize=0, OSize=0;
     int[][] LT, OT;
     int[] LT_ActNum, OT_ActNum;
     int[][] RA, RD;
     Random randAt,randDef;
     File txtfile;
     FileOutputStream txtfos;
     PrintStream txtps;
     int attackerState, defenderState, jointState=0,
attackerAction, defenderAction,
numAt_actions=0,numDef_actions=0;
     double attackerReward, defenderReward;
     double startTime=0,finishTime=0;
```

```
     boolean attackerLearn=true, defenderLearn=true;
     NewLearningAgent learnAttacker, learnDefender;
     NewHashKeyClass attackerHashTable,
defenderHashTable,defenderPiRefTable,defenderPiCurTable;

     public void managerMainFun(String[] args)throws
IOException{
          String datFileName, paramXLoc, amplScriptLoc;
          int runNumber=0;
          double radius=1,val_PiRef=0,val_PiCur=0;
          randAt=new Random();
          randDef=new Random();

          datFileName=args[0];
          sessionNumber=Integer.parseInt(args[1]);
          maxSteps=Integer.parseInt(args[2]);
          alpha=Double.parseDouble(args[3]);
          att_alpha_rate=Integer.parseInt(args[4]);
          beta=Double.parseDouble(args[5]);
          att_beta_rate=Integer.parseInt(args[6]);

          txtfile = new File(args[7]+datFileName+".txt");
          paramXLoc=args[8];
          amplScriptLoc=args[9];

          try {
               txtfos = new FileOutputStream(txtfile);
          } catch (FileNotFoundException e) {
               // TODO Auto-generated catch block
               System.out.println("Error in fileoutput
stream");
               e.printStackTrace();
          }
          txtps = new PrintStream(txtfos);
          System.setOut(txtps);

     readDatFile("securitygames/domains/"+datFileName+".dat
");

          startTime=System.currentTimeMillis();

          defenderPiRefTable = new
NewHashKeyClass(numDef_actions, OSize,OT_ActNum);
          defenderPiCurTable = new
NewHashKeyClass(numDef_actions, OSize,OT_ActNum);
```

```
          File f=   writeParamX(datFileName,
defenderPiRefTable, paramXLoc);
          val_PiRef=runAMPL(datFileName, amplScriptLoc);
          if(f.exists())
               f.delete();
          System.out.println("**Initial in NewManager
val_PiRef="+val_PiRef);

          while(radius>=0.01){

     System.out.println("################################");
               initializeTables_and_LA();

               //Training

     for(runNumber=1;runNumber<=sessionNumber;runNumber++){
                    callLA();
               }
               File fl=writeParamX(datFileName,
defenderPiCurTable, paramXLoc);
               val_PiCur=runAMPL(datFileName,
amplScriptLoc);
               if(fl.exists())
                    fl.delete();
               if(val_PiCur>val_PiRef){
                    val_PiRef=val_PiCur;

     defenderPiRefTable.putAll(defenderPiCurTable);
               }
               System.out.println("**in NewManager
val_PiCur="+val_PiCur);
               System.out.println("**in NewManager
val_PiRef="+val_PiRef);
               //System.out.println("After Learning");
               //displayTables();
               perturbation(radius);
               //System.out.println("After Perturbation");
               //displayTables();
               radius=radius*(0.9);
          }
          finishTime=System.currentTimeMillis();
          System.out.println("**********************\nFinal
Value: "+val_PiRef);
          System.out.println("Time: "+(finishTime-
startTime)/(double)1000);
          System.out.println("Defender Policy: ");
```

```
     defenderPiRefTable.display(defenderPiRefTable,numDef_a
ctions, OSize);
          txtps.close();txtfos.close();
     }

     private void perturbation(double radius){
          //NewHashKeyClass perturbPiCur=new
NewHashKeyClass(numDef_actions, OSize,OT_ActNum);
          for(int i=1;i<=OSize;i++){
               double sum=0;
               double[] p=new double[numDef_actions];
               p=defenderPiCurTable.get(i);
               for(int j=0;j<OT_ActNum[i];j++){
                    p[j]=p[j]+radius;
                    sum+=p[j];
               }
               for(int j=0;j<OT_ActNum[i];j++){
                    p[j]=p[j]/(double)sum;
               }
               defenderPiCurTable.put(i, p);
               //perturbPiCur.put(i,p);
          }
          //defenderPiCurTable.putAll(perturbPiCur);
          System.out.println("$$$Perturbation is done$$$");
     }

     private void initializeTables_and_LA(){
          //System.out.println("Initializing Tables");
          attackerHashTable=new
NewHashKeyClass(numAt_actions, LSize, OSize);
          defenderHashTable = new
NewHashKeyClass(numDef_actions, OSize);
          //displayTables();
          learnAttacker = new
NewLearningAgent(numAt_actions, sessionNumber,
attackerLearn, epsilon, att_alpha_rate*alpha,
att_beta_rate*beta, gamma, attackerHashTable,true);
          learnDefender = new
NewLearningAgent(numDef_actions, sessionNumber,
defenderLearn, epsilon, alpha, beta, gamma,
defenderHashTable,defenderPiCurTable,false);
          System.gc();
     }

     private void displayTables(){
          System.out.println("Att_Hash_Table");
```

```
     attackerHashTable.display(attackerHashTable,numAt_acti
ons, LSize, OSize);
          System.out.println("Def_Hash_Table");

     defenderHashTable.display(defenderHashTable,numDef_act
ions, OSize);
          System.out.println("Def_PiRef_Table");

     defenderPiRefTable.display(defenderPiRefTable,numDef_a
ctions, OSize);
          System.out.println("Def_PiCur_Table");

     defenderPiCurTable.display(defenderPiCurTable,numDef_a
ctions, OSize);
     }

     private void callLA(){
          int stepNum=0;
          //generating start states randomly
          defenderState= randDef.nextInt(OSize)+1;
          attackerState= randAt.nextInt(LSize) + 1;
          setJointState();

          //Start Episodes

     attackerAction=learnAttacker.startEpisode(jointState,L
T_ActNum[attackerState]);

     defenderAction=learnDefender.startEpisode(defenderStat
e,OT_ActNum[defenderState]);
          doAction();

          //Step
          for(stepNum=1;stepNum<=maxSteps;stepNum++){

     attackerAction=learnAttacker.step(attackerReward,
jointState,LT_ActNum[attackerState]);

     defenderAction=learnDefender.step(defenderReward,
defenderState,OT_ActNum[defenderState]);
               doAction();
          }

          //End Episode
          updateRewards();
          learnAttacker.endEpisode(attackerReward);
```

```
          learnDefender.endEpisode(defenderReward);
     }

     private void doAction(){
          updateRewards();//rewards based on current state
not current action
          updateStates();//when actions are performed
          setJointState();
     }

     private void updateRewards(){
          //rewards based on current state not current
action
          attackerReward=RA[defenderState][attackerState];
          defenderReward=RD[defenderState][attackerState];
     }

     private void updateStates(){

     attackerState=(LT[attackerState][attackerAction]);

     defenderState=(OT[defenderState][defenderAction]);
     }

     private void setJointState(){
          String state = Integer.toString(attackerState) +
Integer.toString(defenderState);
          jointState=Integer.parseInt(state);
     }

     private File writeParamX(String datFileName,
NewHashKeyClass defenderPiTable, String paramXLoc) throws
IOException{
          String
parmXFile="securitygames/"+paramXLoc+"/paramX"+datFileName+
".dat";
          File f = new File(parmXFile);
          FileOutputStream fos = null;
          PrintStream ps;
          try {
               fos = new FileOutputStream(f);
          } catch (FileNotFoundException e) {
               // TODO Auto-generated catch block
               System.out.println("Error in writeParamX
fileoutput stream");
               e.printStackTrace();
          }
```

```
          ps = new PrintStream(fos);
          System.setOut(ps);
          System.out.println("\n");
          System.out.println("param x:");
          for(int i=1;i<=OSize;i++){
               System.out.print("\t"+i);
          }
          System.out.print("\t:=");
          for(int j=1;j<=OSize;j++){
              System.out.print("\n"+j);
              int k=0;
              if(j==3){
               System.out.print("\t"+"0.0");
              }
              double[] d=new double[numDef_actions];
              d=defenderPiTable.get(j);
              while(k<OSize){
               if(j==3 && k==2)
                    break;
               System.out.print("\t"+d[k]);
               k++;
              }
          }//for j
          System.out.print(" ;");
          System.setOut(txtps);
          ps.close();fos.close();
          return f;
     }//writeParamX

     private double runAMPL(String datFileName, String
amplScriptLoc) throws IOException{
          double defValue=-1;
          Runtime rt=Runtime.getRuntime();
          String
scriptFile="securitygames/"+amplScriptLoc+"/scriptFor"+datF
ileName+".run";
          String[] cmd={"ampl",scriptFile};
          Process proc=rt.exec(cmd);
          BufferedReader stdInput = new BufferedReader(new
InputStreamReader(proc.getInputStream()));
          BufferedReader stdError = new BufferedReader(new
InputStreamReader(proc.getErrorStream()));
          // read the output from the command
         //System.out.println("****Here is the output of
ampl:\n");
         String s1 = null,s2="defender = ";
         while ((s1 = stdInput.readLine()) != null) {
```

```
          //System.out.println(s1);
          if(s1.contains(s2)){
               String s3=s1.substring(s1.indexOf('=')+2);
               defValue=Double.parseDouble(s3);
               System.out.println("$$defValue="+defValue);
          }
         }
         // read any errors from the attempted command
         System.out.println("****Here is the error of ampl
(if any):\n");
         while ((s1 = stdError.readLine()) != null) {
          System.out.println(s1);
         }
         stdInput.close();stdError.close();rt.gc();
         return defValue;
     }//runAmpl

     //functions used for reading .dat file

     private void readDatFile(String datFileName) throws
IOException{
          int i,j,k,index=0;
          Scanner scan = null;
          RandomAccessFile raf = null;
          MatchResult mr;
          String[] word = new String[100];
          word[0]="LSize := ";
          word[1]="OSize := ";
          Scanner s = null;

          try {
                scan = new Scanner(new File(datFileName));
                raf = new
RandomAccessFile(datFileName,"r");
          } catch (FileNotFoundException e) {
               // TODO Auto-generated catch block
               System.out.println("Error in opening dat
file");
               e.printStackTrace();
          }
          //LSize=raf.read()-48;//ASCII of (0 to 9) digits
is (48 to 57)
          for(i=0;word[i]!=null;i++){
               if (scan.findWithinHorizon(word[i], 0) !=
null) {
                 mr = scan.match();
                 index=mr.end();
```

```
                }else{
                  System.out.println("Match is not found");
                }
                  raf.seek(index);
                  if(i==0){
                       s=new Scanner(raf.readLine());
                       LSize=readLastInt(s.next());
                       for(j=1;j<=LSize;j++){
                            word[1+j]="LT\\["+j+"\\] :=  ";
                       }
                       LT = new int[LSize+1][LSize+1];
                       LT_ActNum = new int[LSize+1];
                  }else if(i==1){
                       s=new Scanner(raf.readLine());
                       OSize=readLastInt(s.next());
                       for(j=1;j<=OSize;j++){
                            word[1+LSize+j]="OT\\["+j+"\\] :=
";
                       }
                       word[1+LSize+j]="RA:";
                       word[1+LSize+j+1]="RD:";
                       OT = new int[OSize+1][OSize+1];
                       OT_ActNum = new int[OSize+1];
                       RA = new int[OSize+1][LSize+1];
                       RD = new int[OSize+1][LSize+1];
                  }else if(i-1<=LSize){
                       k=0;
                       s=new Scanner(raf.readLine());
                       while(s.hasNextInt()){
                            LT[i-1][k]=s.nextInt();
                            k++;
                       }
                       LT[i-1][k]=readLastInt(s.next());
                       LT_ActNum[i-1]=k+1;
                       if(numAt_actions<k+1){
                            numAt_actions=k+1;
                       }
                  }else if((i-(1+LSize))<=OSize){
                       k=0;
                       s=new Scanner(raf.readLine());
                       while(s.hasNextInt()){
                            OT[i-(1+LSize)][k]=s.nextInt();
                            k++;
                       }
                       OT[i-
(1+LSize)][k]=readLastInt(s.next());
                       OT_ActNum[i-(1+LSize)]=k+1;
```

```
                    if(numDef_actions<k+1){
                         numDef_actions=k+1;
                    }
               }else if(i==(2+LSize+OSize)){
                    readRewards(RA,s,raf,index);
               }else if(i==(3+LSize+OSize)){
                    readRewards(RD,s,raf,index);
               }
          }//for loop
          s.close();
          raf.close();
          scan.close();
     }

     private void readRewards(int[][]array, Scanner s,
RandomAccessFile raf,int index) throws IOException{
          int i,j;
          raf.seek(index+2);//to skip next-line--\n
          raf.readLine();//skip Column indices
          for(i=1;i<=OSize;i++){
               s = new Scanner(raf.readLine());
               s.nextInt();//skip Row indices
               for(j=1;j<=LSize;j++){
                    if(i==OSize && j==LSize){
                         array[i][j]=readLastInt(s.next());
                    }else{
                         array[i][j]=s.nextInt();
                    }
               }
          }
     }

     private int readLastInt(String s1){
          int k=0;
          String s2="";
          do{
               s2=s2+s1.charAt(k);
               k++;
          }while(s1.charAt(k)!=';');
          return Integer.parseInt(s2);
     }
}//end of NewManager class
```

APPENDIX B

SAMPLE DATA FILES

```
#security_2_1.dat, a data file with node size 2

param LSize := 2;
param OSize := 2;

set LT[1] :=  2 1;
set LT[2] :=  1 2;


set OT[1] := 1 2;
set OT[2] := 1 2;


set OMap[1] := 1;
set OMap[2] := 2;


param RA:
     1    2    :=
1    -5   8
2    7    -5;


param RD:
     1    2    :=
1    7    0
2    0    5;


#security_5_16.dat, a data file with node size 5

param LSize := 5;
param OSize := 3;

set LT[1] :=  5 3 1;
set LT[2] :=  5 4 2;
set LT[3] :=  1 3;
set LT[4] :=  2 4;
set LT[5] :=  1 2 5;


set OT[1] := 1 2;
set OT[2] := 1 2 3;
```

```
set OT[3] := 2 3;


set OMap[1] := 1;
set OMap[2] := 2;
set OMap[3] := 3 4 5;


param RA:
     1    2    3    4    5    :=
1    -5   4    6    2    10
2    6    -5   7    8    10
3    10   7    -5   -5   -5;


param RD:
     1    2    3    4    5    :=
1    8    0    0    0    0
2    0    6    0    0    0
3    0    0    3    1    5;


#security_10_7.dat, a data file with node size 10

param LSize := 10;
param OSize := 3;

set LT[1] :=  8 1;
set LT[2] :=  5 4 2;
set LT[3] :=  4 3;
set LT[4] :=  3 2 4;
set LT[5] :=  2 7 5;
set LT[6] :=  9 10 6;
set LT[7] :=  5 9 7;
set LT[8] :=  10 1 8;
set LT[9] :=  6 7 9;
set LT[10] :=  6 8 10;


set OT[1] := 1 2;
set OT[2] := 1 2 3;
set OT[3] := 2 3;


set OMap[1] := 1 2 3;
set OMap[2] := 4 5 6;
set OMap[3] := 7 8 9 10;
```

```
param RA:
     1    2    3    4    5    6    7    8    9    10   :=
1    -5   -5   -5   10   8    7    10   8    4    7
2    10   4    6    -5   -5   -5   1    2    9    7
3    5    6    6    2    8    5    -5   -5   -5   -5;


param RD:
     1    2    3    4    5    6    7    8    9    10   :=
1    10   2    5    0    0    0    0    0    0    0
2    0    0    0    9    6    8    0    0    0    0
3    0    0    0    0    0    0    2    10   6    10;
```

REFERENCES

[1] Fudenberg D, Tirole J. Game Theory. Cambridge, MA: MIT Press; 1991. 573 p.

[2] Paruchuri P, Pearce JP, Tambe M, Ordonez F, Kraus S. An efficient heuristic approach for security against multiple adversaries. In: Proceedings of the 6th International Joint Conference on AAMAS; 2007 May; Honolulu (HI); P.14-18.

[3] Brown G, Carlyle M, Salmeron J, Wood K. 2006. Defending critical infrastructure. Interfaces 36(6): 530–544.

[4] Conitzer V, Sandholm T. Computing the optimal strategy to commit to. In: Proceedings of the ACM-EC; 2006; Ann Arbor (MI): Association for Computing Machinery; P.82-90.

[5] Kiekintveld C, Jain M, Tsai J, Pita J, Tambe M, Ordonez F. Computing optimal randomized resource allocations for massive security games. In: Proceedings of The 8th International Conference on AAMAS; 2009; P.689–696.

[6] Pita J, Jain M, Western C, Portway C, Tambe M, Ordonez F, Kraus S, Parachuri P. Deployed ARMOR protection: The application of a game-theoretic model for security at the Los Angeles international airport. In: Proceedings of The 7th International Conference on AAMAS; 2008 May; Estoril, Portugal; P.125–132.

[7] Tsai J, Rathi S, Kiekintveld C, Ordonez F, Tambe M. IRIS: a tool for strategic security allocation in transportation networks. In: Proceedings of The 8th International Conference on AAMAS; 2009 May; Budapest, Hungary; P.37-44.

[8] An B, Pita J, Shieh E, Tambe M, Kiekintveld C, Marecki J. 2011. Guards and protect: Next generation applications of security games. ACM SIGecom Exchanges 10(1): 31–34.

[9] Pita J, Tambe M, Kiekintveld C, Cullen S, Steigerwald E. Guards - game theoretic security allocation on a national scale. In: Proceedings of The 10th International Conference on AAMAS; 2011 May; Taipei, Taiwan; P.37-44.

[10] Sutton RS, Barto AG. Reinforcement Learning: An Introduction. Cambridge, MA: MIT Press; 1998. 313 p.