Pattern Analysis in Dynamic Social Networks

Senior Project submitted to

The Division of Science, Mathematics, and Computing

of

Bard College

by Yu Wu

Advisor: Dr. Sven Anderson, Bard College, Dr. Yu Zhang, Trinity University

May 2010

Annandale-on-Hudson, NY

to my parents

Acknowledgements

I would like to thank Dr. Sven Anderson for the invaluable guidance and teaching I received from him during the past year, Dr. Yu Zhang for leading me into the wonderful area of Multi-Agent System and all the extremely helpful comments she generously gave, and also Sining for being constantly encouraging and supportive. Without your help, I have no way to achieve what I have achieved today.

Abstract

In this project, we explore how decentralized local interactions of autonomous agents in a network relate to collective behaviors. Most existing work in this area models social network in which agent relations are fixed; instead, we focus on dynamic social networks where agents can rationally adjust their neighborhoods based on their individual interests. We propose a new connection evaluation rule called the Highest Weighted Reward (HWR) rule, with which agents dynamically choose their neighbors in order to maximize their own utilities based on the rewards from previous interactions. We show that in the 2-action pure coordination game, our system will stabilize to a clustering state where all relationships in the network are rewarded with the optimal payoff. Preliminary experiments verify this theory and also reveal additional interesting patterns in the network.

Menu

1. Introduction..1

2. Background and Related Work...3

3. Highest Weighted Reward (HWR) Neighbor Evaluation Rule.....................................7

    3.1 HWR Rule...7

    3.2 Discount Factor Range...10

    3.3 Pattern Predictions...11

4. Code...13

    4.1 Optimization...13

    4.2 Debugging Methodology...15

5. Experiment...18

    5.1 Program Flow Chart..18

    5.2 Experiment Environment...19

    5.3 Results..20

        5.3.1 Two-Cluster Stable Clustering State...20

        5.3.2 One-Cluster Stable Clustering State..22

        5.3.3 Pattern Possibilities with Various Network Sizes.............................23

        5.3.4 Threshold & Broken Connection Peak..26

        5.3.5 Scalability...30

        5.3.6 Comparison with Static Network...31

5.3.7 Tolerance....34

6. Future Work....36

7. Conclusion....38

Reference....39

Appendix A....40

Appendix B....44

1

Introduction

Along with the advancement of modern technology, computer simulation is becoming a more and more important tool in today's researches. The simulation of large scale experiments which originally may take people months or even years can now be run within minutes. Computer simulation has not only saved researchers a great amount of time but also enabled them to study many macro topics that were impossible to study experimentally in the past. In this situation, Multi-Agent System (MAS) has emerged as a new area that facilitates the study of large scale social networks.

Researchers have invented many classic networks, such as random network, scale-free network, small world network, to list a few. Although these networks have successfully modeled many social structures, they all share the weakness of being static. In today's world, many important virtual networks, such as e-commerce network, SNS, etc., have much less stable connections among agents and thus the network structures will constantly change. Undoubtedly, the classic networks will fail to capture the dynamics of these networks. Out of this concern, Jason Leezer invented the network model HCR (Highest Cumulative Reward) that enables agents to update their connections through a rational strategy (Zhang, 2009). In this way, any classic network can be easily transformed into a dynamic network through the HCR rule. In Leezer's model, agents

evaluate their neighbors based on their performance history, and all interactions in history are equally weighted. However, people may argue that the evaluation function is not very realistic since in the real world recent events may have a greater influence on people than long past ones. Motivated by this fact, the HWR (Highest Weighted Reward) rule was invented. Unlike HCR, HWR allows the agents to use a discount factor to devalue past interactions with their neighbors. The more recent the interaction is, the more heavily the corresponding reward is weighted in the evaluation function. In our research, we identified certain patterns from the simulation results and then demonstrated the existence of the pattern both theoretically and empirically. Later, we also compared the differences between our model and classic static network.

2

Related Work and Background

MAS is a fast growing sub-area of Artificial Intelligence. As the name implies, MAS usually consists of a set of intelligent agents that interact with each other through a certain network. The agents may have limited intelligence and memory so that they can adjust their future behaviors based on their environment. The simulation is run on various kinds of networks which are meant to simulate various social structures. Much work has been done in the area and there are many well-known results, such as the six-degree separation theory derived from the Small World Network. In reality, the simulation conducted on MAS can go into two directions: One is to build a model as realistic as possible for a specific scenario, such as the moving crowds in an airport. The other one is to build an abstract model that captures the essence of human activities and can be used to model many different social networks of the same kind. Our work here belongs to the latter category.

Research in this area usually focuses on the study of social norms. One of the most basic questions that can be asked about the social norms is whether the agents in the social network will eventually agree on one solution. Will the whole network converge on one solution? The results vary drastically in different networks and under different algorithms. The agents can converge into various patterns and with different time period.

Generally, there are two categories in this area studying the emergence of social norms. The first category studies social norms in static networks. The second studies the evolving or dynamic network.

In the first category of static network, one of the most significant findings was by Shoham and Tennenholtz, who proposed the HCR rule (Shoham, 1997). In each timestep, an agent adopting HCR rule switches to a new action if and only if the total rewards obtained from that action in the last time step are greater than the rewards obtained from the currently chosen action in the same time period. The authors simulated various test trials under different parameter configurations (memory update frequency, memory restart frequency, and both) to experiment agent on an agent's effectiveness in learning about their environment. They also theoretically proved that under HCR rule the network will always converge to one social norm.

Another important work done in the first category is the Generalized Simple Majority (GSM) rule proposed by Jordi Delgado (Delgado, 2002). Agents obeying the GSM rule will change to an alternative strategy if they have observed more instances of it on other agents than their present action. This rule generalizes the simple majority rule. As the randomness $\beta \rightarrow \infty$, the change of state will occur as soon as more than half of the agents are playing a different action. In that case, GSM will behave exactly as the simple majority rule does.

$$f_{\beta}(k_{\bar{s}}) = \frac{1}{1 + e^{2\beta(2k_{\bar{s}}/k - 1)}}.$$

Changing State Probability Defined in GSM

The second category of research on agents' social behavior is the study of evolutionary networks or dynamic networks, such as (Borenstein, 2003; Zimmermann, 2005). Here researchers investigate the possible ways in which an agent population may converge onto a particular strategy. The strong assumptions usually make the experiments unrealistic. For example agents often have no control over their neighborhood. Also, agents do not employ rational selfish reward maximizing strategies but instead often imitate their neighbors. Lastly, agents are often able to see the actions and rewards of their neighbors, which is unrealistic in many social settings.

In the second category of the research, Zhang and Leezer proposed the Highest Rewarding Neighborhood (HRN) rule (Zhang, 2009). The HRN rule allows agents to learn from the environment and compete in networks. Unlike the agents in (Borenstein, 2003), which can observe their neighbors' actions and imitate it, the HRN agents employ selfish reward maximizing decision making strategy and are able learn from the environment. Under the HRN rule, cooperative behavior emerges even though agents are selfish and attempt only to maximize their own utility. This arises because agents are able to break unrewarding relationships and therefore are able to maintain mutually beneficial relationships. This leads to a Pareto-optimum social convention, where all connections in the network are rewarding.

This paper proposes the HWR rule, which is extended from the HRN rule (Zhang, 2009). In HRN rule, the agent values its neighbors based on the all rewards collected from them in the past, and all interactions are weighted equally throughout the agent's history. However, this will cause the agent to focus too much on the history of the

neighbor and fail to respond promptly to the neighbor's latest action. Therefore, we introduce the HWR rule, which introduces a time discount factor that helps the agents to focus on the recent history.

3

Highest Weighted Reward (HWR) Neighbor Evaluation Rule

3.1

HWR Rule

The HWR rule is based on a very simple phenomena we experience in our everyday lives, that we human beings tend to value recent events more than events which happened a long time ago. To capture this feature, we introduce a time discount factor, which is usually smaller than 1, in order to linearly devalue the rewards collected from past interactions. (When time discount factor equals 1, the HWR rule will behave in the same way as the HRN rule does.) Notice that both HWR and HRN are neighbor evaluation rules, which means the purpose of these rules is to help the agents value their connections with neighbors more wisely. In other words, these rules can only make decisions regarding connection choosing, but will not influence the agent's action choosing decisions.

According to the HWR rule, an agent will maintain a relationship if and only if the weighted average reward earned from that relationship is no less than a specified percentage of the weighted average reward earned from every relationship. Now we

carefully go through one cycle of HWR evaluation step-by-step.

1. For each neighbor, we have a variable named $TotalReward$ to store the weighted total reward from that neighbor. In each turn, we update the $TotalReward$ through the following equation in which "c" represents the discount factor:

$$TotalReward = TotalReward \times c + RewardInThisTurn$$

2. In a similar way, we keep a variable named $GeneralTotalReward$ to store the weighted total reward the agent got from all neighbors in the history. In each turn, we update the $GeneralTotalReward$ through the following equation:

$$\begin{gathered} GeneralTotalReward = GeneralTotalReward \times c \\ + TotalRewardInThisTurn / NumOfNeighbors \end{gathered}$$

Here $TotalRewardInThisTurn$ needs to be divided by the $NumOfNeighbors$ in each turn in order to find the average reward per connection. Since we want to calculate the average reward of all interactions, the total reward in each round needs to be divided by the number of interactions in that turn.

3. When we start to choose non-rewarding neighbors, we first calculate the average reward for every agent. The $AvgReward$ is calculated in the following way: suppose the agent has been playing with the neighbor for n turns, then:

$$AvgReward = \frac{TotalReward}{1 + c + c^2 + ... + c^{n-1}} = \frac{TotalReward}{\frac{1 - c^{n-1}}{1 - c}}$$

4. Calculate the average total reward. Similar to step 3:

$$AvgTotalReward = \frac{GeneralTotalReward}{1 + c + c^2 + ... + c^{n-1}} = \frac{GeneralTotalReward}{\frac{1 - c^{n-1}}{1 - c}}$$

Then we compare the ratio of *AvgReward/AvgTotalReward* with the *threshold*. If the former is greater than the later, then we keep the neighbor; if not, we regard that agent as a bad neighbor. In this way, the agent can evaluate its neighbor with an emphasis on recent history.

The above explanation has assumed an ideal environment in which agents have infinity amount of memory for ease of exposition. In reality and our experiments, agents have a limited amount of memory and can only remember the interactions of a certain number of turns. For the complete the evaluation process see Appendix B.

3.2

Discount Factor Range

Generally speaking, the HWR rule can work with different games and in various networks. Since this rule just adds a dynamic factor into the network, it does not contradict other aspects of the network. However, in some special cases, a certain experiment setup may make some discount factor value meaningless. Assume we are playing pure coordination game with payoff table (C: Cooperate; D: Defect):

	C	D
C	1	0
D	0	1

Also assume the agents only want to keep the neighbors whose average reward is positive for them. Then in such a case, no matter how many rounds an agent has been playing with a neighbor, its most recent reward will still dominate the whole reward sequence. To be more specific, if an agent receives a reward of 1 from a neighbor in the last interaction but have constantly received -1 in the past *n* trials, the cumulative reward will be $1-c-c^2-...-c^n$. If *c* is too low, the cumulative reward will still be bigger than 0. We can find the range of *c* through the following derivation:

$$1-c-c^2-...-c^n>0$$
$$1>c+c^2+...+c^n$$
$$1>c(1+c^n)/(1-c)$$

Since when *n* approaches infinity, c^n approaches 0, then

$$1 > \frac{c}{1-c}$$
$$c < \frac{1}{2}$$

Therefore, we usually limit *c* in range of 0.5 to 1.

In fact, for most cases *c* will take a value much higher than 0.5. Since the past rewards were devalued exponentially, a small *c* value will cause the past rewards to be practically ignored very soon. In our common experiment setup, we let the agent have a memory size of 30. In this case, if $c = 0.5$, then we see the oldest action will be devalued by a factor of $0.5^{29} \approx 1.86\text{E-9}$, which makes the reward totally insignificant. In fact, even for a relatively high *c* value such as 0.9, the last factor will still be as small as $0.9^{29} \approx 0.047$. Therefore, just in case the past rewards will be devalued too heavily, we usually keep the *c* in the range of 0.8 to 1.

3.3

Pattern Predictions

Based on the HWR rule, we make some predictions about the outcome of the network pattern.

Argument: *Given a pure coordination game, placing no constraints on the initial choices of action by all agents, and assuming that all agents employ the HWR rule, then the following holds:*

For every $\epsilon > 0$ there exists a bounded number M such that if the system runs for M iterations then the probability that a stable clustering state will be reached is greater than $1 - \epsilon$.

If a stable clustering state is reached then the all agents are guaranteed to receive optimal payoff from all connections.

In order to further clarify the argument, the following definitions are given:

Stable Clustering State: A network reaches a stable clustering state when all the agents in the network belong to one and only one closed cluster.

Closed Cluster: A set of agents forms a closed cluster when there does not exist any connection between any agent in the set and another agent outside of the set, and also all the agents in the set are playing the same action.

Basically, what the argument claims is that the whole network will eventually converge into one single cluster of same action or two clusters with different actions, and once in such a state, the rewards collected from the network will be maximized. The logic behind the argument will be shown in the Appendix A.

4

Code

4.1

Optimization

In the first phase of the research, we ran experiments on a revised version of Leezer's original program. We changed several core functions in order to implement the HWR rule. The original work was written in Java and well done. We believe when Leezer wrote his code, his major goal was to provide a clear and open multi-agent platform that is easy to understand and easy to change. Fully utilizing the power of generics and inheritance in Java, his goal was fully achieved. After going through his codes several times, we edited his codes in some parts and made it ready for the new experiments without too much effort. However, Leezer's original program faced a trade-off between a clear, open structure and the performance of the program. For an experiment with a scale of thousands of agents, it usually takes hours to finish one trial, while in order to get convincing results, we usually need to get the average results of at least 30 trials. To carry out experiments under different setups as many times as possible, we usually run small test trials with 500 agents. Although in this way we were able to get many interesting

results in a relatively short time, one issue remained. For the patterns we observed in a small network, whether they would change or emerge in a slower/faster rate in a large network remained as a question. In order to get more convincing results and explore more possible outcomes, we had to build a platform that allows us to run larger scale experiments more quickly.

For the above reasons, we decided to rebuild the network model in C++. In fact, this rebuilding is more than a conversion from Java to C++; it represents a shift of design philosophy. Unlike the previous attempt, this time we tried to code in a way that places efficiency in the highest priority. The current C++ program may be less clear than Leezer's work and also harder to change, but it clearly outperformed the previous one. With the current C++ program, we were able to run one experiment trial with tens of thousands of agents within minutes. The performance improvement comes in two aspects. The first is speed. Efficiency improvement from Java to C++ brought a speed boost to the program. We also optimized several algorithms. Second is memory consumption. Since we are running experiments with scales up to a million agents, the program will have a very big memory consumption. If the memory consumption ever exceeds the capacity of the computer's ram size, the program will begin to access data from hard drive and that will greatly slow the whole program. While the original program used many objects and maps to store variables, this time we tried to use bits and short integers to store information as efficiently as possible. The results turned out to be very satisfying. Now on a computer with 8 GB ram, we are able to run a network of up to one million agents.

In fact, performance optimization should begin as early as the design level.

Whenever we design a new network model, we should intentionally avoid the design that will take up too much computation power or memory. For example, when Leezer ran his experiments with the HRN rule in a random network, every single agent in the network had a neighborhood of roughly 15% the size of the whole network. Later we argue this might not be a very realistic assumption. Since in many social networks, especially very large ones, one agent cannot reasonably expect to connect with 15% of the total agent population. For example, in a typical modern social network system (SNS), such as Facebook, it will be unlikely for one user to know 15% of all the registered users. Therefore, we began to limit this ratio to as small as 1% percent and as a result, the performance was drastically improved. Similar questions should be considered in the beginning phase of the design of a new model. Usually, we should make sure that a trial with ten thousand agents can be finished within ten minutes. In such a case, it is safe to say that the experiment part will not take too much time to get usable results, and the whole research program can finish in a reasonable time.

4.2

Debugging Methodology

Since the program contains many complicated calculations and procedures, it is prone to many bugs that can hardly be detected by common debugging methods. What is worse, the bugs usually will not cause the program to crash, but it will cause the whole

system to behave abnormally. During our research, we mainly used the following three methods to debug the program.

1. Run a test trial in a toy network. The first step to check whether everything works fine is to run a tiny network model step by step and manually check whether every parameter works right. One can start from a network with only 1 or 2 agents and print everything out at each time step, and check later whether every parameter has behaved in the way as expectation.

2. Run a relatively large network and focus on only one agent. Some bugs are easier to find in a large network. After finishing the first step of debugging, we should run a test trial with at least 1000 agents. Since it is manually impossible to check whether all agents behave correctly, we can keep checking on one agent and print out all of its parameters in each time step. Because all agents in the network adopt the same rule and play the same game, even a couple of agents can be a good representative of the whole agent set.

3. Run large scale experiments for many times and compare the outcomes with expected results. Even after the previous two steps, it is still possible for some bugs to remain hidden. Some bugs will cause the network to

behave abnormally once after a few dozens of trials. In order to find these bugs, we first need to make several theoretical predictions and then keep checking whether every experiment trial matches these predictions. Sometimes, this step can help to identify bugs that are very difficult to find, or to find a design flaw.

After the above three steps of bug-checking, the program should be free of critical bugs. However, it is still not safe to say that the whole program is bug free. Some bugs will cause the network to behave abnormally only in few rare situations. So far we do not have any good way to find these bugs except by running the experiments and checking the results as many times as possible. For this reason, whenever people observe a new pattern or a new phenomenon emerging in the network, they should also first check whether that is caused by a bug.

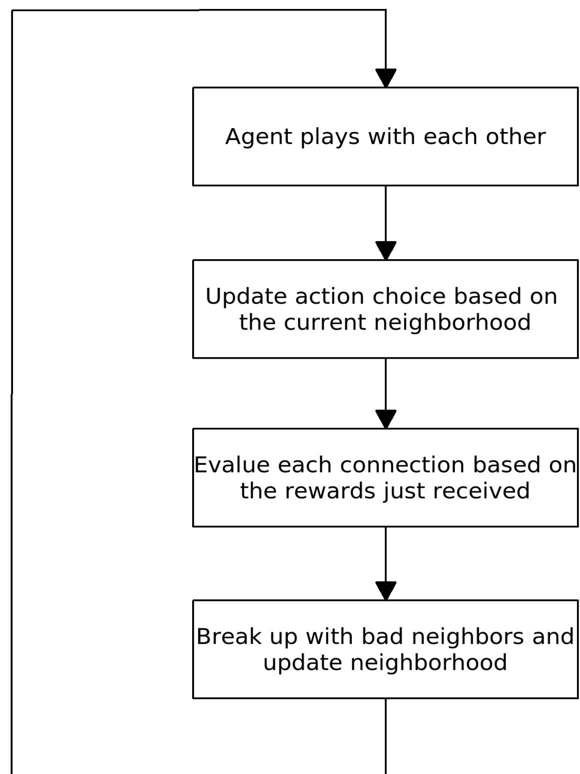
5

Experiment

5.1

Program Flow Chart

The operating procedure can be shown by the following flowchart:



5.2

Experiment Environment

All experiments are conducted in an environment that has the following properties:

1. All trials are run in a random network having 1000 to 10000 agents.

2. The number of connections in the network remains the same throughout the trial.

3. Every time a connection is broken, both agents have a 50% chance to gain the right to connect to a new neighbor. Exactly one of the two agents will find a new neighbor. This restriction guarantees the number of connections remains the same.

4. All agents have a limited memory size.

5. All agents adopt the ideal learning rule, which means the agent will always choose the action that the majority of its neighbors used in the last turn. If there is the same number of neighbors adopting different actions, the agent will not change its current action. In other words, the agents obey the simple majority rule.

All code was written, compiled and ran in Ubuntu 9.10 with GCC 4.4.1. One test trial with 10,000 agents and an average neighborhood size of 100 takes approximately 90 seconds to run for 30 time steps on my HP HDX-16 laptop. The laptop has an Intel Core 2 Dual-Core 2.13 GHz CPU and 3 GB RAM.

5.3

Results

5.3.1

Two-Cluster Stable Clustering State

First, we show the emergence of the stable clustering state. Since the final outcome can have two different patterns with either one or two final stable clusters, we present the patterns in two parts. First, let's check the situation where the network eventually converges into one single cluster. The following experiments are conducted with the parameters:

Number of Runs: 50
Time Steps: 30
Network Size: 5000
Neighborhood Size: 10
Reward Discount: 0.9
Threshold: 0.9

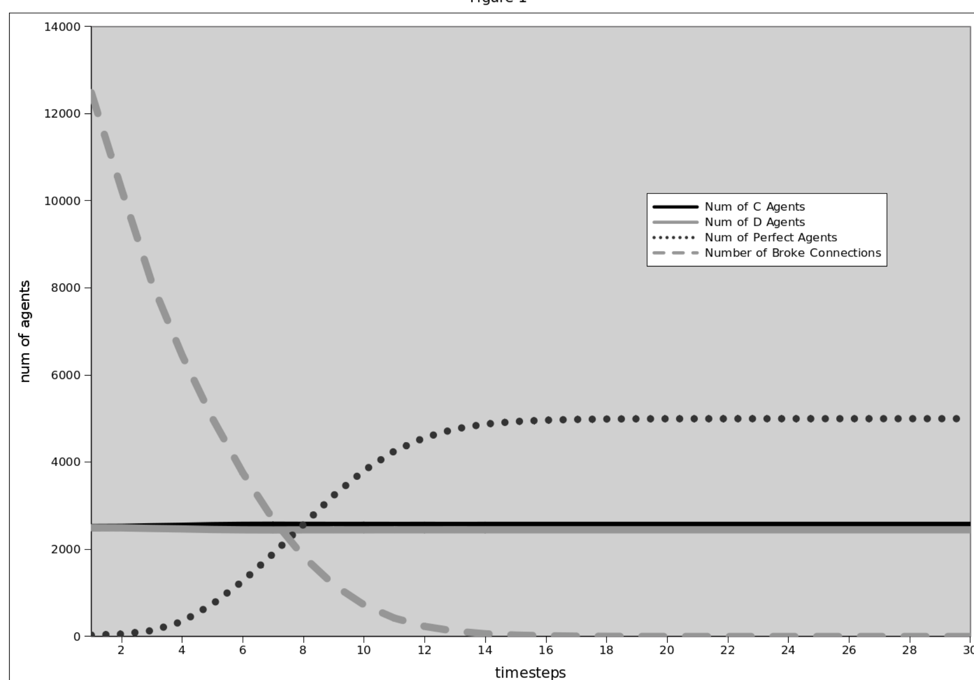


Figure 1: Network Converges into Two Clusters

Figure 1 shows the average results collected from 50 trials. We see that at the end of each trial, all agents have split into two camps and entered into a unique stable clustering state. First, we notice that the lines that represent the numbers of C and D agents almost merge together in the middle of the plot. Since the pay-off matrix for C and D actions is symmetric and there is no difference between these two actions except different symbolic names, the average numbers of C and D agents should be both around 2500, just as the plot shows. Here in order to capture the dynamics of the network, we introduce a new term named perfect agent. An agent will be counted as a perfect agent only when all of his neighbors plays the same actions as he does. In Figure 1, while the numbers of C and D agents remain almost constant, the number of perfect agents rises

steadily from 0 to 5000 throughout the trial. The increasing number of perfect agents shows the network is evolving from a chaotic state towards a stable clustering state. At the same time, we see the number of broken connections is steadily dropping towards 0. As there are more and more agents becoming perfect agents, connections between neighbors are broken less and less frequently.

5.3.2

One-Cluster Stable Clustering State

Now we present the experiment results where the agents all adopt the same action at last and merge into one cluster. In order to show the results more clearly, we have run 50 trials and selected 23 trials where C eventually becomes the dominant action. The experiments are carried out with the following parameters:

Number of Runs: 50
Time Steps: 30
Network Size: 5000
Neighborhood Size: 100
Reward Discount: 0.9
Threshold: 0.9

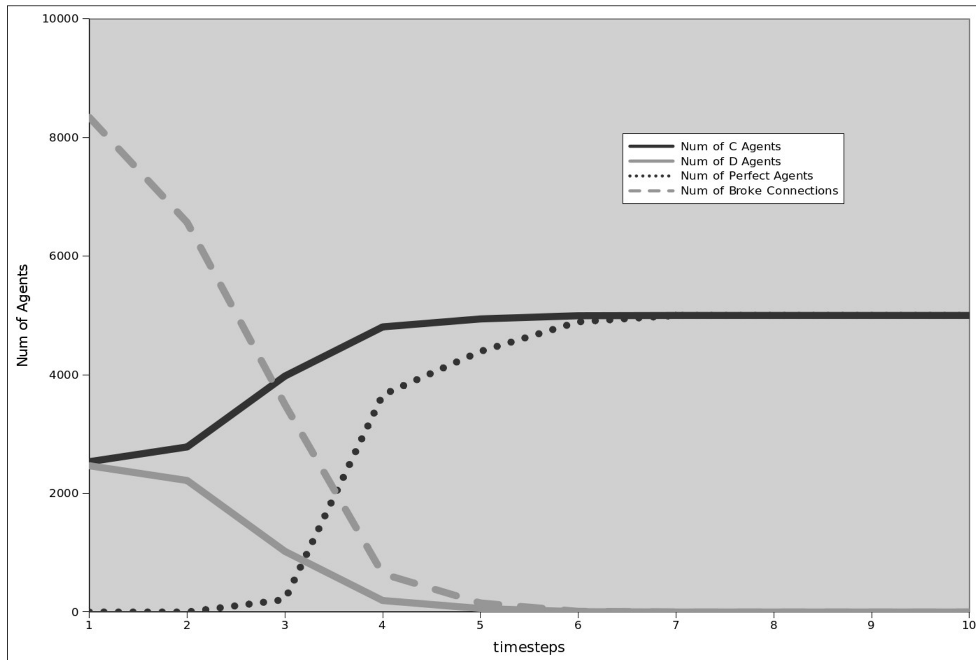


Figure 2: Network Converges into One Single Cluster

In Figure 2, we see number of C agents is steadily rising while number of D agents dropping until the whole network converges into a single cluster. The number of perfect agents and broken connections behaves in a manner similar to that in Figure 1.

5.3.3

Pattern Possibilities with Various Network Sizes

The above experiments demonstrate that the network sometimes converges into one single cluster and sometimes into two clusters. After a series of simulations, we found out that the size of neighborhood has a significant influence over the probability

that a network will converge into one cluster or not. In order to show this, we created 16 networks with different sizes and ran 50 trials with each network. The following plot shows the number of one-cluster trials and two-cluster trials with each network.

Experiment Parameters:

Number of Runs: 50
Time Steps: 30
Network Size: 1000
Reward Discount: 0.9
Threshold: 0.9

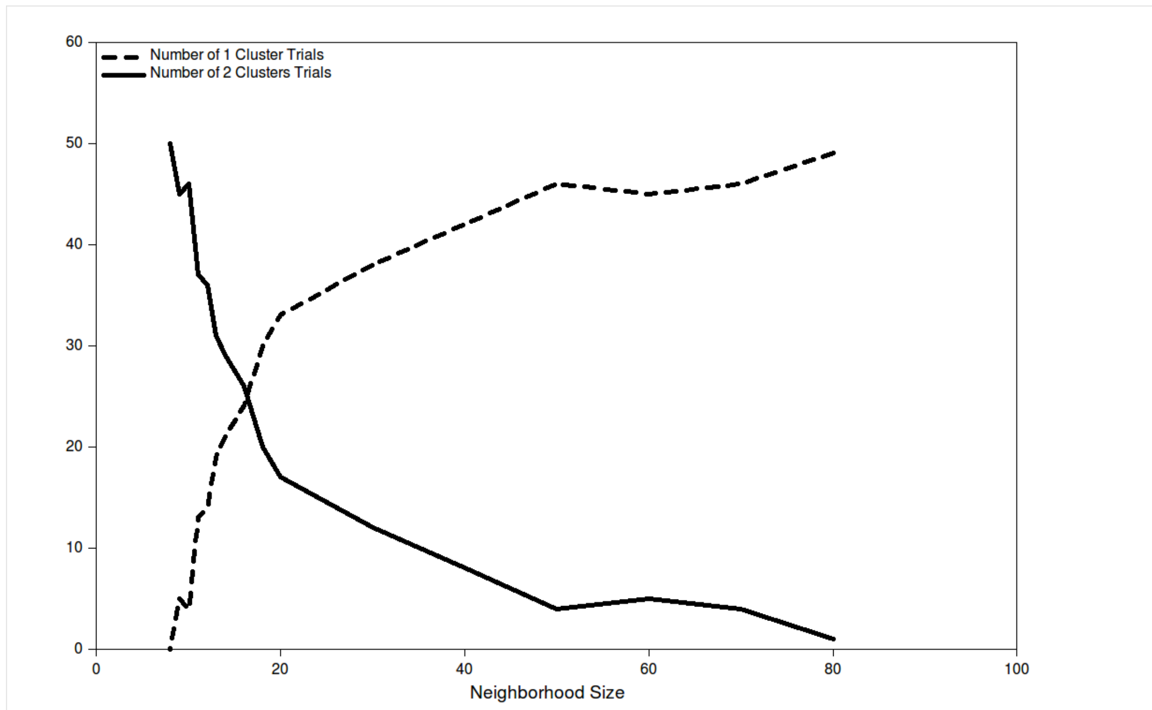


Figure 3: Relationship between the Neighborhood Size and Network Pattern

Figure 3 shows that as the size of the network increases, there is a higher and higher probability that the network will converge into one single cluster. There is a turning point around neighborhood of size 15, when the neighborhood size is approximately 1.5% of the size of the whole network. In such cases, the network can converge into one or two clusters with an approximately even possibility.

This phenomena can be explained in the following way. The size of neighborhood directly decides how much influence the majority action in the network places upon a single agent. The bigger the neighborhood is, the more the agent is influenced by the entire network. Let's try to understand this from two extreme cases. Imagine the experiment is carried out in a complete network, where every single agent is connected to all the other agents in the network. In such a case, the agent's choice of action is affected by the actions of all the other agents, and based on the simple majority rule, all the agents will immediately switch to the dominant action in this turn. On the other hand, if we let the agents keep a neighborhood of size 0, that means no agents has any neighbors, and then each agent will be under no influence from the network as a whole and will just stick to its current action. With similar logic, we see the size of neighborhood really decides the influence each agent receives from the network and also the possibility that they will switch to the dominant action in the network. Therefore, the possibility that the agent will adopt the dominant action in the network and become one single cluster increases with increasing neighborhood size.

5.3.4

Threshold & Broken Connection Peak

As the reader may notice, the preceding trials we used a threshold of 0.9. We choose a relatively low value here in order to make the agents more tolerant of their neighbors. We will discuss tolerance further in the later part of this report. In this case, as long as the weighted average reward from a certain neighbor is higher than 90% of the weighted average reward from all the other agents, the neighbor will be counted as a good one and the connection will be reserved. In addition, once the threshold reaches 1, an interesting phenomenon will happen as the following Figure 4 shows.

Experiment Parameters:

Number of Runs: 50
time steps: 30
network size: 2000
neighborhood size: 10
memory size: 15
reward discount: 0.9
threshold: 1.0

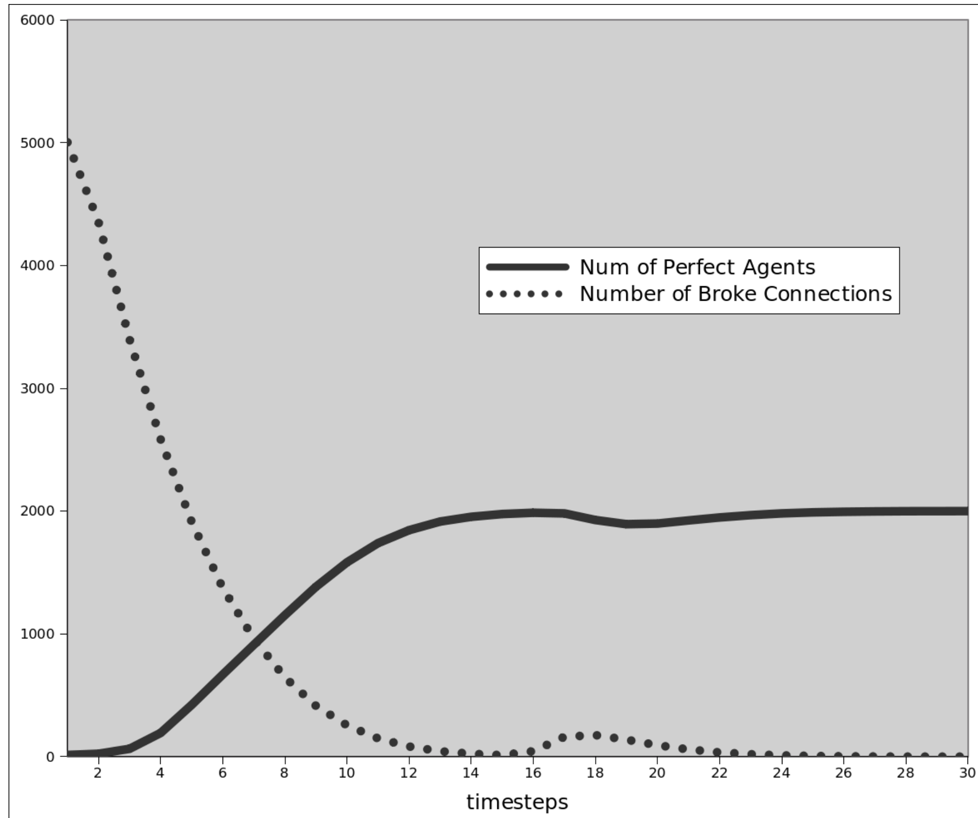


Figure 4: Network with Threshold 1.0, Mem Size 15

As we can see from Figure 4, at time step 17, when many agents have already become perfect agents, there suddenly emerges a small peak in the number of broken connections, and the number of perfect agents also drops correspondingly. This phenomena is caused by the fact that the agents are being too strict about the performance of their neighbors. At time step 16 or 17, since the memory size is only 15, agents begin to "forget about" the bad rewards they received in the very first few turns. As a consequence, the weighted average reward for the agent rises quickly, and becomes higher than the weighted average reward for a certain few neighbors. Therefore, the agent

chooses to break up with the neighbors, even though they have been rewarding in all interactions except the first few turns. By lowering the threshold, we can make the agents more tolerant of their neighbors and thus eliminate the peak. Also, we can change the memory size of the agents to control the time when the peak appears. In order to support our argument, we ran a second trial with the exactly same setup except a threshold of 0.9.

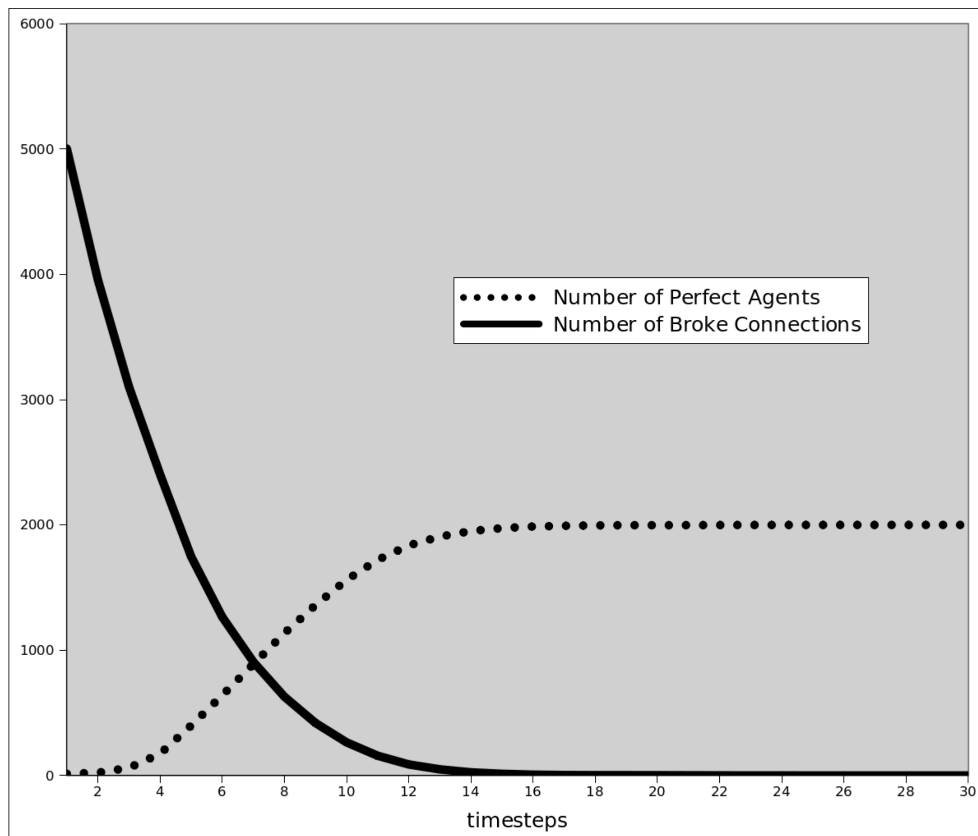


Figure 5: Network with Threshold 0.9, Mem Size 15

From Figure 5, we see patterns similar to ones in Figure 4, except there is no peak. With a threshold of 0.9, the agents are more tolerant of their neighbors. Therefore,

there is almost no connection being broke off once the network enters the stable clustering state. Through this comparison, we see that the existence of the peak is in direct relationship to the value of the threshold. Now we examine the relationship between the peak and the size of memory. The following trail is carried out with memory size of 20 and threshold of 1.0.

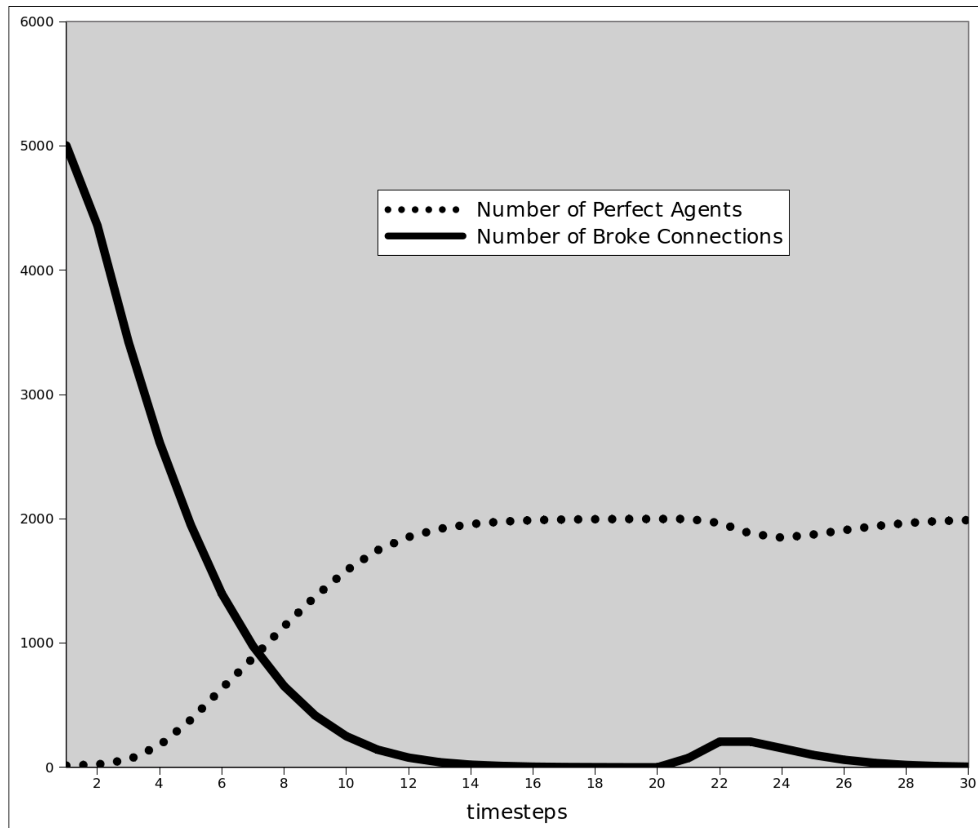


Figure 6: Network with Threshold 1.0, Mem Size 20

As we can see here in Figure 6, when memory size grows from 15 to 20, the time when the peak emerges is also pushed back by about 5 turns. Thus further supports our previous stated theory.

5.3.5

Scalability

In the following experiment, we are going to examine the scalability of the network. We run different trials on various sizes of networks, and in each network, the size of the neighborhood is 0.3% of the size of the whole network.

Experiment Parameters:

number of runs: 50
time steps: 30
reward discount: 0.9
threshold: 0.9

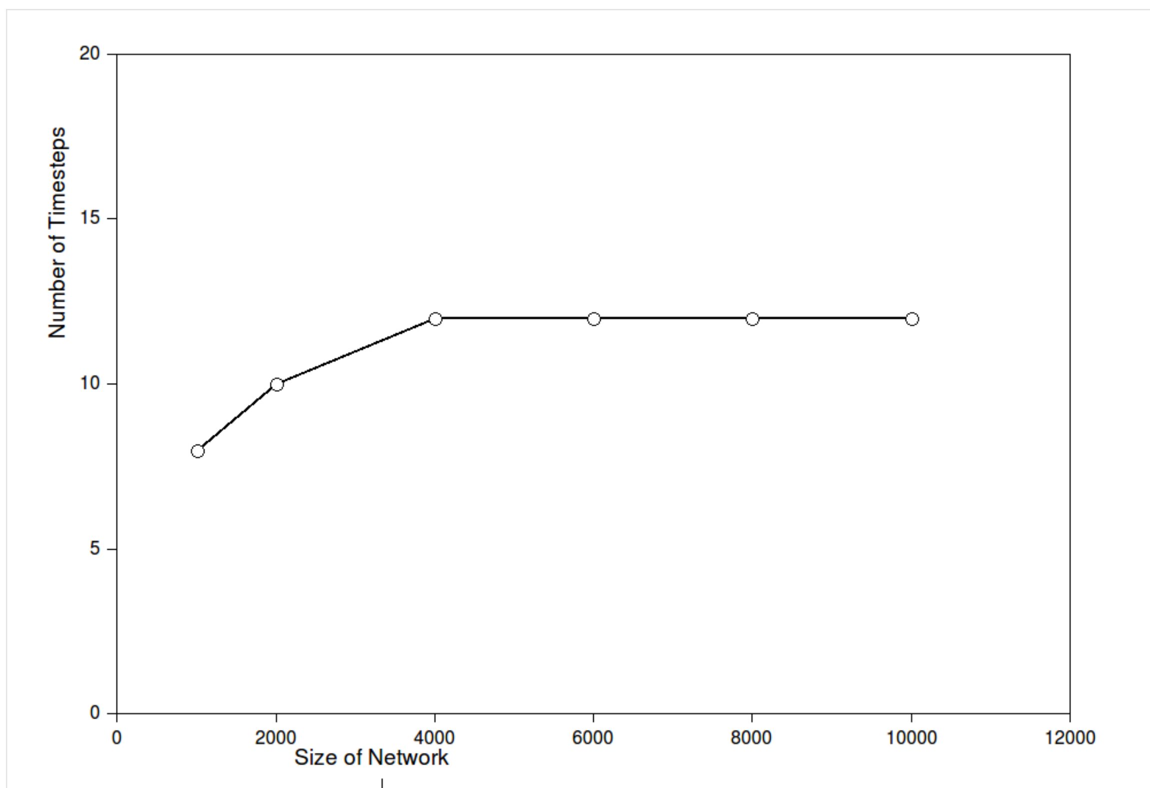


Figure 7: Converging Speed in Various Sizes of Networks

As we can see from Figure 7, small networks generally converge faster than large networks. However, once reach a certain level, the size does not matter much any more and the speed of convergence remains almost as a constant. Though the neighborhood size varies in each trial, the relative size to the network size constantly remains 0.3%. It appears that the relative size of the neighborhood is the key factor that decides the convergence speed of the network.

5.3.6 Comparison with Static Network

At the beginning of this report, we claim that we create this dynamic network in order to model modern dynamic networks that traditional static networks fail to model. In order to show this point more clearly, here we carry out a comparison simulation between dynamic and static networks.

Experiment Parameters:

number of runs: 50
time steps: 30
network size: 5000
neighbor size: 10
reward discount: 0.9
threshold: 0.9

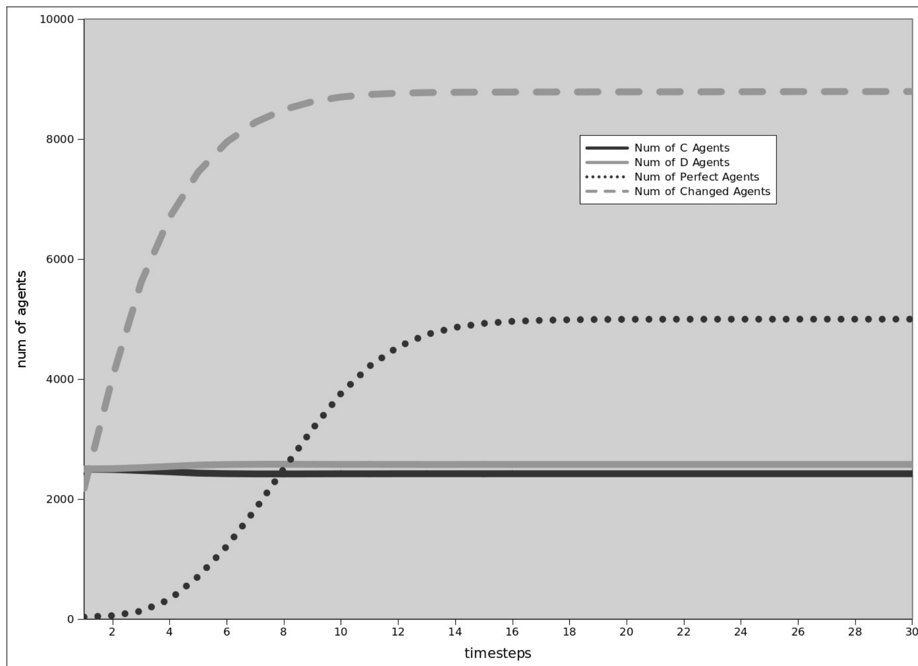


Figure 8: Pattern in Dynamic Network

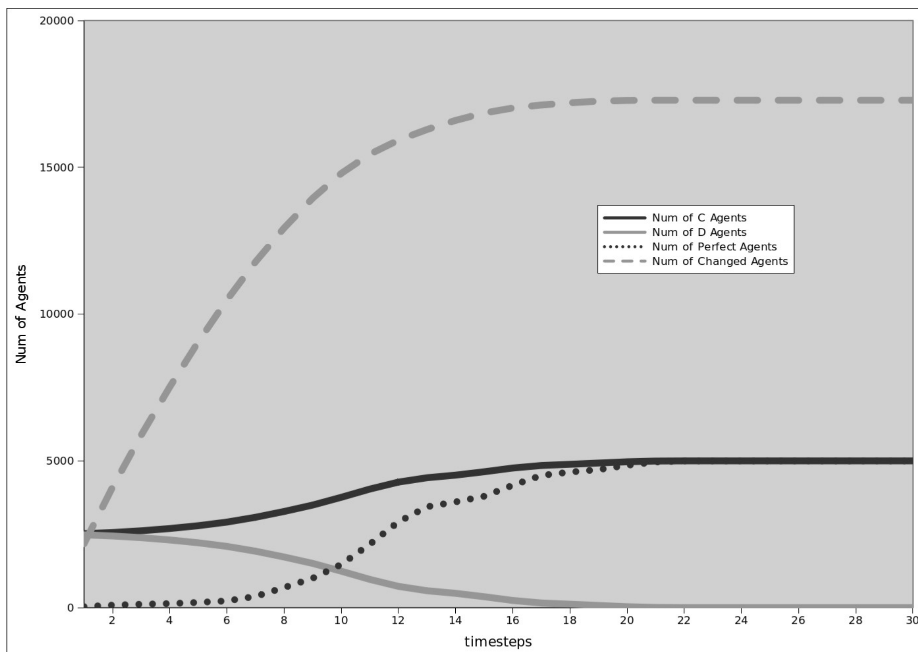


Figure 9: Pattern in Static Network

Figure 8 shows the results from a dynamic network and Figure 9 shows the one form a static network. First, the most obvious difference between these two trials may be the different cumulative number of changed agents. The cumulative number of changed agents measures how many times the agents in the network change their actions. While the number for dynamic network is only about 9000, it is as high as 20000 for the static network. Since in a dynamic network, agents can maximize their rewards not only through changing actions but also through finding coordinating neighbors, agents need to change their actions less frequently than they do in static network, where all of the agents have to switch to one action in order to maximize the rewards received from the whole network. In addition, as we notice in the network, a dynamic network also converges much faster than the static network. It only takes about 12 rounds for a dynamic network to converge while it takes about 20 rounds for static networks. It appears that by changing their neighbors, the agents can maximize their rewards in a much shorter time period.

5.3.7

Tolerance

Since an agent evaluates its neighbors based on their cumulative rewards rather than the reward of one single time step, the agent will occasionally tolerate its neighbor's non-coordinating action because of its good history. We examine this in the following experiments.

Experiment Parameters:

number of runs: 50
time steps: 30
network size: 2000
neighbor size: 20
reward discount: 0.95
threshold: 0.9/0.7

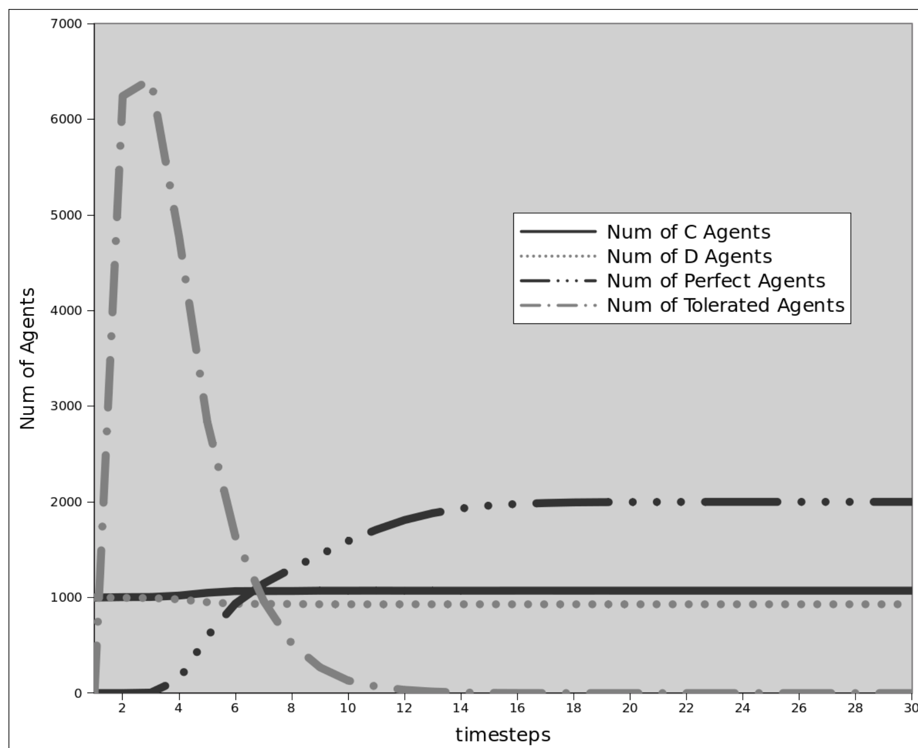


Figure 10: Tolerance Trial with Threshold 0.9

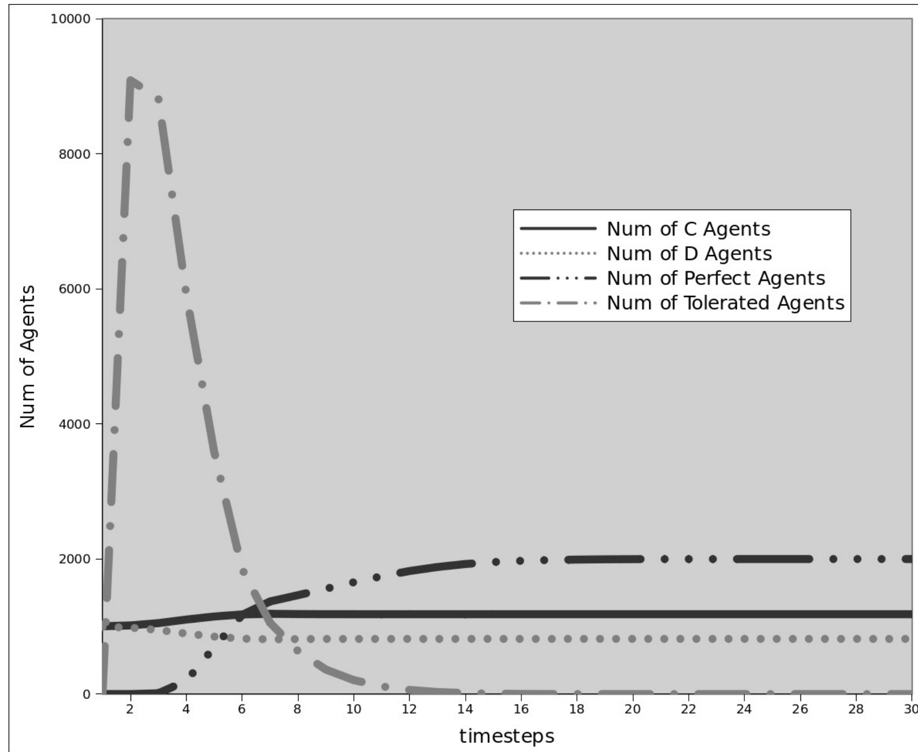


Figure 11: Tolerance Trial with Threshold 0.9

Figure 10 and Figure 11 show two trials with threshold of 0.9 and 0.7 respectively. As we can observe from the plots, in both trials the agents are tolerating some of their neighbors. In addition, when the threshold is low (Threshold 0.7, Tolerated Agents around 9000 at peak), agents are tolerating more neighbors than they do with a high threshold (Threshold 0.9, Tolerated Agents around 6000 at peak).

6

Future work

Although the HWR rule itself is not very complicated and easy to understand, we have observed a wide range of patterns emerging from the experiment results. More work needs to be done in order to fully explore the potential of this dynamic network model. The future work mainly involves the following two directions.

The first direction is to explore the phenomenon of tolerance. As we have seen in the network, the agents adopting the HWR rule occasionally tolerate some bad neighbors. In fact, this phenomenon was expected when we first designed the HRN and HWR rules. The point of letting agents to evaluate their neighbors based on history rather than the most recent action is to allow the agents to keep the good neighbors who once were non-rewarding for very few turns. Though this phenomenon was observed in the network, it did not bring significant change to the network structure and we also have little control over its influence. One main cause for this fact is the simple majority rule. Although the simple majority rule has been widely acknowledged and adopted in various research, the assumption it made is rather unrealistic. Under the simple majority rule, agents should immediately switch their action once they observe a majority of different action. However, this is apparently not true in real world. Human beings turn to be much more stubborn about their action choices. People will not change their belief immediately after

they see a majority of a different belief. It usually takes a long time to change one's habits or belief, if possible at all. Therefore, in order to model the stubbornness of human being, our model should include a new parameter named resistance in the future. The resistance, r, is an integer value bigger or equal to 1. It simply means how many turns an agent can stay in a non-coordinating neighborhood until it changes its action. For example, if $r=3$, then the agent will need to play with a majority of non-coordinate neighborhood three turns in a roll in order to change its action. We believe that by introducing the resistance value, tolerance will play a much more important role in the model.

The second direction is code optimization. During this project, we have drastically improved the performance of the model through a conversion from Java to C++ and some algorithm improvements. At the current stage, there may be little work remaining to be done in order to improve the efficiency of the program. However, so far the program has been programmed for single core only. Nowadays, multicore CPU is the mainstream technology and the number of cores in a CPU is quickly increasing along with technology advancement. If we want to utilize the power of modern CPUs to further improve the performance of the model, we have to bring in parallel computation in our program. It is just a matter of time for us to modify the program into this direction.

Besides the previous mentioned two directions, there are also some other topics we can explore in the future. We can allow the agents in the current model to be added and removed, thus creating a new model dynamic in a different aspect. Then we can compare the patterns of this new model with our existing model and study the differences. We can also let the agents play games other than the Pure Coordination

Game, and study what social norm will emerge with a different game. After we have finished the work in the above two major directions, we can further explore the possibilities of dynamic networks in various ways.

7

Conclusion

The HWR rule enables agents to increase their rewards not only through switching actions but also through updating neighborhoods. As the experiment results show, agents adopting the HWR rule are able to change the network structure to maximize their own interests. Even though the agents behave selfishly, the network still reaches a Pareto-optimum social convention at last.

Reference

Shoham, Y, and M Tennenholtz. “On the emergence of social conventions: Modeling, analysis and simulations.” *AI*, 1997:139-166.

Delgado, Jordi. “Emergence of Social Conventions in Complex Networks.” *Artificial Intelligence*, 2002: 171-175

Borenstein, Elhannan and Ruppin, Eytan, “Enhancing Autonomous Agents Evolution with Learning by Imitation.” *Journal of Artificial Intelligence and Simulation of Behavior*, 1(4):335-348, 2003.

Zimmermann, martin, and Victor Eguiluz. “Cooperation, Social Networks and the Emergence of Leadership in a Prisoners Dilemma with Adaptive Local Interactions.” *Physical Review*, 2005.

Zhang Y and Leezer J, Emergence of Social Norms in Complex Networks, Symposium on Social Computing Applications (SCA09), The 2009 IEEE International Conference on Social Computing (SocialCom-09), Vancouver, Canada, August 29-31, 2009, pp 549-555.

Appendix A

There are some special network which will never reach a stable ending state. Researchers have proved under most circumstances a static network will reach convergence when agents adopt the simple majority rule. However, during our research we found out certain networks that would never converge. The most simple one can be referred as the "traffic light", which is a network containing only two agents. At the beginning, the two agents are connected to each other and they use different actions. Since for each agent the only neighbor it has is the other agent, they both will change their action in the next turn based on the simple majority rule. Apparently, these two agents will keep repeating this process and flipping between two actions. Thus the network can never reach convergence. Even though this is a special case of static network, dynamic networks will have the same problem. If our network is initialized as a complete network (theoretically there is a extremely low probability that this situation will indeed happen), then the network will remain as static since it is impossible to add more connections between any pair of agents (because all agents are connected to each other). Thus, the special case we just pointed out can be treated as an extremely rare but valid case for our model.

Due to the existence of these special cases, not all networks will reach a stable clustering state. Therefore, inspired by Shoham's work, we argue that as time step

approaches infinity, the probability of a network reaching the stable clustering state will also approaches to 1. In the next page, we will show our argument in detail. In order to make our argument easier to understand, here we first explain the basic idea behind our argument. The idea is very similar to the logic of the classic monkey and Shakespeare problem: let a monkey randomly type on a keyboard. Given infinite time, the monkey will eventually finish a work of Shakespeare. The idea is: given a specific process that can happen with a very low probability and infinite time, the probability that this process can happen will approach to 1.

The argument contains three major parts: first, a starting state that should be valid at any time step throughout the trial; second, an ending state, which is also the goal state of the argument; third, a specific process that will lead the system from the starting state to the ending state. In our argument, the starting state is a normal random state of the system. The goal state is the stable clustering state. We have also described the specific process in great detail in our argument.

1 Definition

A_i represents the i'th agent in the network.

a_i represents the action choice for agent A_i. In coordination games, a_i have binary values representing different action choice.

R_{ij} represetns the connection between A_i and A_j.

N_i represents agent A_i's neigbhorhood. For any $A_j \in N_i$, there exsits R_{ij}.

C_i represents agent A_i's coordinating neighbors. $C_i \subseteq N_i$. If $A_j \in C_i$, then $a_j = a_i$.

$\bar{C}_i$ represents agent A_i's non-coordinating neighbors. $\bar{C}_i \subseteq N_i$ and $\bar{C}_i \cap C_i = \emptyset$. If $A_j \in \bar{C}_i$, then $a_j \neq a_i$.

We call agent A_i a majority coordinating neighbor agent if $|C_i| \geq |\bar{C}_i|$; we call agent A_i a majority non-coordinating neighbor agent if $|\bar{C}_i| > |C_i|$.

We call agent A_i a perfect agent if for all $A_j \in N_i$, $a_i = a_j$.

We say a network is in a stable clustering state if for any agent A_i in the network, $|\bar{C}_i| = 0$.

We say a group of agents D form a closed cluster if for any agent $A_i \in D$, $|\bar{C}_i| = 0$, and for any $A_j \in C_i$, $A_j \in D$. Also, for any pair of agent A_i and A_j, $a_i = a_j$.

We say an agent has a free connection when the agent broke one old connection and gains the right to connect to a new neighbor.

We call an agent a free agent when the agent has enough free connections to make it possible to choose its action in the next turn. In other words, the number of free conections the agent has should be bigger than $||\bar{C}_i| - |C_i||$.

We define **whipping** as the following process: every time when a certain free agent looks for new neighbors, it only connects to the agent who uses different action with him. By simple majority rule, the agent will keep flipping between two actions until he breaks off with all his neighbors. During this process, every time when the agent breaks a connection, he will take the right to reconnect only when there are not enough free connections left for him to be a free agent. Otherwise, we will let its neigbhor to have the right to connect new neighbors. Eventually, the agent will become isolated with only one free connection left.

2 Theorem

Theorem Given a pure coordination game, placing no constraints on the initial choices of action by all agents, and assuming that all agents employ the HWR rule, then the following holds:

For every $\epsilon > 0$ there exists a bounded number M such that if the system runs for M iterations then the probability that a stable clustering state will be reached is greater than $1 - \epsilon$.

If a stable clustering state is reached then the all agents are guaranteed to receive optimal payoff from all connections.

3 Proofs

Proof First, we check whether there exists any majority non-coordinating agent in the network.If there does not, then all the agents must be majority coordinating agent. In that case, we let the agent only able to connect to coordinating agents, and the network will reach stable clustering state in less than k steps (k is the memory size of agent).

If there does exist some majority non-coordinating agents, then we need to first form a closed cluster in the network if there doesn't exsit any. We start off by randomly choosing a mojority non-coordinating agent. Then we let it go through the whipping process and become an isolated agent, which is a closed cluster state by itself. We keep selecting other non-coordinaing agents and let them go through similar whipping process until they only have one free connection left. In this way, we can link the original non-coordinating agents to the closed cluster one by one. We carry out the process repeatedly until there are only majority coordinating agents left in the network. Then the network can reach the stable clustering state in less than k turns. The whole process as we discuss above will take up to $g(n)$ turns to finish, and there is a probability $p = 1/f(n)$ that the process will happen.Therefore, if the system runs for $M = x \times g(n) \times f(n)$ iterations then the probability that a stable clustering state will not be reached is at most e^{-x}. Taking $x > -\log(\epsilon)$ yields the desired result.

Appendix B

In the following pages, I attached the codes I used during the experiments.

```
#include <iostream>
#include <map>
#include <list>
#include <cstring>
#include <cstdio>
#include <string>
#include <queue>
#include <cstdlib>
#include <math.h>
#include <sstream>
#include <algorithm>

using namespace std;
using std::string;


// the followings are the major parameters of the experiment, you can change these parameters to adjust
the set-up of the experiment
int NUM_OF_RUNS = 10;
int TOTOAL_NUM_OF_TIMESTEP = 50;
int MEM_SIZE = 30;
int NET_SIZE = 10000;
int NEIGHBOR_SIZE = 100;
float REWARD_DISCOUNT = 0.9;
float THRESHOLD = 0.7;

// the followings are two frequently used constant values. Let m=MEM_SIZE, c=REWARD_DISCOUNT, then C_SUM
= 1+c+c^2+...+c^(m-1), LAST_FAC = c^(m-1)
float C_SUM = (1-pow(REWARD_DISCOUNT, MEM_SIZE))/(1 - REWARD_DISCOUNT);
float LAST_FAC = pow(REWARD_DISCOUNT, MEM_SIZE-1);

class Agent{

    private:
        // use a bit(bool) vector to store the past actions of the agent
        vector<bool> actionHist;

        // save the pointers to three maps, for the purpose of memory-freeing in the future
        pair<const int, Agent*>* np;
        pair<const int, float>* nrp;
        pair<const int, short>* ntp;

        // store the list of neighbors
        map<int, Agent*> neighbors;
        // store the
        map<int, float> neighborsRewards;
        // store the time of interactions the agent has with each neighbor
        map<int, short> neighborTimes;
        queue<float> totNeighborRewHistory;
        // store the list of bad neighbors
        list<Agent*> baddies;
        char action, nextAction;
        float reward, totalReward, avgTotReward;
        int TIMESTEP, numLost;
        int ID;
        // the flag marking whether an agent is a perfect agent or not
        int perfectFlag;


    public:
        Agent(int id){
            np = neighbors.get_allocator().allocate((int)NEIGHBOR_SIZE*1.5);
            nrp = neighborsRewards.get_allocator().allocate((int)NEIGHBOR_SIZE*1.5);
```

```
        ntp = neighborTimes.get_allocator().allocate((int)NEIGHBOR_SIZE*1.5);
        actionHist.reserve(TOTOAL_NUM_OF_TIMESTEP+1);

        reward = 0;
        totalReward = 0;
        avgTotReward = 0;
        TIMESTEP = 0;
        perfectFlag = 0;

        ID = id;

        if(drand48()<.5){
            action = 'c';
            actionHist.push_back(1);
        }
        else{
            action = 'd';
            actionHist.push_back(0);
        }
    }

    ~Agent(){
    }

    // function that frees the agent at last
    void freeMem(){
        neighbors.get_allocator().deallocate(np,(int)NEIGHBOR_SIZE*1.5);
        neighborsRewards.get_allocator().deallocate(nrp,(int)NEIGHBOR_SIZE*1.5);
        neighborTimes.get_allocator().deallocate(ntp,(int)NEIGHBOR_SIZE*1.5);
    }

    // function that chooses the action for agent based on simple majority rule
    int chooseAction(){
        int c1=0, c2=0;
        int neighborSize = neighbors.size();
        int i;
        map<int, Agent*>::iterator it = neighbors.begin();
        for(i=0; i<neighborSize; i++){
            if((it->second->getAction()) == 'c') c1++;
            else c2++;
            advance(it, 1);
        }

        if(c1>c2) nextAction = 'c';
        else if(c2>c1) nextAction = 'd';
        else nextAction = action;

        if(nextAction == 'c')
            actionHist.push_back(1);
        else
            actionHist.push_back(0);

    }

    // update the agent's action
    int updateAction(){
        int tmp;
        if(action==nextAction) tmp=0;
        else tmp=1;

        action = nextAction;

        return tmp;
```

```
        }

        // play games and update related values
        void update(int timestep){
            float tempReward = 0;
            TIMESTEP = timestep;
            reward = 0;

            int neighborSize = neighbors.size();
            int i;
            map<int, Agent*>::iterator it = neighbors.begin();
            for(i=0; i<neighborSize; i++){
                if((it->second->getAction()) == action) tempReward = 1;
                else tempReward = 0;

                int neighTimes = 1 + neighborTimes.find(it->second->getID())->second++;

                float* neighRew = &((neighborsRewards.find(it->second->getID()))->second);

                if(neighTimes > MEM_SIZE){
                    bool neighPastAct = it->second->getPastAct(TIMESTEP - MEM_SIZE);

                    if(neighPastAct == getPastAct(TIMESTEP - MEM_SIZE)){
                        *neighRew = (*neighRew - LAST_FAC)* REWARD_DISCOUNT + tempReward;
                    }
                    else{
                        *neighRew = *neighRew * REWARD_DISCOUNT + tempReward;
                    }
                }
                else{
                    *neighRew = *neighRew * REWARD_DISCOUNT + tempReward;
                }

                reward += tempReward;

                advance(it, 1);
            }

            if((int)reward==neighborSize) perfectFlag = 1;
            else perfectFlag = 0;

            if(neighbors.size()!=0){
                totNeighborRewHistory.push(reward/neighborSize);

                if(TIMESTEP < MEM_SIZE){
                    totalReward = totalReward * REWARD_DISCOUNT + reward/neighborSize;
                }
                else{
                    totalReward = (totalReward - LAST_FAC * totNeighborRewHistory.front())*
                    REWARD_DISCOUNT + reward/neighborSize;
                    totNeighborRewHistory.pop();
                }
            }
            else{
                totNeighborRewHistory.push(0.0);

                if(TIMESTEP < MEM_SIZE){
                    totalReward = totalReward * REWARD_DISCOUNT;
                }
                else{
                    totalReward = (totalReward - LAST_FAC * totNeighborRewHistory.front()) *
                    REWARD_DISCOUNT;
                    totNeighborRewHistory.pop();
```

```
            }
        }
    }

    int checkPerfectFlag(){
        return perfectFlag;
    }

    map<int, Agent*>* getNeighbors(){
        return &neighbors;
    }

    int getNeighborsSize(){
        return neighbors.size();
    }

    list<Agent*>* getBadNeighbors(){
        return &baddies;
    }

    bool hasNeighbor(Agent a){
        return (neighbors.find(a.getID()) == neighbors.end());
    }

    // find non-rewarding neighbors
    int updateBadNeighbors(){

        float avgReward = 0;
        int times = 0;
        int numOfTolerated = 0;

        baddies.clear();

        if(TIMESTEP < MEM_SIZE){
            avgTotReward = totalReward/((1-pow(REWARD_DISCOUNT, TIMESTEP+1))/(1 - REWARD_DISCOUNT));
        }
        else{
            avgTotReward = totalReward/C_SUM;
        }

        int neighborSize = neighbors.size();
        int i;
        map<int, Agent*>::iterator it = neighbors.begin();
        for(i=0; i<neighborSize; i++){
            times = (neighborTimes.find(it->second->getID()))->second;

            if(times > 0){
                if(times < MEM_SIZE){
                    avgReward = neighborsRewards.find(it->second->getID())->second / ((1-
pow(REWARD_DISCOUNT, times))/(1 - REWARD_DISCOUNT));
                }
                else{
                    avgReward = neighborsRewards.find(it->second->getID())->second / C_SUM;
                }

                // the "0.001" logical statements are reserved for float rounding off problems

                if(avgReward/avgTotReward < THRESHOLD || avgTotReward < 0.001 || avgReward < 0.001){
                    baddies.push_back(&(*it->second));
                }
                else if(action!=it->second->getAction()){
                    numOfTolerated++;
                }
```

```
            }

            advance(it, 1);
        }

        numLost = baddies.size();

        return numOfTolerated;
    }

    int getNumNeighborsLost(){
        return numLost;
    }

    void addNeighbor(Agent* a){
        queue<float> q;

        neighbors.insert(pair<int, Agent*>(a->getID(), a));
        neighborsRewards.insert(pair<int, float>(a->getID(), 0.0));
        neighborTimes.insert(pair<int, int>(a->getID(), 0));
    }

    bool removeNeighbor(Agent* a){
        if(neighbors.find(a->getID()) == neighbors.end()) return false;
        neighbors.erase(a->getID());
        neighborsRewards.erase(a->getID());
        neighborTimes.erase(a->getID());
        return true;
    }

    bool getPastAct(int times){
        return actionHist[times];
    }

    char getAction(){
        return action;
    }

    float getReward(){
        return reward;
    }

    int getTimestep(){
        return TIMESTEP;
    }

    int getID(){
        return ID;
    }


    // the following functions are created for debug purpose

    void actionPrintout(){
        int i;
        for(i=0; i<actionHist.size(); i++){
            if(actionHist[i]==1) cout << "c";
            else cout << "d";
        }
    }

    void debugPrintout(){
```

```
        int neighborSize = neighbors.size();
        int i;
        map<int, Agent*>::iterator it = neighbors.begin();

        cout << "\nAgent #" << ID << endl;
        cout << "current action: " << action << endl;
        cout << "total reward: " << totalReward << endl;
        cout << "average reward: " << avgTotReward << endl;
        cout << "action history: ";

        actionPrintout();

        cout << endl;

        it = neighbors.begin();
        for(i=0; i<neighborSize; i++){
            cout << "Neighbor " << it->second->getID() << ":" << endl;
            cout << "\taction: " << it->second->getAction() << endl;
            cout << "\taction history: ";
            it->second->actionPrintout();
            cout << endl;
            cout << "\tRewards: " << neighborsRewards.find(it->second->getID())->second << endl;
            cout << "\tTimes Played: " << neighborTimes.find(it->second->getID())->second << endl;

            int times = (neighborTimes.find(it->second->getID()))->second;
            float avgReward;

            if(times > 0){
                if(times < MEM_SIZE){
                    avgReward = neighborsRewards.find(it->second->getID())->second / ((1-
pow(REWARD_DISCOUNT, times))/(1 - REWARD_DISCOUNT));
                }
                else{
                    avgReward = neighborsRewards.find(it->second->getID())->second / C_SUM;
                }
                cout << "\tNeighbor's average reward: " << avgReward << endl;
            }

            advance(it, 1);
        }

        if(baddies.size()>0){
            list<Agent*>::iterator itt;
            cout << "\nThe following bad neighbors are going to be removed:" << endl;
            for(itt=baddies.begin(); itt!=baddies.end(); itt++){
                cout << "\tAgent " << (*itt)->getID() << endl;
            }
            cout << "@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@" << endl;
        }
    }
};

// break up with the non-rewarding neighbors
int updateNeighbors(Agent* agent, list<Agent*>* baddies, vector<Agent>* pop){
    int count = 0;
    list<Agent*>::iterator it;
    list<Agent*>::iterator end = baddies->end();
    for(it = baddies->begin(); it!=end; it++){
        Agent* baddy = *it;
        agent->removeNeighbor(baddy);
        baddy->getBadNeighbors()->remove(agent);
        baddy->removeNeighbor(agent);
```

```
        if(drand48()<.5){
            count++;
            Agent* choice = &(*pop)[NET_SIZE*drand48()];
            while(choice==agent || choice==baddy || agent->getNeighbors()->find(choice->getID())!=agent-
>getNeighbors()->end()){
                choice = &(*pop)[NET_SIZE*drand48()];
            }
            agent->addNeighbor(choice);
            choice->addNeighbor(agent);
        }
        else{
            count++;
            Agent* bchoice = &(*pop)[NET_SIZE*drand48()];
            while(bchoice==baddy || bchoice==agent || baddy->getNeighbors()->find(bchoice-
>getID())!=baddy->getNeighbors()->end()){
                bchoice = &(*pop)[NET_SIZE*drand48()];
            }
            baddy->addNeighbor(bchoice);
            bchoice->addNeighbor(baddy);
        }
    }

    agent->getBadNeighbors()->clear();

    return count;
}

int main(){

    srand48(time(0));

    int tot_out_num_of_broken_connection[50] = {0};
    int tot_out_num_of_c_agent[50] = {0};
    int tot_out_num_of_d_agent[50] = {0};
    int tot_out_num_of_perfect_agent[50] = {0};
    int tot_out_num_of_changed_agent[50] = {0};
    int tot_out_num_of_tolerated_agent[50] = {0};

    int count = 0;

    int x;

    for(x=0;x<NUM_OF_RUNS;x++){

        // print out in order to show the progress of the experiment
        cout << "." << flush;

        vector<Agent> agents;
        vector<int> agentNames;

        agents.reserve(NET_SIZE+100);
        agentNames.reserve(NET_SIZE+100);

        int i, j;
        for(i=0; i<NET_SIZE; i++){
            Agent a(i);
            agents.push_back(a);
            agentNames.push_back(i);
        }

        int connection_num = 0;

        for(i=1; i<NET_SIZE; i++){
```

```
            for(j=0;j<i;j++){
                double d = drand48();
                if(d < (float)NEIGHBOR_SIZE/(float)NET_SIZE && i!=j){
                    agents[i].addNeighbor(&agents[j]);
                    agents[j].addNeighbor(&agents[i]);
                    connection_num++;
                }
            }
        }

        int timestep;

        int tmp_out_num_of_broken_connection[50] = {0};
        int tmp_out_num_of_c_agent[50] = {0};
        int tmp_out_num_of_d_agent[50] = {0};
        int tmp_out_num_of_perfect_agent[50] = {0};
        int tmp_out_num_of_changed_agent[50] = {0};
        int tmp_out_num_of_tolerated_agent[50] = {0};

        int out_num_of_broken_connection = 0;
        int out_num_of_c_agent = 0;
        int out_num_of_d_agent = 0;
        int out_num_of_perfect_agent = 0;
        int out_num_of_changed_agent = 0;
        int out_num_of_tolerated_agent = 0;

        for(timestep = 0; timestep < TOTOAL_NUM_OF_TIMESTEP; timestep++){

            for(i=0; i<NET_SIZE; i++){
                agents[i].update(timestep);
                if(agents[i].checkPerfectFlag()==1) out_num_of_perfect_agent++;
                if(agents[i].getAction()=='c') out_num_of_c_agent++;
                else out_num_of_d_agent++;
            }

            for(i=0; i<NET_SIZE; i++){
                out_num_of_tolerated_agent += agents[i].updateBadNeighbors();
            }

                agents[0].debugPrintout();
                agents[1].debugPrintout();

            for(i=0; i<NET_SIZE; i++){
                agents[i].chooseAction();
            }

            for(i=0; i<NET_SIZE; i++){
                out_num_of_changed_agent += agents[i].updateAction();
                out_num_of_broken_connection +=
updateNeighbors(&agents[i],agents[i].getBadNeighbors(),&agents);
            }

            // the following print-outs are kept for debugging purpose

//              cout << "TIMESTEP " << timestep << " COMPLETED;" << endl;
//              cout << "\tNUMBER OF C AGENTS: " << out_num_of_c_agent << endl;
//              cout << "\tNUMBER OF D AGENTS: " << out_num_of_d_agent << endl;
//              cout << "\tNUMBER OF PERFECT AGENTS: " << out_num_of_perfect_agent << endl;

            tmp_out_num_of_broken_connection[timestep] += out_num_of_broken_connection;
            tmp_out_num_of_c_agent[timestep] += out_num_of_c_agent;
            tmp_out_num_of_d_agent[timestep] += out_num_of_d_agent;
            tmp_out_num_of_perfect_agent[timestep] += out_num_of_perfect_agent;
```

```
            tmp_out_num_of_changed_agent[timestep] += out_num_of_changed_agent;
            tmp_out_num_of_tolerated_agent[timestep] += out_num_of_tolerated_agent;


            out_num_of_perfect_agent = 0;
            out_num_of_c_agent = 0;
            out_num_of_d_agent = 0;
            out_num_of_broken_connection = 0;
            out_num_of_tolerated_agent = 0;
        }

        // the following if statement decides what results will be filtered out

//      if(tmp_out_num_of_c_agent[TOTOAL_NUM_OF_TIMESTEP-1]!=NET_SIZE &&
tmp_out_num_of_c_agent[TOTOAL_NUM_OF_TIMESTEP-1]!=0){
        if(true){
            for(i=0;i<TOTOAL_NUM_OF_TIMESTEP;i++){
                tot_out_num_of_broken_connection[i] += tmp_out_num_of_broken_connection[i];
                tot_out_num_of_c_agent[i] += tmp_out_num_of_c_agent[i];
                tot_out_num_of_d_agent[i] += tmp_out_num_of_d_agent[i];
                tot_out_num_of_perfect_agent[i] += tmp_out_num_of_perfect_agent[i];
                tot_out_num_of_changed_agent[i] += tmp_out_num_of_changed_agent[i];
                tot_out_num_of_tolerated_agent[i] += tmp_out_num_of_tolerated_agent[i];
            }
        count++;
        }

        for(i=0;i<NET_SIZE;i++){
            agents[i].freeMem();
        }

    }

    cout << "COUNT: " << count << endl;

    for(x=0;x<TOTOAL_NUM_OF_TIMESTEP;x++)

        // print out the output in a table format

        cout << (float)tot_out_num_of_c_agent[x]/(float)count << "\t" <<
        (float)tot_out_num_of_d_agent[x]/(float)count << "\t" <<
        (float)tot_out_num_of_perfect_agent[x]/(float)count << "\t" <<
        (float)tot_out_num_of_changed_agent[x]/(float)count << "\t" <<
        (float)tot_out_num_of_broken_connection[x]/(float)count << "\t" <<
        (float)tot_out_num_of_tolerated_agent[x]/(float)count << endl;

    return 0;
}
```