Practical Results From Artificial Economy With Application to Reinforcement Learning

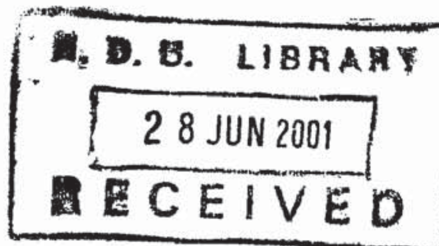
By

AMINE SOUEIDY

A Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Department of Computer Science
Faculty of natural and applied sciences
Notre Dame University – Louaize
Zoukl Mousbeh, Lebanon
June 2001

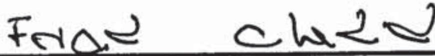

N. D. U. LIBRARY
28 JUN 2001
RECEIVED

Practical Results From Artificial Economy With Application to Reinforcement Learning

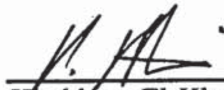
By

AMINE SOUEIDY

Approved by:



Fouad Chedid: Associate Professor of Computer Science and Chairperson.
Advisor.



Khaldoun El-Khaldi: Assistant Professor of Computer Science.
Member of Committee.

Hoda Maalouf: Assistant Professor of Computer Science.
Member of Committee.

Holem Saliba: Assistant Professor of Mathematics.
Member of committee.

Date of thesis defense: Wednesday, 20 June 2001

ACKNOWLEDGMENTS

I would like to thank the many people who have given me their help, advise and encouragement in researching and writing this thesis. I'm particularly indebted to:
My supervisor, Dr Fouad Chedid for his help, elaborate suggestion and encouragement without which this thesis would not be in existence.
I would like to thank Dr Khaldoun Khalidi for his time and support.
I would also like to thank Dr Eric Baum for his time and encouragement.
I would not have been able to carry out this work without the friends who have given me their support and companionship. I'm particularly grateful to Dennis and Yvonne Curyer, and the wonderful Bonny, Patricia Nassif and Walid Mourad.
I also owe a thanks to my family for the support they gave me, namely my mother, Elie, Nadim and Bechir.
Last but not least, I wish to acknowledge here, my father, GOD bless his soul.

ABSTRACT

This thesis discusses the topic of learning in a complex environment. Both the historical basis of the field and a broad selection of the current works are summarized. Reinforcement learning is the problem faced by an agent that learns behavior through trial-and-error interactions with a dynamic environment. The work described here has a resemblance to work in psychology, but differs considerably in the details and in the use of the word "reinforcement". We describe the foundations of a new field of reinforcement learning named artificial economy. We experiment with a prototype based on the artificial economy model to solve the Blocks World problem and show how the artificial economy approach can be integrated with other reinforcement learning techniques.

TABLES OF CONTENTS

LIST OF FIGURES vi

I THE PROBLEM 1

II REINFORCEMENT LEARNING 3

2.1 What is Reinforcement Learning? 3
2.2 Element of Reinforcement Learning 6
2.2.1 The Environment 9
2.2.2 The Reinforcement Function 10
2.2.3 The Value Function 10
2.2.4 Value Iteration 11
2.2.5 The Policy Iteration 11
2.2.6 Goals and Rewards 12
2.2.7 Returns 12
2.2.8 Markov-Decision Process 13
2.3 Dynamic Programming Method 15
2.4 Temporal Difference Learning 18
2.4.1 Predicting Delayed Reward 18
2.4.2 The TD(0) Learning Rule 20
2.4.3 The TD(λ) Learning Method 21
2.4.4 TD and Dynamic Programming..22
2.5 Conclusion 23

III ARTIFICIAL ECONOMY

3.1 Introduction 25
3.2 Artificial Economy Model 26
3.3 Artificial Economy Principles 28
3.4 Conclusion 29

IV DEVELOPING A PROTOTYPE

4.1 The World 31
4.2 Agent Representation 31
4.3 Grab and Drop. 32
4.4 The Bidding and Then Estimation Functions..32
4.5 Changing And Mutating. 34
4.6 Pseudo-Code. 36

4.7 Experiments..37

4.7.1 A Function experiment......37
4.7.2 Average Wealth Property......38
4.7.3 Changing experiment......40
4.7.4 Variance In Average Wealth Property......41
4.7.5 Remove Agents Experiment......42
4.7.6 Estimation Updates......44
4.7.7 Distributed Tax......45
4.7.8 Sub-Agent Experiment......46
4.7.9 Recursive Reward......48
4.8 Conclusion......49

V CONCLUSION......58

REFERENCES......59

APPENDIX A......60

LIST OF FIGURES

Figure

2.1 The Environment	5
2.2 Elements of RL	8
2.3 BW Instance	9
3.1 BW Game	27
4.1 BW Instance	31
4.2 BW Instance (1)	33
4.3 BW Instance (2)	33
4.4 Function Experiment	38
4.5 Average Wealth Property	39
4.6 Changing Experiment	40
4.7 Variance in Average Wealth Property	42
4.8 Remove Agents Experiment	43
4.9 Estimation Updates	45
4.10 Distributed Tax	46
4.11 Sub-Agent Experiment	47
4.12 Recursive Reward	48

CHAPTER I

THE PROBLEM

There are many unsolved problems that computers could solve if the appropriate software existed. For example, a flight control system, a medical software that detects and cures diseases, and sophisticated avionics systems all present difficult, nonlinear control problems. Many of these problems are currently unsolvable, not because current computers are too slow or have too little memory, but simply because it is too difficult to determine what the program should do. Attempts to solve these problems generally use techniques drawn from a specific area of computer science named artificial intelligence or machine intelligence.

The literature of computer science emphasizes two domains or paradigms for creating machine intelligence. First, there is the symbolic processing paradigm that assumes that knowledge can be embedded in the form of if-then-else statements. Some programs that use this technique are known as expert systems. Second, there is the supervised learning paradigm, which doesn't assume that we know as much. We only need to know a set of questions with the right answers. For example, we might not know the best way to program a computer to recognize an infrared picture of a tank, but we do have a large collection of infrared pictures, and we do know whether each picture contains a tank. The computer is expected to look at all the examples with answers and learn on its own how to recognize tanks in general. One area of research that uses supervised learning is artificial neural networks.

Unfortunately, there are many situations where we don't know enough about the world to build an expert system, and we don't even know the correct answers that supervised

learning requires. For example, in a flight control system, the input would be the set of all sensor readings at a given time, and the output would be how the flight control surfaces should move during the next millisecond. Simple artificial neural networks cannot learn to fly the plane unless there is a set of known answers. Therefore, if we don't know how to build a controller in the first place, simple supervised learning would not help. That is why there has been much interest recently in a different approach, mainly reinforcement learning. In this approach, the computer is simply given a goal to achieve. The computer then learns how to achieve that goal by trial and error. The subject of reinforcement learning coupled with a new technique named artificial economy is the main subject of this thesis.

The rest of this thesis is organized as follows: Chapter II gives a review of the major components of reinforcement learning techniques. Chapter III discusses the main features of artificial economy. Chapter IV develops a prototype based on the artificial economy framework to solve the Blocks Word problem. We experiment with a number of parameters and report our findings. Chapter V is the conclusion of the thesis.

CHAPTER II

REINFORCEMENT LEARNING

This chapter is divided into four parts. Part I introduces the main ideas behind reinforcement learning. Part II defines the main components of reinforcement learning Part III describes two basic solution methods: dynamic programming and Monte Carlo methods. Part IV introduces a powerful technique called temporal-difference learning.

2.1 What Is Reinforcement Learning?

Reinforcement Learning, one of the most active research areas in artificial intelligence, is a computational approach to learning whereby an agent tries to maximize the total amount of reward it receives when interacting with a complex, uncertain environment. We know for example, that animals cannot be reasoned with, yet somehow it is possible to train a pet to stop doing a bad behavior, such as chewing on furniture, and get it to perform good tasks such as fetching the newspaper.

The normal technique used by most people to train (program) this living agent is an example of reinforcement learning. The pet receives a treat when it does something good and is punished when it does something bad. It is possible to train computer-based agents using a similar technique of giving "general" rewards and penalties. Furthermore, reinforcement learning makes it possible to train agents in unknown environments where there may be a delay before the effects of actions are understood. i.e. rewards and penalties are not issued right away. For example, an agent playing chess may not realize that it has made a "bad move" until it loses its queen a few turns later.

As another example, imagine that you want to learn how to balance a pole on your hands. In principle you can solve this task in two different ways. You can find a teacher who can tell you how to balance the pole or who allows you to observe him balancing the pole for that you can copy his strategy. This approach is called supervised learning and is usually used to train fuzzy systems or artificial neural networks. You can also try to balance the pole on your own and adapt your strategy if you fail. This is exactly the idea behind reinforcement learning, a special kind of unsupervised learning. Formally reinforcement learning is an approach to machine intelligence that combines dynamic programming and supervised learning to successfully solve problems that neither discipline can address individually. Dynamic programming is a field of mathematics that has traditionally been used to solve optimization and control problems. However traditional dynamic programming is limited in the size and complexity of the problems it can address. Supervised learning is a general method for training a parameterized function approximator, such as artificial neural network, to represent function. However supervised learning requires sample input-output pairs from the function to be learned.

In reinforcement learning, the learner interacts with the environment and senses it in order to gather some information about which action he should select or which task he should perform. If this action is good, the learner will get a reward. It is similar to a certain extent to how babies learn to think. The main difference between an ordinary program and a reinforcement learning one is that in all ordinary programs, the programmer has to think about the algorithm of the program. n other words, he should solve the problem and code the solution into a computer program. In reinforcement

learning however, we expect the program to "think" (by trial and error) and return an algorithm for solving the problem.

Reinforcement learning is a synonym of learning by interaction. During learning, the adaptive system tries some actions (i.e., output values) on its environment, then it is reinforced by receiving a scalar evaluation (the reward) of its actions. Reinforcement learning algorithms selectively retain the outputs that maximize the received reward over time. Reinforcement learning tasks are generally treated in discrete time steps. At each time step, t, the learning system receives some representation of the environment's state s, it takes an action a, and one step later it receives a scalar reward r, and finds itself in a new state s'.

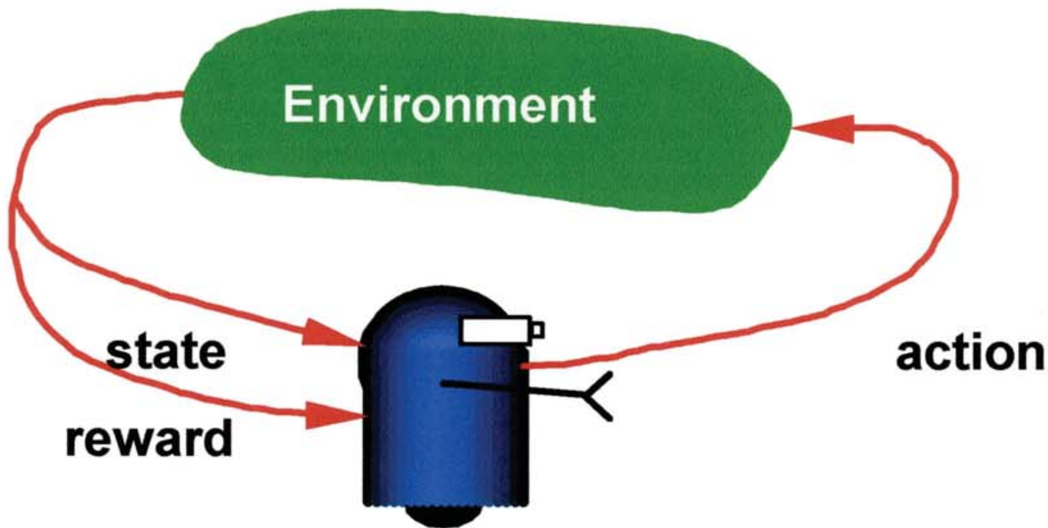


Figure 2.1: The Environment

There are two basic concepts behind Reinforcement Learning are trial and error search and delayed reward [7].

One key aspect of reinforcement learning is the trade-off between exploitation and exploration [10]. To accumulate a lot of rewards, the learning system must select the best experienced actions. In order to do that, it has to keep on trying or experiencing new actions to discover better actions for the future. A central idea of reinforcement learning is a technique called temporal difference (TD) learning [9]. TD methods are general learning algorithms that makes long-term predictions about dynamical systems. They are based on estimating value functions, functions of states V(s) or action-states pairs Q(s,a) that estimate how good it is for the learning system to be in a given state or to take a certain action in a given state. Such value functions guide the action selection mechanism, to achieve a balance between exploration and exploitation, in order to maximize reward over time, and finally let the learning system achieve its goal.

2.2 Elements of Reinforcement Learning

Reinforcement learning problems involve an agent interacting with an environment. The agent must learn about the environment, and must also discover how to act optimally in that environment. Beyond the agent and the environment, one can identify three main sub-elements of a reinforcement learning system: a policy, a reward function and a value function. See Figure 2.2.

A policy defines the le

It is a rule the agent builds and maintains in order to achieve the goal. The policy is the heart of reinforcement learning. Since the agent is not told what to do, its only tool for

achieving the goal is this policy which evolves during the learning process using an extensive search and exploring system.

A reward function defines the goal of the RL problem. The reward function is the immediate reward an agent receives upon executing a certain action. The reward function defines what is good and what is bad for the agent. The main goal of an agent is to maximize the reward function.

Whereas a reward function indicates what is good or bad in an immediate sense, a value function defines what is good or bad in the long run sense. The value function, also called a utility function, calculates the total reward received in the future starting from the current state. For example, a state might always yield a low immediate reward, but may be followed by states that yield high rewards. Rewards are in a sense primary, whereas values, as predictions of rewards, are secondary. Without a reward we cannot calculate the value, and the only purpose of estimating values is to increase the reward. Nevertheless, it is the value function with which we are mostly concerned when making an evaluative decision. Rewards are given directly to the agent from the environment without any time consuming processes. It is very easy to determine the immediate reward. As for the value, the long term reward, is not given by the environment, but instead it must be estimated and re-estimated from the sequences of observations and actions an agent makes throughout all its lifetime executions. Estimation of the long-term value, or the value function is the most difficult part of the RL; it can be very time-consuming depending on the space of the problem.

Although this value function is important, it is not strictly necessary to use it in order to solve a RL problem. There are a lot of search methods that search the space of the

problem by simply trying all the combinations and then come up with the best estimation value. These methods are called evolutionary methods. These methods are applicable only if the space of the problem is very small. This allows the search function to be executed in a reasonable amount of time. Another advantage of the evolutionary method is its independence from the environment. Therefore in some problems where the agent cannot sense the environment, evolutionary method is more applicable.

Policy: What to do.

Reward: What is good.

Value: What is good because it predicts rewards.

Estimation Function: What is the best.

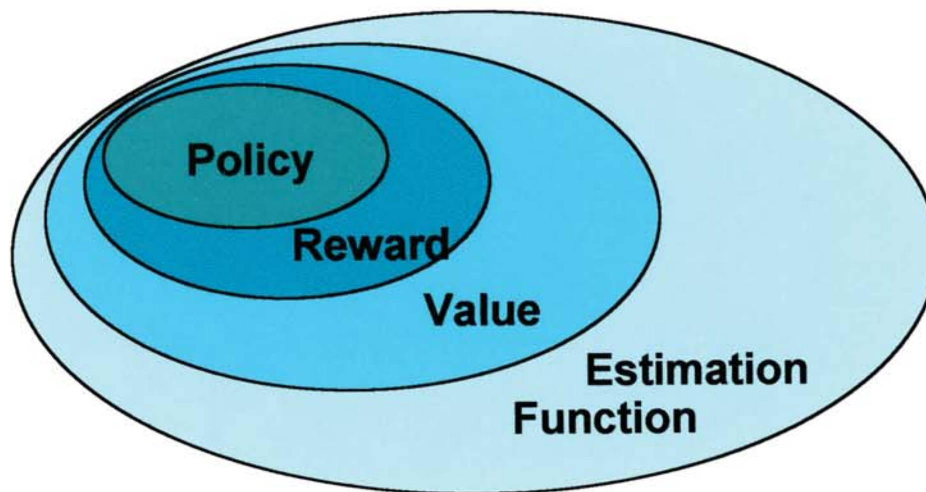


Figure 2.2: Elements of RL

2.2.1 The Environment

In RL the learner interacts with the environment and senses it in order to take action. What is this environment and what are its characteristics? The environment of the reinforcement learning system, is all the states that the system could be in. For example, consider the Blocks World problem (BW is described in detail in chapter IV), the environment here is all the states that this game could represent. See Figure 2.3.

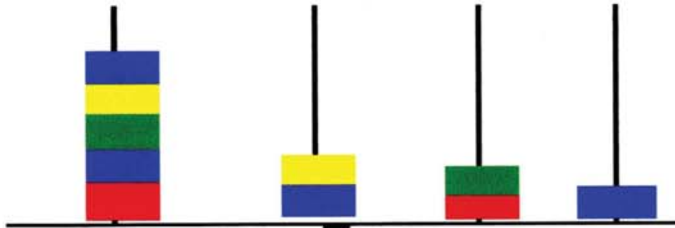


Figure 2.3: Blocks World Instance

Every RL system learns a mapping from situations to actions by trial-and-error interactions with a dynamic environment. This environment must at least be partially observable by the reinforcement learning system. The observations may come in the form of sensor readings, symbolic descriptions, or possibly "mental" situations (e.g., the situation of being lost). The actions may be low level (e.g., voltage to motors), high level (e.g., accept job offer), or even "mental" (e.g., shift in focus of attention). If the reinforcement learning system can observe perfectly all the information in the environment that might influence the choice of action to perform, then the reinforcement learning system can choose actions based on true "states" of the environment. This ideal

case is the best possible basis for reinforcement learning and, in fact, it is a necessary condition for much of the associated theory.

2.2.2 The Reinforcement Function

The Goal of RL system is defined using the concept of reinforcement function, which is the exact function of future reinforcement the agent seeks to maximize. In other words, there exists a mapping from state to state of the environment. In order to map, the agent should take an action, and the reinforcement learning agent will receive reward in form of value. The reinforcement learning agent learns to performs actions that will maximize the sum of rewards when starting from some state, proceeding to the final state.

2.2.3 The Value Function

Although the environment and the reinforcement function are discussed, the issue

reinforcement learning defines two terms. A policy determines which action should be taken or performed in each state, so it is a mapping from state to action.

The value of a state is defined as the sum of rewards received when starting from this state and following some policy to a final state. The best or the optimal policy would be the mapping from state to action that maximizes the sum of rewards when starting in a state and performing actions until the final state is achieved.

$$\text{Value} = \sum (\text{rewards})$$

2.2.4 Value Iteration

Value iteration is an approach for solving MDPs (mathematical dynamic problems) in which the expected values for acting optimally from each state are updated based on the expected optimal values from subsequent states.

This approach falls in the class of iterative relaxation algorithms that take the general form:

$$
\begin{aligned}
V_t(s) &= (1-\eta)V_{t-1}(s) + \eta\hat{V}(s) \\
&= V_{t-1}(s) + \eta(\hat{V}(s) - V_{t-1}(s))
\end{aligned}
$$

where $V_t(s)$ is the value approximation at time t for state s, is a new value estimate for state s at time t+1, and n is an update rate, $0 < n < 1$. Thus, the new value approximation is derived from the previous value approximation combined with an immediate estimate of the value for that state.

2.2.5 The Policy Iteration

Policy iteration is similar to value iteration in that a sequence of value functions is maintained. In value iteration, each value function is the approximation of the expected cost of applying an optimal policy, $V_t(s)$, from state s. In policy iteration, each value function, P, is the approximation of the expected cost of applying a greedy policy from state s based on the previous value function Vt-1(s) These equations can be solved by any algorithm for solving systems of simultaneous equations (e.g., gaussian elimination). The policy iteration is halted when there is no change in the value estimate, for all s.

Unfortunately, these two approaches (value iteration and policy iteration)of RL are not promising is solving ultra complex problems. the value iteration approach

depends on a finite and not dynamic space, whish is problematic when the space is huge. It depends on finding a function that gives reward after each single action. Such function, if it exists, is very difficult to learn.

Also, the policy iteration approach has the space problem because the space of policies will be huge. Thus if the problem is not simple this approach will fail in giving promising results.

2.2.6 Goals and Rewards

As we stated in the previous section, the main goal of the agent is to maximize his rewards. The reward signal is the only way of communicating to the agent what we want to achieve, not how we wanted it achieved. At each time step the reward is a function which returns a number R. The agent tries to maximize the total reward. That means, not only maximizing the immediate rewards, but the overall total reward. We can think of a very simple function which is:

$$R = \frac{\sum \mathrm{Re}\,wards}{nb_of_actions}$$

It is very critical that the rewards function we set indicates what we want to accomplish. The agent might not know what the goal is, but he knows that he has to maximize its rewards, and in doing that, the goal will be achieved. If maximizing the rewards will not

2.2.7 Returns

We said that the agent should maximize his rewards, but what does this mean mathematically? Simply we want to maximize the expected return from selecting an

action. We mean by return the sum of all the rewards received after selecting this action. In the simplest cases, the return is the sum of the rewards:

$$R_t = r_{t+1} + r_{t+2} \qquad\qquad T$$

where T is the final time step.

This makes sense if we know that the problem has a final state. The return in this case will be the sum of rewards from the current state till the final state. If the problem does not have a final state, the reward could reach infinity, therefore, is this case we need something called a discount factor. According to this approach, we try to maximize the discounted rewards. In other words, we decrease the value of the rewards according to their distance from the current state. It means that the rewards, which result from nearer states, are more valuable than those that result from states that are far from the current state.

Let λ be the discount factor, $0<\lambda<1$

$$R_t = r_{t+1} + \lambda^1 r_{t+2} + \lambda^2 r_{t+2} \qquad\qquad \sum_{K=0}^{\infty} \lambda^k r_{t+k+1} \quad 0<k<\text{infinity}$$

Notice that if $\lambda=0$, the agent is myopic concerning maximizing immediate rewards. While if λ approaches 1, the objective takes future rewards into account more strongly and the agent becomes more farsighted.

How to regulate λ is a subject that will not be discussed in this thesis.

2.2.8 Markov-Decision Process

In the previous sections, we mentioned something about the current state of the agent. We also said that the agent could sense the environment. What does that means? How can the agent sense the environment and to which extent?

We define now the current state as whatever information is available to the agent at this state. Sensing means the copying of this information. The information (sensory measurement) could be very simple or very complex depending on the nature of the problem. On the other hand, the state signal should inform the agent about everything in the environment that will be useful in making the decision. For example, if the agent is playing poker, we should not expect it to know what the card in the next deck is. There is hidden information in every environment that the agent cannot get, otherwise there is no

important in solving the problem, but we blame the agent if it knew something and then forgot it. In that case

In order not to forget, we need a complete history of past sensations, and not only the immediate ones. We need to summarize all the sensations compactly, where all the relevant information must be retained. A state signal that succeeds in retaining all relevant information is said to be Markov or have the Markov property.

The Markov property is very important for reinforcement learning because the decisions and the values are assumed to be a function of only the current state.

A reinforcement-learning task that satisfies the Markov property is called finite Markov decision process.

Reinforcement learning is a difficult problem because the learning system may perform an action and not be told whether that action was good or bad. For example, a learning auto-pilot program might be given control of a simulator and be told not to crash. It will have to make many decisions each second and then, after acting on thousands of decisions, the aircraft might crash. What should the system learn from this

experience? Which of its many actions were responsible for the crash? Assigning blame to individual actions is the problem that makes reinforcement learning difficult. Surprisingly, there is a solution to this problem. It is based on a field of mathematics called dynamic programming and it involves just two basic principles. First, if an action causes something bad to happen immediately, such as crashing the plane, then the system learns not to do that action in that situation again. So whatever action the system performed one millisecond before the crash, it will avoid doing in the future. But that principle doesn't help for all the earlier actions which didn't lead to immediate disaster.

The second principle is that if all the actions in a certain situation lead to bad results, then that situation should be avoided. So if the system has experienced a certain combination of altitude and airspeed many different times, whereby trying a different action each time, and all actions led to something bad, then it will learn that the situation itself is bad. This is a powerful principle, because the learning system can now learn without crashing. In the future, any time it chooses an action that leads to this particular situation, it will immediately learn that a particular action is bad without having to wait for the crash.

By using these two principles, a learning system can learn to fly a plane, control a robot, or do many tasks. It can first learn on a simulator, then fine tune on the actual system. This technique is generally referred to as dynamic programming and a slightly closer analysis will reveal how dynamic programming can generate the optimal value function.

2.3 Dynamic Programming Methods

In here, we assume that the environment is a finite MDP. That is, its states and action sets, s and A(s) are finite, and a terminal state exists.

Dynamic programming algorithms are of limited utility in reinforcement learning, both because of their assumption of a perfect environment and because of their computational cost, yet they are theoretically important. Searching for good policies is the key of the dynamic programming. Using a value function, dynamic programming tries to create the best policies in the system. Assume V^{π} the state-value function for an arbitrary policy π. Computing V^{π} is called policy evaluation in the dynamic programming literature.

$$V^{\pi}(s) = E_{\pi}\{ r_{t+1} + \lambda^{1} r_{t+2} + \lambda^{2} r_{t+2} \qquad _{t} = s \}$$

This means that at state (s) using the policy π, the value function is the immediate reward plus the reward of this state, plus the reward of the following state, multiplied by a discount factor, plus the subsequent one, and so on till the final state. But note that

$$E_{\pi}\{ \lambda^{1} r_{t+2} + \lambda^{2} r_{t+2} \qquad _{t} = s \} = V^{\pi}(s+1)$$

then we can write:

$$V^{\pi}(s) = E_{\pi}\{ r_{t+1} + \lambda V^{\pi}(s+1) \mid s_t = s \}$$

That means, the value-function of the current state (s) is equal to the immediate reward plus the value-

know which policy is used with what state and action, we introduced the term $\pi(s, a)$, which is the probability of taking action a in state s under policy π.

$$V^{\pi}(s) = \Sigma\pi(s, a) \,\Sigma\, P^{a} [Ra + \lambda V^{\pi}(s')]$$

The Iterative policy evaluation algorithm could be used to compute the evaluation function, as long as the problem is FMDP. One way to compute the successive approximations V(k+1) from V (k), is the iterative policy. Evaluation which applies the same operation to each state s, and the expected immediate rewards, along all the one step-transition possible under the policy being evaluated. *This kind of operation is called full back up. There are several kinds of back up, depending on the problem, but in dynamic programming there is always a full back up. This means that every value function used is backed up in a table.

In programming or coding the iterative policy evaluation algorithm, one can use two arrays; one for the old values and another for the new values. One method does not require changing the old values while the other method of coding depends upon creating only one array and immediately updating the old value by the new one. Since not all the problem converges, because it depends on the nature of the problem, the algorithm should stop and detect that this is a non-converging problem. Iterative policy relaxation algorithms converge only at the limit, but they should stop if no convergence exists. One way for detecting this case is the difference between V (k+1) and V (k).

If $V(k+1) \quad V(k) < \epsilon$ then the algorithm halts, and we call the task an undiscounted episodic task.

Why are we concerned with computing the value function? Because we want to improve the policies. The value-function declares how good it is to follow a specific policy from a defined state, but it tells nothing about the effectiveness of the policy. Would it be better or worse to change the new policy? In order to answer this question let us assume the following:

Consider selecting action (a) in S and thereafter follow the existing policy π, let us call it V^{π} and compare it to V^{π}. If $V^{\pi} < V^{\pi}$, then the policy π is not the optimal one. So far we have seen how, given a policy and its value function, we can easily evaluate a change in the policy at a single state and to a particular action. It is a natural extension to consider changes at all states and at all possible actions.

2.4 Temporal Difference Learning

Sutton and Barto [8] extended the concept of reinforcement comparison to provide a solution to the temporal credit assignment problem in delayed reinforcement learning. The resulting algorithms are called temporal difference (TD) learning methods.

2.4.1 Predicting Delayed Reward

Before describing TD learning procedures it is necessary to outline some of the terminology that applies to sequential decision tasks. It is also important to consider exactly what function of future rewards the learning system should seek to maximize. After dealing with these preliminaries, the basic TD predictor is outlined and then generalized to give a family of TD learning methods.

Sequential decision tasks: In most delayed reinforcement tasks, there is a dynamic interaction between the environment and the learning system. The future reinforcement and the future context are dependent both on the past contexts and on the actions of the system. In sequential decision tasks of this nature, the concept of state is useful in describing the condition of the environment at any given time. A state description contains sufficient information such that, when combined with knowledge of future actions, all aspects of future environmental states can be determined. In other words,

given a state description, an appropriate action can be selected without previous knowledge about the history of the past states. This is the path independence property of the state description also known as the Markov property described in previous sections.

The set of all possible states for a task is called the state space. The transition function for a state space S is the set of probabilities P (i,j $\in$ S), giving the likelihood that any one state will follow another. A state space together with a transition.

In general, for any given state, difference transition probabilities will be associated with difference actions available in that state. A function that maps states to preferred actions (or action probabilities) is called a policy (described in previous sections). A second function that maps states to the expected future rewards for a given policy is denoted V^{π} and called the evaluation function(also described in previous sections). A common assumption in delayed reinforcement learning is that the task constitutes a Markov decision process in which the set of possible contexts corresponds to the state space for the task. The behavior acquired by the agent constitutes the policy function, and predictions are summarized by the evaluation function.

Time horizons and discounting: In tasks with delayed rewards, actions must be selected to maximize the reinforcement received at future points in time where the intervening delay can be of arbitrary length. The extent of how any sequence of actions is optimal can therefore be determined only with respect to some prior assumption about the relative values of immediate and long-term rewards. Such assumptions are made concrete by the concept of a time horizon. For example, a system that has an infinite time horizon values all rewards equally no matter how far they lie ahead, whereas a system with a finite time horizon values rewards only up to some fixed delay limit. To allow lower values to be

attached, it is usual to use an infinite discounted time horizon where the value of each future reward is given by an exponentially decreasing function of the length of the delay period. In TD learning the slope of discounted time horizon is specified by a constant denoted by λ ($0< \lambda <1$) and known as a discount factor. Assuming a sequence t =

-steps the total discounted R(t), at a given time t, is then obtained by summing the discounted future rewards

$$R(t) = r(t + 1) + \lambda\, r(t + 2) + \lambda^2\, r(t+3) + \lambda^3\, r(t+4) + K$$

which can be written as

$$\sum_{k=1}^{\infty} \lambda^{k-1}\, r(t + k)$$

The goal is to learn to anticipate the expected value of this return, meaning that the prediction V(t) should be estimated from the above formula.

2.4.2 The TD(0) learning rule

One way of estimating the expected return would be the average values of the truncated return after n-steps

$$\sum_{k=1}^{n} \lambda^{k-1}\, r(t + k)$$

In other words, the system could wait n steps, calculate this sum of discounted rewards, and then use it as target value for computing a gradient descent error term. However, for any value of n, there will be an error in this estimate equal to the prospective rewards (for as yet inexperienced time-step)

$$\sum_{k=n+1}^{\infty} \lambda^{k-1}\, r(t + k)$$

The central idea of TD learning is to notice that this error can be reduced by making use of the predicted return V(t+n) associated with the context input at time t+n. Combining the truncated return with this prediction gives an estimate called the corrected n-step return $R_n(t)$.

$$R_n(t) = \sum_{k=1}^{n} \lambda^{k-1} r(t+k) + \lambda^{n} V(t+n)$$

The estimator used in the TD method which Sutton[8] calls TD(0) is the one-step corrected return

$$R_1(t) = r(t+1) + \lambda V(t+1)$$

which leads to a gradient descent error term called the TD error.

The TD(0) learning rule carries the expectation of reward one interval back in time, thus allowing for the backward chaining of secondary reinforcement. For example, consider a task in which the learning system experiences, over repeated trials, the same sequence of context input (with no reward attached) followed by a fixed reward signal. On the first trial, the system will learn that the final pattern predicts the primary reinforcement. On the second trial, it will learn that the penultimate pattern predicts the secondary reinforcement associated with the final pattern. In general, on the Kth trial, the context that has seen k steps before reward will start to predict the primary reinforcement.

2.4.3 TD (λ) Learning Method

The TD(0) learning rule will eventually carry the expectation of a reward signal back along a chain of stimuli of arbitrary length. The question that arises, however, is whether it is possible to propagate the expectation at a faster rate. Sutton[8] suggested that this can be achieved by using the TD error to update the prediction associated with a

sequence of past contexts where the update size for each context is weighed according to its recent occurrence. A learning rule that incorporates this heuristic is

$$V(t) = \beta\, e(t+1) \sum_{k=0}^{r-1} \lambda^{r-k}\ \varphi(t - k)$$

where λ ($0<\lambda<1$) is a decay parameter that causes an exponential fall-off in the update size as the time interval between context and reward lengthens.

One of the advantages of this rule is that the sum of the right hand side can be computed recursively using activity trace vector ϕ

$$\phi \qquad \phi(t) + \lambda\phi \quad -1)$$

where ϕ λ) update rule

$$V(t) = \beta\, e(t+1)\ \phi$$

2.4.4 TD and Dynamic Programming

A second way to understand TD learning is in relation to the DP methods for determining optimal control actions. Dynamic programming is a search method for finding a suitable policy to a MDP. A policy is optimal if the action chosen in every state maximizes the expected return. Computing this optimal control requires accurate models of both the transition function and the reward function. Given these prerequisites dynamic programming proceeds through an iterative exhaustive search to calculate the maximum expected return, or optimal evaluation, for each state. Once this optimal evaluation function is known, an optimal policy can be easily found by selecting in each state the action that leads to the highest expected return in the next state.

A significant disadvantage of DP is that it requires accurate models of the transition and reward functions. TD algorithm can be considered as incremental forms of DP that require no advanced or explicit knowledge of state transitions or of the distribution of available rewards. Instead, the learning system uses its ongoing experience as a substitute for accurate models of these functions.

Analysis of DP leads to a learning method called Q learning that arises directly from viewing the TD procedure as incremental dynamic programming. In Q learning a prediction is associated with each of the different action available in a given state. While exploring the state space the system improves the prediction for each state-action pair using a gradient learning rule. This learning method does away with the need for explicitly learning the policy. The preferred action in any state is simply the one with the highest associated value. Therefore, as the system improves its predictions it also adapts its policy. If each action in each state is attempted a sufficient number of times then Q learning will eventually converge to an optimal set of evaluations.

2.5 Conclusion

Reinforcement learning methods provide powerful mechanisms for learning in circumstances of truly minimal feedback from the environment. The research reviewed here shows that these learning systems can be viewed as climbing the gradient in the expected reinforcement to a locally maximal position. The use of a secondary system that predicts the expected future reward encourages successful learning because it gives better feedback about the direction of this uphill gradient.

The rest of this thesis is organized as follows: Chapter III introduces a new sub-field of reinforcement learning called artificial economy. It differs from reinforcement learning in

its multi-agent nature, and the bidding function it uses. Chapter IV provides some experimental results based on the artificial economy model that solve the Blocks World problem. Chapter V concludes with new ideas to enhance the artificial economy results.

CHAPTER III

ARTIFICIAL ECONOMY

Although using reinforcement learning techniques was successful in many domains, this is not the case in Blocks World problem, because BW has a huge space, and the use of linear evaluation function is not possible in the domain of BW. What to do in such a case? Eric Baum answered this question and introduced a sub-field of reinforcement learning called artificial economy (AE)[1,2,3,4]. This Chapter discusses the main features of artificial economy, and introduces the BW game as an experimental field.

3.1 Introduction

Artificial economy addresses the problem of learning in an ultra complex environment. Since all reinforcement learning techniques failed in the ultra complex environment, Baum thought of a new technique that could manage this huge space. The idea is to assign credit for individuals who gathered to solve one problem, much like the work in our brain, where million of agents collaborate to make a decision. In doing that, Baum thinks that an agent can evolve to be rational, i.e intelligent, and this leads us to create an intelligent program.

Since we speak about program intelligence, let us make the following definition

larasa); the

program will be able to evolve into a state in which it discovers the rules of the game on its own. Discovering the rule of the problem should also mean that the program would be

In other words, the program builds up the algorithm of the problem and not the programmer, otherwise the program would be stupid by doing what the programmer wants it to do. But how could a computer program solve a problem and give an algorithm?

3.2 Artificial Economy Model

The Blocks World problem (figure 4.1) creates a challenge to reinforcement learning techniques because of its huge space. The Blocks World space grows exponentially with the number of blocks. TD learning, the main value iteration approach, succeeded in solving about 8 block problems, but not larger ones. Koza[6] applied genetic programming to the far simpler problem of solving a single instance of blocks world; however, it only succeeded in solving a single 9 block problem. To solve this game using reinforcement learning techniques, Baum created his artificial economy program [1,2,3,4,6] and he called it hayek. Hayek consists of a collection of agents, where each agent has some initial wealth, and can sense the environment and take actions accordingly.

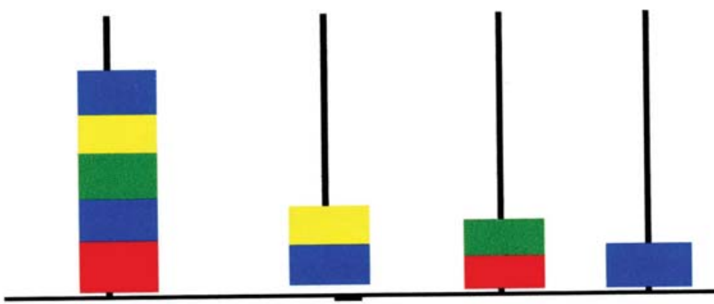


Figure 3.1: BW Game

An instance of the Blocks World contains 4 stacks of colored blocks, with 2n blocks and k colors. The leftmost stack (stack 0) serves as the goal stack and is of height n. The other three stacks contain, between them, the same set of n colored blocks as stack 0. The user may pick the top block of any stack but the goal and may drop that block on any stack but the goal. The objective of the game is to copy the zero stack in the first (or target) stack.

The system works in a series of auctions. All the agents which are allowed to bid raise their bids, and the one with the highest bid wins the auction and the whole world, and move it from one state to another. If this agent, which won the auction, enhanced the world, it receives rewards, and the next agent which wins the next auction will pay this agent these rewards. Agents, who do not interact at all after X iterations, are deleted from the world. Agents whose wealth is ten times more than the average wealth will have the right to create children and give them wealth, and these children will pay ten percent of their profit plus a small constant sum to the creator. Each agent also pays taxes proportional to the number of instruction it executes.

If we want to translate this work in computer like language, we say the following: the program would have a class called creator, which creates agents with a wealth equal constant, and for each agent it creates a set of strings and rules. The string simulates a

state of the world, and the rule simulates the action from one state to another. All these creations will be random; thus all agents created will have the same chances. The programmer defines the number of strings and rules. The program begins by creating an instance of the problem, thus creating an instance of the B W problem. Then it starts creating agents, and each agent which has at least one string that matches the world state can fire one of its rules and own the world. If more than one agent has matched the world, the agent with the highest wealth will fire. This loop continues until the problem is totally solved; all the agents which interact are saved with their rules, and another instance of the problem will be presented. We repeat the same process until the software is able to solve all or approximately all the instances. The set of rules of the agents that interact in all the instances is the algorithm of the problem.

3.3 Artificial Economy Principles

The two most significant principles of artificial economy are property rights and conservation of money.

Property rights means that all the agents have the same chance, and the whole world is owned by auctioning to one and only one agent. The agent has the right to refuse to sell by outbidding other agents.

Conservation of money insures that there is no money leaking from the system, and the agents can earn money only if they increase the payment to the system from the world. To do so the agents should be rational; indeed they are, because all irrational agents are deleted from the system.

These principles are even true in life. Imagine an economy where each individual tries to increase his own profit disregarding whether this profit affects the world or not. This is

hing of the world.

Similarly in the system, if each agent wants to execute its rule knowing that this rule will harm the system, the software will not succeed because the active rule will be the rule

fit, so the software knows how

It is known that many multi-agent systems where property rights were not

t

solved no more than 10 blocks, whereas with these properties, Hayek was able to solve more than 200 blocks.

3.4 Conclusion

Artificial Economy is a new form of reinforcement learning. It proves that the human intelligence can be mimicked into computer programs. This intelligence can be obtained as a result of the collaboration among multiple agents that corporate to solve a given problem. The next chapter summarizes some experimental results based on the artificial economy model that solves the Blocks World problem.

CHAPTER IV

DEVELOPING A PROTOTYPE

In this chapter, a prototype based on the artificial economy framework was developed to solve the B W problem. Center to our implementation is a data structure which describes objects of the type agent. In our implementation, an agent is an instance of the following class:

```
#ifndef AGENTIncluded
#define AGENTIncluded

const int m = 5; // number of rules or strings per agent
const int n = 4; // number of blocks in the goal stack

class BlockWorld;


// Class agent represente the agent configuration
class AGENT
{
    friend BlockWorld;

    private:

    long int SubAgent;     // hold the id of the creator (if exists)
    long int id;           // sequential ID of the agent
    int exp1 [m][3];       // first expression of the left side ot the rule
    int operation[m];      // could be = or !=
    int exp2 [m][3];       // second expression of the left side of the rule
    int rule  [m][2];      // grab and drop, the right side of the rule
    double estimation [m];         // the updated value function
    int frequency[m];              // frequency  of the rule used
    double wealth;                 // agent wealth
    static double AverageWealth;   // total average wealth of all the agents
    static long int oid;           // object ID
    BlockWorld* world;             // The Class world reference

public:

    AGENT(BlockWorld* theWorld);
    AGENT(const AGENT&);
    int NumCorrect ();          // NumCorrect function
    int A();                    // calculate the A value of A * NumCorrect + B
    double UpdateB();           // update the B value of A * NumCorrect + B
    void B(double value);       // calculate the B value of A * NumCorrect + B
    void view ();                         // view the rule srtucture
    void changerules();                   // changing and mutation function
//    void copytofile(int);
    const AGENT& operator = (const AGENT& a);
};

#endif
```

4.1 The Blocks World Problem

The BW world consists of four stacks (See Figure 4.1). The goal stack, stack 1, stack 2, and stack 3. The aim of the agent is to construct stack1 with the same configuration as the goal stack. Only one iteration is allowed at a time; a grab or a drop. A grab means grabbing one block from a specific stack, while a drop means dropping the grabbed block in another specific stack.

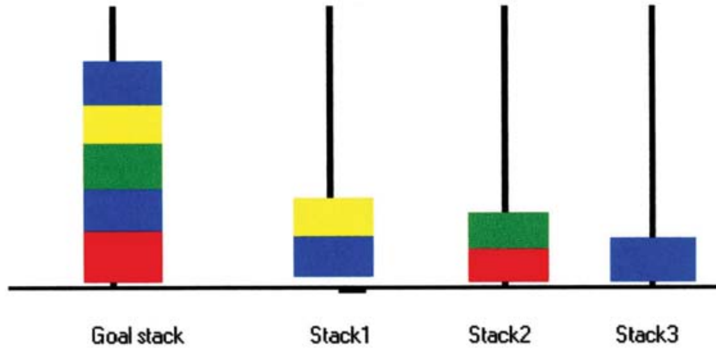


Figure 4.1: The World

4.2 Agent Representation

Each agent represents a configuration of the world. The general form of a configuration is:

If exp1 operation exp2 then grab[stack x],drop[stack y], where x different from y.

Exp1 (Exp2) is a two dimensional array, the first dimension of which represents the rule number, and the second dimension is a string of the following form:

[stack number, block number, type]

where

esents

be matched. Notice that exp1 and exp2 can have a wild char representation; for example,

4.3 Grab and Drop

For example, (grab[stack1],drop[stack3]) means, grab the top block from stack one, if not empty, and drop it on stack3. This drag-drop operation is represented in the line of code.

rule[m][2]

where m is the number of rule per agent. The agent constructor, randomly assigns values for rule[m][0] and rule[m][1]. These random values are the stack number which could vary from one to three. In order to eliminate the occurrence of unused rule like (grab[1],drop[1]), we make sure that rule[m][0] is always different from rule[m][1].

4.4 The Bidding and The Estimation Functions

The estimation function is used by the agent to estimate the value of its rules. It has the following representation.

$$A * NumCorrect + B$$

where $A = \sum_{i=1}^{n} (i * \alpha)$

where α is a Boolean value which is equal to 1 if the block on the first stack at position x matches the block on the goal stack at position x; it is equal to zero otherwise

Consider for example, the instance shown in Figure 4.2.

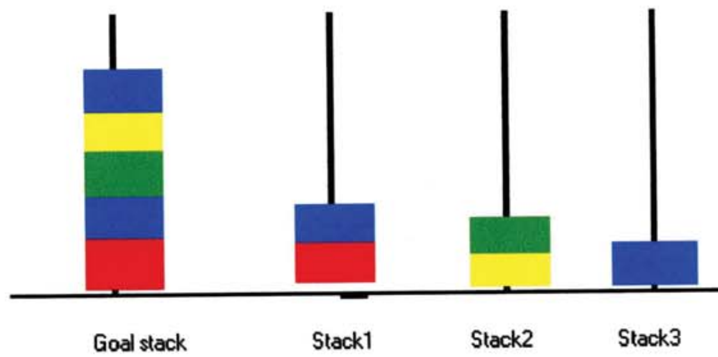


Figure 4.2: BW instance (1)

In this case

NumCorrect = 2
A = 5+4 = 9
B is initially equal to zero.
A*Numcorrect + B = 18.

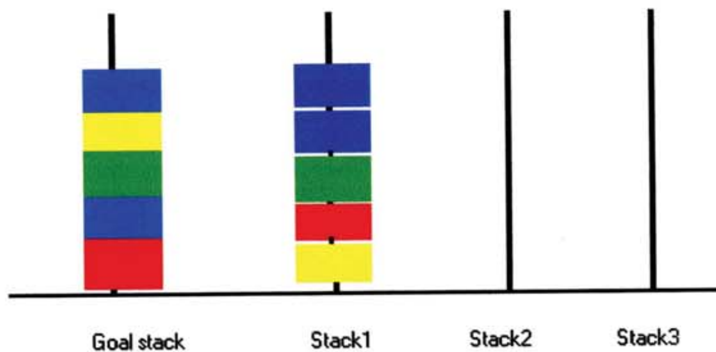


Figure 4.3: BW instance (2)

Now consider the instance representation in Figure 4.3. In this case

NumCorrect = 2

A = 3+1 = 4

B is initially equal to zero.

A*Numcorrect + B = 8.

Although both representations have the same number of matched blocks, the first representation is better. Eventually, it is better to have matched blocks at the lower level; that is why our estimation function gives more value for matched blocks in the lower levels. B is updated every time the agent makes a bid. If the agent receives rewards, then it updates B to increase this reward in the next bidding process, otherwise it decreases the value of B, depending on its bidding value.

Once the rule has been estimated and selected among all the other rules, the bidding function uses the estimated value to calculate the bidding value.

4.5 Changing And Mutating

Each agent has the right to change or mutate its rules. A wealthy agent (set in our implementation wealth > 10*average wealth) can make a sub-agent that is a modification of itself.

To modify a rule, repeat with probability P=0.25 the following:

With P = 0.3 Replace an old rule with a new rule

With P = 0.3 Reshuffle rule

With P = 0.3 exchange rule left side

With P = 0.1 mutate an old rule

TO mutation of a rule do the following:

Left side

With P = 1/3 delete a symbol

With P = 1/3 insert a symbol

With P = 1/3 replace a symbol

Right side

With P = 0.25 delete a symbol

With P = 0.25 insert a symbol

With P = 0.25 replace a symbol

With P = 0.25 reshuffle rules

In our prototype the changing function is described as follows:

Back to the exp1 representation (See Section 4.2), let us assume that we have the following rule:

If (stack1,2,ONE) = (stack3,1,ONE) then (grab[stack1], drop[stack2])

(stack1,2,ONE) = (stack3,1,ONE) is the left side of the rule, while

(grab[stack1], drop[stack2])

is the right side. A mutation rule on the left side could be

changing symbol 2 from exp1, or changing the assignment operator.

4.6 Pseudo-Code

Let n be the number of blocks used in the game, and m be the number of rule per agent.

k is the number of different block colors.

[0] Fix n ,m and k

[1] Generate an instance of the blocks world problem of n blocks and k color.

[3] Let t be the
configuration of the world.
[4] If t = 0 then generate a new agent with m number of rules. Set agent initial wealth to 0 else if n = 1 and that agent wealth = 0 (first time winning) then set it wealth to 1 else if n > 1 then let each agent, selected in step[3], bid for the world using the bidding function described id sec 4.3. If any of these agents is a first time auctioning, then set its bid to the highest bid in the auction plus ϶ and set his wealth to equal that bid value. (this is to encourage new agents)
[5] Select the highest bidder, and make it pay its bid to the previous owner (if exists). If that previous owner used to be a sub-agent, it must pay a percentage P of its profit(if there is a profit) to its creator . This agent will become the new owner of the world. Update the estimation function.
[6] The new owner executes its action on the world, transforming it to a new state
[7] Update the average total wealth
[8] If the new state is the goal configuration, then the agent collects all rewards and declares the game as over. Go to step [1] to solve another instance of the game.
[9] The winner agent pays a tax proportional to that action it has made. This tax will be distributed evenly to all the living agents.
[10] For all living agents do

if agent wealth exceeds some upper bound W, that agent can generate additional sub-
own wealth. these sub-agents will be made to pay some percentage P of their profit to their creator. Also, at any time a sub-agent wealth falls below some lower bound L, it is removed from the system and its wealth is returned to its creator
nd L, that agent is removed

from the system.
[12] For all living agents, change and mutate rules (See sec 4.4)

from the system and its wealth if it has any, will be leaked from the system.
[14] Go to step[2]

4.7 Experiments

This section summarizes a set of experiment done on the Blocks World problem. We experiment with a number of parameters and report our findings

4.7.1 Function experiment

We define the function NumCorrect as Follows.

NumCorrect = the number of correct blocks on the first stack

We define the estimation function as follows:

$$E = \frac{A * NumCorrect + B}{MaxVal}$$

where $A = \sum (n * \alpha)$ $1 < n < \text{total number of blocks}$ (on the zero stack)

n is the height of the block in the first stack, starting from the bottom of the stack

α is a Boolean which is equal to 1 if the block on the first stack at position x matches the block on the zero stack at position x, and zero elsewhere.

$B = (\beta - \delta) * (n / \varphi)$

β is the number of correct blocks on the first stack

δ is the total number of blocks on the first stack

φ is a constant which we vary during the experiment.

Note that $B <= 0$.

MaxVal is the maximum value that A * NumCorrect + B can reach, and it is used to normalize the estimation function

$$MaxVal = \frac{A * NumCorrect}{\sum(n*1)*n + \frac{n2}{\varphi}}$$

Since at the maximum level α always =1 and (β - δ) = n

Note that in this experiment the system was trained to solve 20 instances simultaneously. It took almost 48 hours

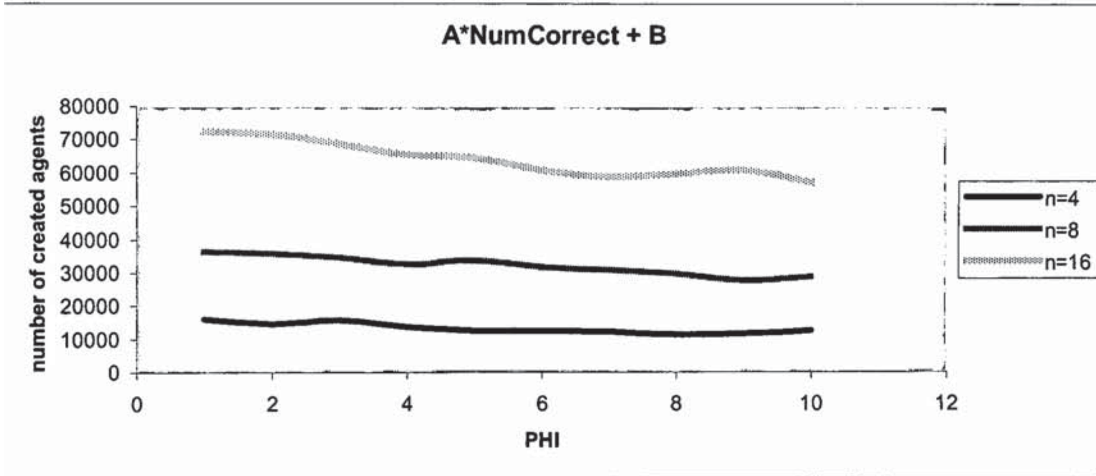


Figure 4.4: Function Experiment

This experiment shows that the value of A is more important than B, which means that the information of matching blocks on the first stack is much more helpful than the information about the correct blocks which are not on the first stack.

4.7.2 Average Wealth Property

We use the same architecture as above, but we compute B as follows.

B = the value which will be added to the A*numCorrect in order to make a break even.

Break-even Formula for calculating B.

Assume that the estimation function (A* NumCorrect + B) returns a value X, and the reward returned by the following agent is Y.

B will be updated to be.

If $Y > X$, then B is still the same. (originally B =0) Else,

$$B = Y \quad \frac{A * NumCorrect + B}{MaxVal}$$

We define β as the average wealth of the agents.

If $(\beta * 0.5) < W < (\beta * 0.8)$ we mutate (See Sec 4.4) with probability X

If $W < (\beta * 0.5)$ we change (See Sec 4.4) totally the rule with probability X

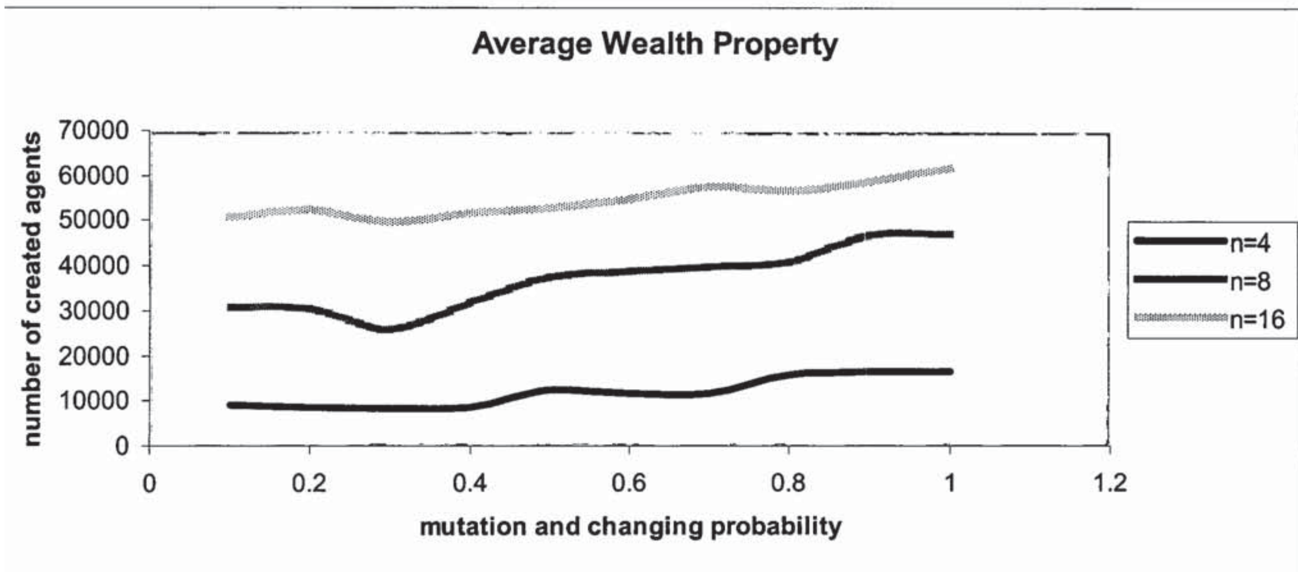


Figure 4.5: Average Wealth Property

This experiment shows that fixing the changing and mutation probability by 30% yield the best result. Notice that this is the probability of changing after the condition on

the average wealth is achieved. For example, all agents with wealth below 0.5 * average wealth, will not mutate or change their rule. Note that a mutation is included in the changing process (See Sec 4.4)

4.7.3 Changing experiment

In this experiment, we exclude the use of average wealth. All living agents can change and mutate equally. There is an average wealth as a condition for changing function executes.

a set of good rules, we should be able to change and update the randomly created rule by the system. The aim of the changing function and the mutation function is the exploration process. Without changing or mutation, there is no exploration. In order to study the exploration process, we study the behavior of the system by varying the changing rate from zero to 100 percent, and we report the result in Figure 4.6

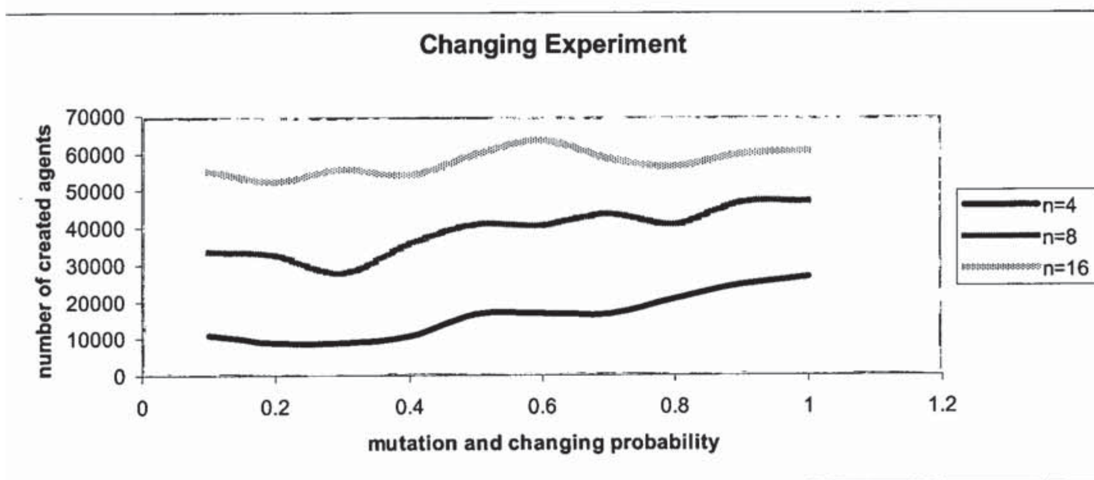


Figure 4.6: Changing experiment

Notice that the previous experiment is better in terms of the number of created agents. They are both good at 30%, but keeping the average wealth property is better. This means that limiting the exploring process to a certain agent is better. Since the use of average wealth in the previous experiment was random, we want to study the behavior of the system when we vary the average wealth percentage, as a condition for the changing and mutating rule.

4.7.4 Variance In Average Wealth Property

We use the same experiment as in experiment 4.6.2 but we vary the percentage of being far from the average wealth and we fix the changing function probability to 30%.

Assume W is the average wealth.

If agent wealth <= average wealth* P use changing rules with probability 30%

we are studying the exploration in relation to the average wealth. How does the average wealth affect the exploration? Should we have a condition for exploring? The previous experiment shows that conditional changing (not all the agent can change their rule) is better, but the condition of the average wealth was fixed. How is this exploration technique affected in the average wealth? This experiment reports results concerning the above questions.

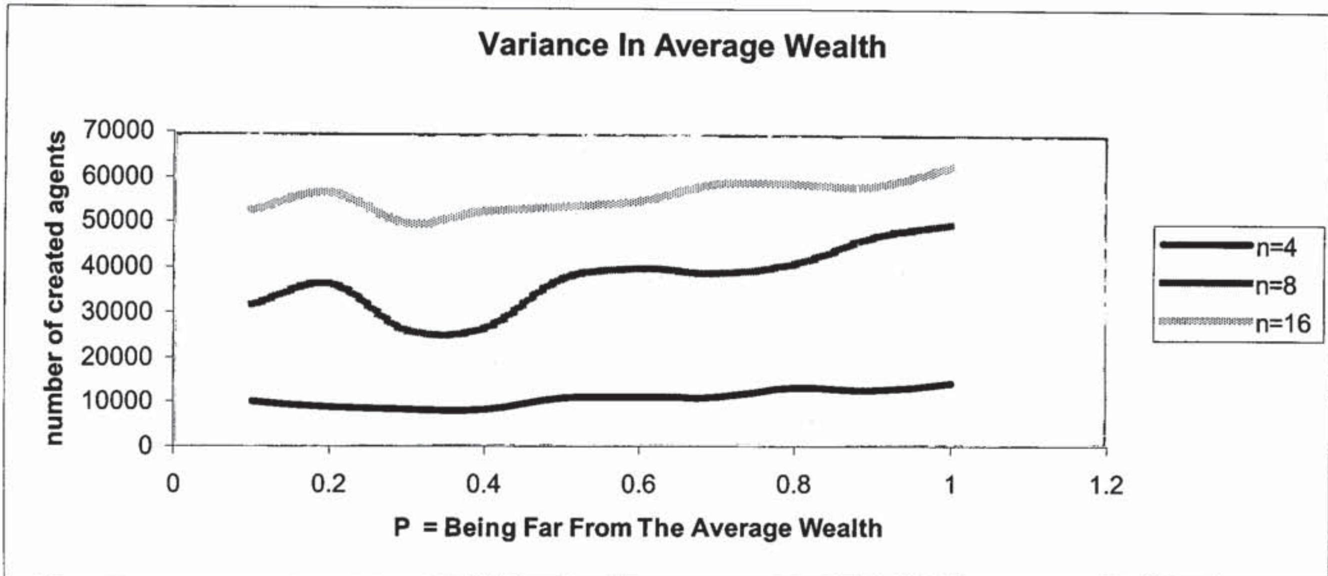


Figure 4.7: Variance In Average Wealth Property

Notice that fixing the average wealth percentage to 30% is the best configuration. This means that only agents with wealth far by 30% than the average wealth explore, we will have a better result in terms of number of created agent. Comparing the average wealth to the agent wealth is a hint about how much that agent was successful compared to the other agents.

4.7.5 Remove Agents Experiment

We stated in our pseudo-code that if an agent wealth decreases below some lower bound L, that agent is removed from the system. We remove these lazy agents because

eir money. This means that there is

lose any computational time on them. But, what is that lower bound L?

An agent will be removed from the system if its wealth decreases bellow X

We vary X between 0.1 and 0.0001 $0.1 < X < 0.0001$

We fix the percentage of being far from the average wealth to 30%

We also fix the percentage of changing (mutation is a sub -field in the changing function) to 30 %

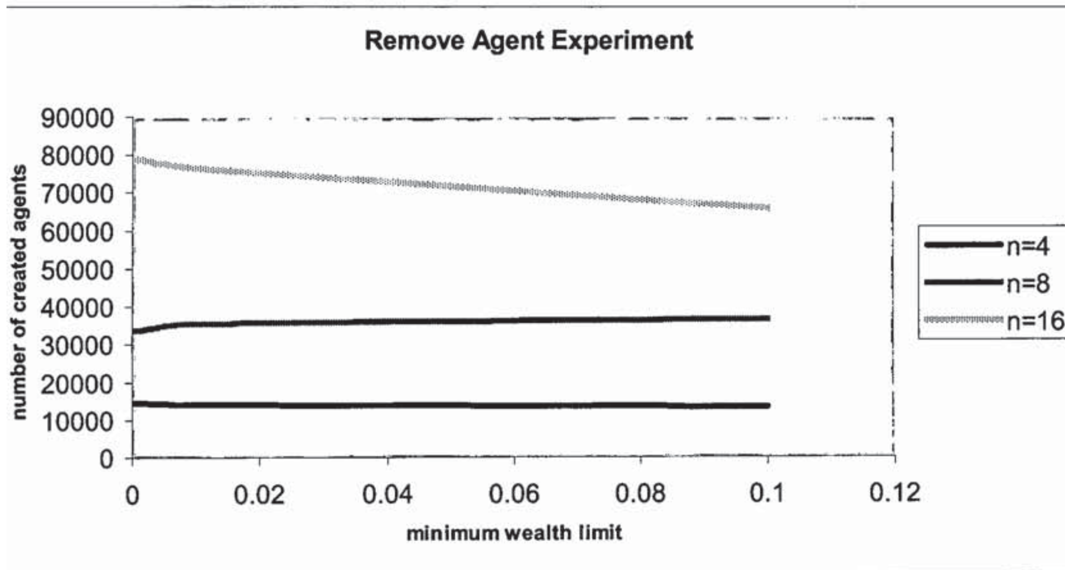


Figure 4.8: Remove Agents Experiment

It is normal that the number of created agents will decrease, since we are missing the opportunity of letting some agents contribute. Note that the slight increase in n=8, is due to the random process. It is important to mention that the result for $0<X<0.05$ is constant. And this is true for n=4,8 and 16.

Removing an agent from the system is very important for the time spent and the cost of the calculation process, and is vital for convergence.

4.7.6 Estimation Updates

we define the estimation function as follows:

$$E = \frac{A * NumCorrect + B}{MaxVal}$$

We want to update the estimation function B by adding X, to reinforce learning. This

tribution

was bad, and has moved the world to a worst state, it must receive a negative reward, and elsewhere it must receive a reward. The value X in the function below is the element of punishment and reward. If the action is good, the agent will increase his estimation function by X, otherwise it will decrease it by X. In out prototype, this update was kept in the agent class above by estimation [m].

$$E = \frac{A * NumCorrect + B + X}{MaxVal}$$

X = (Bid Return) * ϵ

we vary ϵ from 0.01 to 0.1 0.01 < ϶ < 0.1

We fix the percentage of being far from the average wealth to 30%

We also fix the percentage of the changing function

An agent will be removed from the game if his wealth decreases bellow 0.1

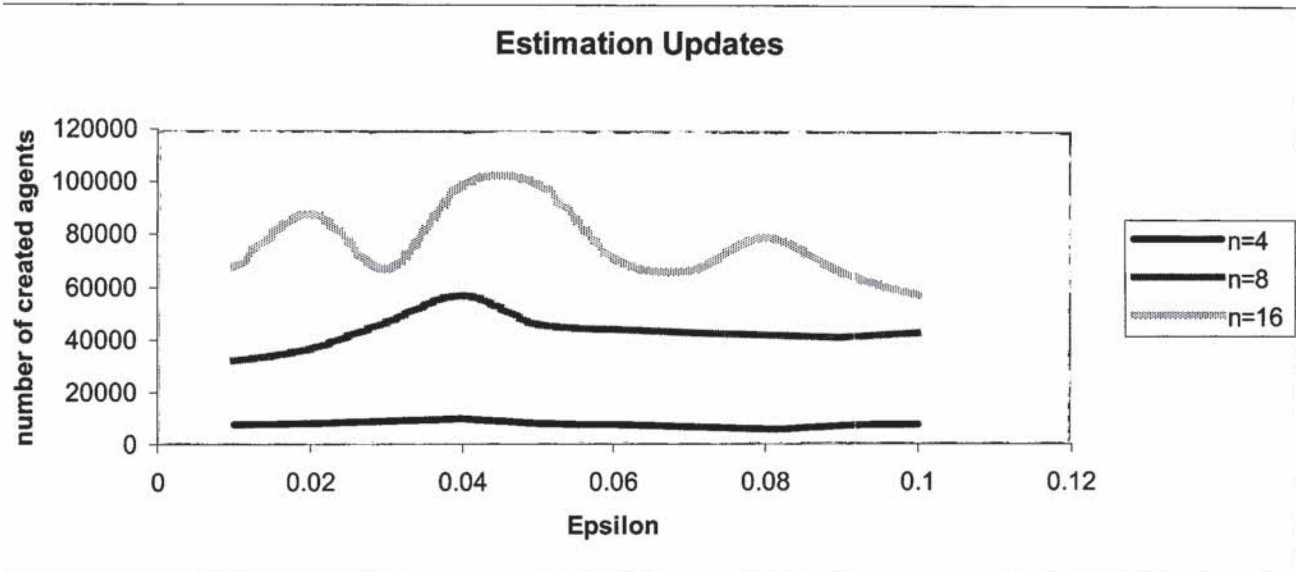


Figure 4.9: Estimation Updates

Notice that when ϶ = 0.8, n=4 and n=8, are optimal, but this is not true for n=16 where it is optimal for ϶ = 0.1. This is due to the random process used in our prototype.

4.7.7 Distributed Tax

Taxes are very important in real economy, but are they important in artificial economy? We experiment the effect of taxes in our prototype. Each agent who contributes has to pay taxes to the system. These taxes will be distributed evenly to all agents. The tax is related to the bidding amount and to the wealth of the agent. In our prototype taxes are computed as a percentage of the bidding value.

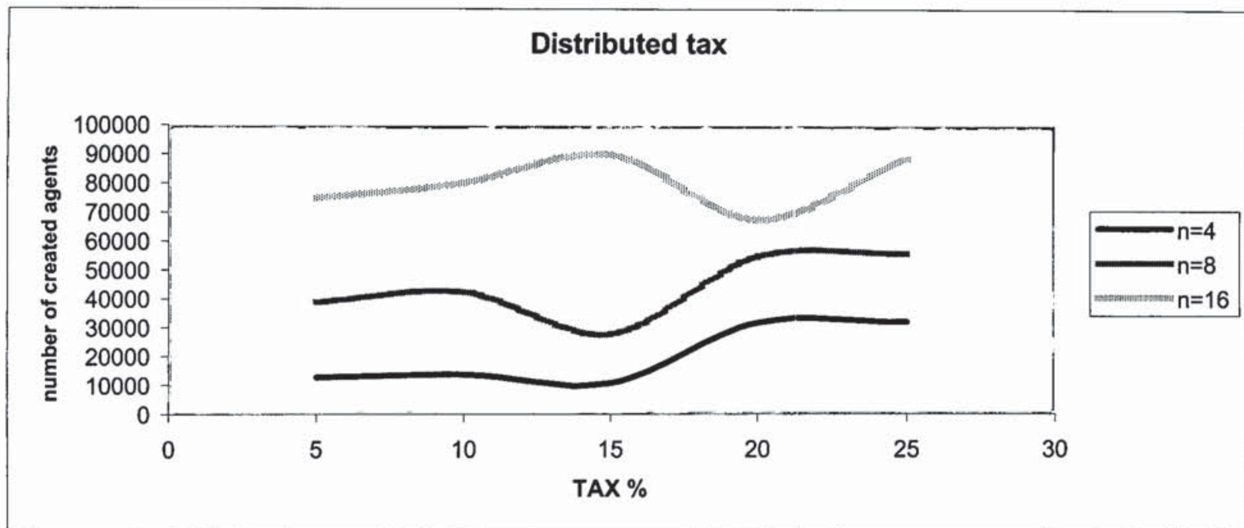


Figure 4.10: Distributed Tax

We vary the tax percentage from 5 to 30%, and we conclude that the best configuration is when the tax = 15 % for n=4 and n=8, and 20% for n=16. This means that if we keep the tax in the system, this will allow some agents to live more, and have a chance to contribute. But if we raise the tax above 20%, the agent will start to lose too much of their wealth, and they will blame the evaluation function, and this is obvious if we look at the figure above

4.7.8 Sub-Agent Experiment

Any wealthy agent, and we mean by a wealthy agent, an agent whose wealth is at least ten times more than the average wealth, can create a sub-agent. The main idea is the success of the wealthy agent. If an agent is wealthy, them its rule and its estimation function are very good. Therefore the sub-agent will be a mutation (See Sec 4.4) of its

creator, keeping the good rules in the system. How often we should create sub-agent, is in the result of this experiment.

We fix the tax to10%

A wealthy agent can create a sub-agent that is a modification of itself,

with probability of creation = X

We vary X between 10% and 40%

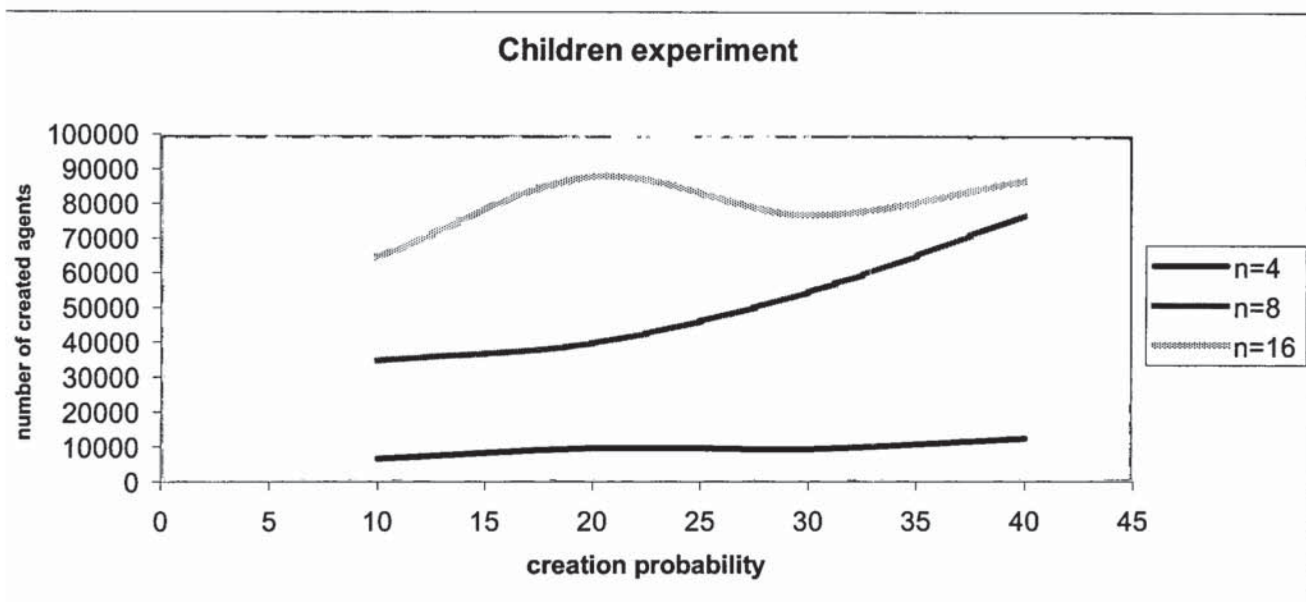


Figure 4.11: Sub-Agent Experiment

The best results in terms of the number of created agents are yielded when the probability of creating a sub-agent is 10%. The increased number of agents that might never contribute to the system could explain this.

4.7.9 Recursive Reward

In the previous experiment, the reward which the agent get, is not shared with any other agent. Here we change the design of our prototype. We want to examine the result of reward sharing.

We let the rewarded agent share its reward with all the other agents who contributed.

The update is as follows:

$$\text{reward at state(i-1)} = \text{reward at state(i-1)} + \frac{reward\ at\ stat(i)}{n^{distance}}$$

where distance is the distance between the state(i) and the current state,

and n is the number of executed agent (agents who contributed)

Note that the reward could be negative

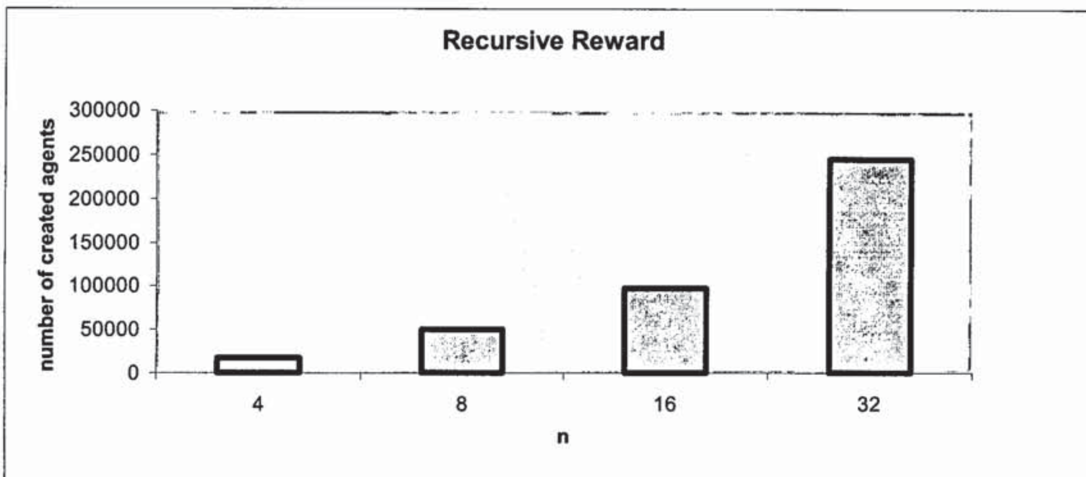


Figure 4.12: Recursive Reward

Since the reward was distributed, this means that the agents which contribute only one time(even if their contribution is bad) still get a reward from the system. Similarly, a good agent could be affected by the bad action of bad agent, since it will always get the negative reward. As a conclusion, the number of agents increased, and the time calculation was huge. We say that distributing the reward, gives advantage to poor users, which will not eventually evolve to be rational, because the bad rules they have are the

probability of changing if given a second chance?

4.8 Conclusion

In this section we shall study the results reported by our prototype.
We simulate our system on n=4, 8 and 16.
The system was able to solve 20 instances simultaneously, and provided the following results.

Results for n=4

We sort the agents in decreasing order of wealth, and we chose the wealthiest four agent

Agent 645 total wealth = 13.231 frequency = 231

Rule[1] if (stack(1),*,All) != (stack(goal),*,ALL) then (grab(1),drop(3))
Bidding function = 0.76 * 13.231

Rule[2] if (stack(1),*,ALL) != (stack(3),*,ALL) then (grab(2),drop(3))
Bidding function = 0.14 * 13.231

Rule[3] if (stack(3),2,ONE) != (stack(goal),4,ONE) then (grab(3),drop(1))
Bidding function = 0.42* 13.231

Rule[4] if (stack(goal),2,ALL) = (stack(2),*,ALL) then (grab(2),drop(1))
Bidding function = 0.11* 13.231

Rule[5] if (stack(l),3,ONE) = (stack(3),1,ONE) then (grab(3),drop(1))
Bidding function = 18%

Agent 962 total wealth = 13.031 frequency = 211

Rule[1] if (stack(2),*,ALL) != (stack(goal),*,ALL) then (grab(1),drop(3))
Bidding function = 12 * 13.031

Rule[2] if (stack(2),1,ONE) = (stack(1),2,ONE) then (grab(3),drop(1))
Bidding function = 8* 13.031

Rule[3] if (stack(goal),*,ALL) != (stack(1),*,ALL) then (grab(1),drop(2))
Bidding function = 0.42* 13.031

Rule[4] if (stack(goal),2,ONE) = (stack(1),1,ONE) then (grab(2),drop(1))
Bidding function = 0.21* 13.031

Rule[5] if (stack(3),2,ALL) != (stack(goal),2,ALL) then (grab(2),drop(3))
Bidding function = 0.15* 13.031

Agent 318 total wealth = 9.31 frequency = 291

Rule[1] if (stack(goal),*,ALL) != (stack(3),*,ALL) then (grab(3),drop(1))
Bidding function = 0.59 * 9.31

Rule[2] if (stack(3),1,ALL) != (stack(2),1,ALL) then (grab(1),drop(2))
Bidding function = 0.12* 9.31

Rule[3] if (stack(1),2,ONE) != (stack(2),1,ONE) then (grab(2),drop(1))
Bidding function = 0.7* 9.31

Rule[4] if (stack(2),*,ALL) ! = (stack(1),*,ALL) then (grab(1),drop(3))
Bidding function = 0.25* 9.31

Rule[5] if (stack(2),3,ONE) = (stack(1),1,ONE) then (grab(1),drop(2))
Bidding function = 0.22%* 9.31

Agent 95 total wealth = 4.31 frequency = 91

Rule[1] if (stack(1),*,ALL) != (stack(goal),*,ALL) then (grab(1),drop(2))
Bidding function = 0.69 * 4.31

Rule[2] if (stack(2),2,ONE) = (stack(1),1,ONE) then (grab(1),drop(2))
Bidding function = 0.11* 4.31

Rule[3] if (stack(1),1,ALL) != (stack(2),1,ALL) then (grab(1),drop(2))
Bidding function = 0.17* 4.31

Rule[4] if (stack(2),1,ONE) ! = (stack(1),3,ONE) then (grab(1),drop(3))
Bidding function = 0.3* 4.31

Rule[5] if (stack(2),*,ALL) != (stack(1),*,ALL) then (grab(2),drop(1))
Bidding function = 0.13* 4.31

Algorithm.

These rules can be used to generate an algorithm for the BW problem. Such an algorithm is shown below.

Rule[1] of agent 645 and rule[3] of agent 962 and rule[1] of agent 95, summarize the fact of emptying the stack 1 if there is no match. Therefore as long as there is no match between stack(goal) and the first stack, keep emptying the first stack

Rule[4] of agent 645, and rule[2] of agent [962] and rule[5] of agent [95] and rule[1] of agent[318] are building up the first stack

All the other rules are simply used to fetch for the correct block

If we want to build an algorithm, we can state the following

stack1.
If there is a full match, move block from stack 2 or stack 3 to stack 1.

Fill the first stack from stack 3 or stack 2
If stack1 and the goal stack match, add block from stack 2 to stack1
If stack3 and the goal stack match, add block from stack 3 to stack 1
If the block moved is not correct, remove it and put another block, till you have a match.

Results for n= 8

We took the best four agents from the results with n = 4 and we integrated them in the execution of n=8
We initialized the frequency and the weight of the integrated agent to zero, but we kept the estimation function value as it is

We sort the agents in decreasing wealth, and we select the best four agents

Agent 1548 total wealth = 31.432 frequency = 219

Rule[1] if (stack(goal),*,ALL) = (stack(3),*,ALL) then (grab(3),drop(1))
Bidding function = 0.65 *31.432

Rule[2] if (stack(2),2,ALL) = (stack(3),*,ALL) then (grab(1),drop(2))
Bidding function = 0.21%*31.432

Rule[3] if (stack(goal),2,ONE) != (stack(2),5,ONE) then (grab(2),drop(3))
Bidding function = 0.63*31.432

Rule[4] if (stack(1),4,ONE) = (stack(3),2,ONE) then (grab(2),drop(1))
Bidding function = 21.3*31.432

Rule[5] if (stack(3),*,ALL) = (stack(2),*,ALL) then (grab(3),drop(2))
Bidding function = 0.33*31.432

Agent 962 total wealth = 26.537 frequency = 111

Rule[1] if (stack(2),*,ALL) != (stack(goal),*,ALL) then (grab(1),drop(3))
Bidding function = 0.9*26.537

Rule[2] if (stack(3),1,ONE) != (stack(1),6,ONE) then (grab(2),drop(1))
Bidding function = 0.027*26.537

Rule[3] if (stack(goal),*,ALL) != (stack(1),*,ALL) then (grab(3),drop(1))
Bidding function = 0.83*26.537

Rule[4] if (stack(1),2,ONE) = (stack(2),2,ONE) then (grab(3),drop(3))
Bidding function = 0.13*26.537

Rule[5] if (stack(2),3,ONE) != (stack(goal),1,ONE) then (grab(1),drop(2))
Bidding function = 0.15*26.537

Agent 1835 total wealth = 19.31 frequency = 191

Rule[1] if (stack(2),5,ONE) != (stack(1),2,ONE) then (grab(1),drop(2))
Bidding function = 0.13*19.31

Rule[2] if (stack(2),3,ALL) != (stack(3),3,ALL) then (grab(1),drop(3))
Bidding function = 0.21*19.31

Rule[3] if (stack(goal),*,ALL) != (stack(2),*,ALL) then (grab(2),drop(1))
Bidding function = 0.11%*19.31

Rule[4] if (stack(2),1,ALL) = (stack(1),3,ONE) then (grab(3),drop(1))
Bidding function = 0.34*19.31

Rule[5] if (stack(goal),*,ALL) != (stack(1),*,ALL) then (grab(3),drop(1))
Bidding function = 0.35*19.31

Agent 475 total wealth = 17.31 frequency = 296

Rule[1] if (stack(goal),3,ONE) != (stack(1),4,ONE) then (grab(1),drop(2))
Bidding function = 0.77*17.31

Rule[2] if (stack(1),*,ALL) = (stack(3),*,ALL) then (grab(3),drop(2))
Bidding function = 0.69%*17.31

Rule[3] if (stack(2),6,ONE) != (stack(3),1,ONE) then (grab(1),drop(2))
Bidding function = 0.29*17.31

Rule[4] if (stack(goal),1,ONE) ! = (stack(1),3,ONE) then (grab(1),drop(2))
Bidding function = 0.83*17.31

Rule[5] if (stack(2),*,ALL) = (stack(3),*,ALL) then (grab(1),drop(2))
Bidding function = 0.16*17.31

Algorithm.

These rules can be used to generate an algorithm for the BW problem. Such an algorithm is shown below.

Notice that agent 962 stayed in the system, although it changed some of its rule, but its bidding estimation function increased. Notice that its highest bidding rule is not the same.

Rule[3] of agent 962 summarizes the fact of emptying the stack 1 if there is no match. So as long as there is no match between stack(goal) and the first stack, keep emptying the first stack

Rule[1] of agent 1548, tries to put a correct block on stack1
Rule[2] of agent 962 and rule[3] of agent [1835] are building up the first stack

If we want to build an algorithm, we can state the following
match between the goal stack and the first stack, empty stack1.
If there is a full match, move block from stack 2 or stack 3 to stack 1.

Fill the first stack from stack 3
If stack1 and the goal stack match, add block from stack 2 to stack1
If stac3 and the goal stack match, add block from stack 3 to stack 1
If the block moved is not correct, remove it and put another block, till you have a match.
If all the blocks match on stack1, grab from stack2 and drop instack1.

Results for n= 16

We took the best four agents from the results with n = 8 and we integrated them in the execution of n=16
We initialized the frequency and the weight of the integrated agent to zero, but we kept the estimation function value as it is

We sort the agents in decreasing wealth, and we select the best four agents

Agent 1548 total wealth = 45.432 frequency = 443

Rule[1] if (stack(goal),*,)ALL != (stack(1),*,ALL) then (grab(1),drop(3))
Bidding function = 0.85* 45.432

Rule[2] if (stack(2),7,ONE) = (stack(1),4,ONE) then (grab(1),drop(2))
Bidding function = 0.16* 45.432

Rule[3] if (stack(1),*,ALL) = (stack(2),*,ALL) then (grab(2),drop(3))
Bidding function = 0.46* 45.432

Rule[4] if (stack(goal),12,ONE) = (stack(1),2,ONE) then (grab(2),drop(1))
Bidding function = 0.86* 45.432

Rule[5] if (stack(3),2,ONE) != (stack(2),5,ONE) then (grab(1),drop(2))
Bidding function = 0.43* 45.432

Agent 99 total wealth = 32.271 frequency = 311

Rule[1] if (stack(1),11,ONE) != (stack(goal),2,ONE) then (grab(3),drop(1))
Bidding function = 16*32.271

Rule[2] if (stack(1),*,ALL) != (stack(2),*,ALL) then (grab(1),drop(3))
Bidding function = 27*32.271

Rule[3] if (stack(goal),5,ALL) = (stack(2),5,ALL) then (grab(3),drop(1))
Bidding function = 15*32.271

Rule[4] if (stack(goal),6,ALL) != (stack(1),6,ALL*) then (grab(3),drop(2))
Bidding function = 19*32.271

Rule[5] if (stack(3),13,ONE) = (stack(1),3,ONE) then (grab(1),drop(3))
Bidding function = 55%

Agent 10656 total wealth = 23.012 frequency = 263

Rule[1] if (stack(1),12,ONE) != (stack(goal),1,ONE) then (grab(2),drop(1))
Bidding function = 0.24%

Rule[2] if (stack(1),4,ALL,) = (stack(2),4,ALL) then (grab(3),drop(1))
Bidding function = 0.29%

Rule[3] if (stack(goal),*,ALL) != (stack(1),*,ALL) then (grab(2),drop(1))
Bidding function = 0.68%

Rule[4] if (stack(2),6,ALL) = (stack(3),6,ALL) then (grab(3),drop(1))
Bidding function = 0.17%

Rule[5] if (stack(1),4,ONE) != (stack(3),10,ONE) then (grab(3),drop(2))
Bidding function = 0.37%

Agent 5398 total wealth = 17.31 frequency = 745

Rule[1] if (stack(goal),5,ALL) != (stack(1),5, ALL) then (grab(1),drop(2))
Bidding function = 0.51*17.31

Rule[2] if (stack(3),*,ALL) = (stack(2),*,ALL) then (grab(3),drop(2))
Bidding function = 0.29*17.31

Rule[3] if (stack(1),14,ONE) != (stack(2),1,ONE) then (grab(3),drop(2))
Bidding function = 0.11*17.31

Rule[4] if (stack(1),4,ALL) ! = (stack(3),4,ALL) then (grab(1),drop(2))
Bidding function = 0 38*17.31

Rule[5] if (stack(2),8,ONE) = (stack(goal),3,ONE) then (grab(1),drop(2))
Bidding function = 0.64*17.31

Algorithm.

These rules can be used to generate an algorithm for the BW problem. Such an algorithm is shown below.

Notice that agent 1548 stayed in the system, although it changed some of its rule, but its bidding estimation function increased. Notice that its highest bidding rule is not the same.

Rule[1] of agent 1548 and rule[3] of agent 10656 summarize the fact of emptying the stack 1 if there is no match. Therefore, as long as there is no match between stack(goal) and the first stack, keep emptying the first stack

Rule[4] of agent 99 and rule[1] of the same agent, try to put a correct block on stack1
Rule[1] of agent 5398 and rule[3] of agent 9 are building up the first stack

All the other rules are helping in popping you the correct block to the surface.

If we want to build an algorithm, we can state the following:
As long as we do
stack1.If there is a full match, move block from stack 2 or stack 3 to stack 1.
Search for the corresponding next correct block, by moving blocks around
Move it to stack1
Fill the first stack from stack 2 and 3 with the same process.

CHAPTER V

CONCLUSION

This thesis discussed the behavior of an economy from a reinforcement learning perspective. A literature review of reinforcement learning was summarized and experiments on solving the Blocks World problem using the artificial economy framework was developed. In these experiments we studied the effect of taxes, exploration versus exploitation, property rights, average wealth and other variables related to artificial economy. The economy

We found that human like intelligence can be obtained as a result of the collaboration among multiple agents that cooperate to solve a given problem.

For future research, we are interested in developing a system .based on the artificial economy framework that evolve to solve the stock bidding problem

REFERENCES

fttp://www.neci.nj.com:80/homepages/eric/eric.html, 1999.

[2] E.B

fttp://www.neci.nj.com:80/homepages/eric/eric.html, 1999.

fttp://www.neci.nj.com:80/homepages/eric/eric.html, 1999.

-

from fttp://www.neci.nj.com:80/homepages/eric/eric.html, 1999.

ational Implications of Computer Learning by Artificial

[6] J. Koza, Genetic Programming. MIT press , Cambridge, (1992).

-line q-learning using connectionist
-INFENG/TR 166, Cambridge,
University Engineering Department, 1994.

[8] R.S. Sutton and A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, 1998.

[9] Sutton, R.S. (1988). Learning to predict by the methods of temporal differences. Machine Learning 3: 9--44.

[10] Sebastian B. Thrun. The role of exploration in learning control. In David A. White and Donald A. Sofge, editors, Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches. Van Nostrand Reinhold, New York, NY, 1992.

[11] Singh, S.P., Sutton, R.S. (1996). Reinforcement Learning with replacing eligibility traces. Machine Learning 22: 123-158.

- earning, vol.
8, pp. 279--292, 1992.

APPENDIX A

```
#ifndef AGENTIncluded
#define AGENTIncluded

const int m = 5; // number of rules or strings per agent
const int n = 4; // number of blocks in the goal stack


class BlockWorld;


// Class agent represente the agent configuration
class AGENT
{
    friend BlockWorld;

    private:


        long int SubAgent;     // hold the id of the creator (if exists)
        long int id;           // sequential ID of the agent
        int exp1 [m][3];       // first expression of the left side ot the rule
        int operation[m];      // could be = or !=
        int exp2 [m][3];       // second expression of the left side of the rule
        int rule  [m][2];      // grab and drop, the right side of the rule
        double estimation [m];         // the updated value function
        int frequency[m];              // frequency  of the rule used
        double wealth;                 // agent wealth
        static double AverageWealth;   // total average wealth of all the agents
        static long int oid;           // object ID
        BlockWorld* world;             // The Class world reference


public:

    AGENT(BlockWorld* theWorld);
    AGENT(const AGENT&);
    int NumCorrect ();          // NumCorrect function
    int A();                    // calculate the A value of A * NumCorrect + B
    double UpdateB();           // update the B value of A * NumCorrect + B
    void B(double value);       // calculate the B value of A * NumCorrect + B
    void view ();                         // view the rule srtucture
    void changerules();                   // changing and mutation function
//    void copytofile(int);
    const AGENT& operator = (const AGENT& a);
};

#endif
```

```
#ifndef AGENTIncluded
#include "agent.h"
#endif

#ifndef BlockWorldIncluded
#include "blockworld.h"
#endif

#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
#include <time.h>


long int AGENT::oid = 1;        //satic value to keep the agent's ID
double AGENT::AverageWealth=0; //total average wealth of all the agents

AGENT::AGENT(BlockWorld* theWorld)
{
    int stacknumber;  //stack number could be 0(goal),1,2,3.
    int l,nb1,nb2;    //used to store some random numbers

    world = theWorld;

    // the agent rule has the following shape
    // if exp1[stack number,block number,type] OP exp2[stack number,block
number,type] then (grab[stack number] drop[stack number])
    // OP could be (= or !=)
    // block number is a number representing the block position on the stack or
the number of blocks to be matched
    // depending of the value of TYPE
    // if type = 1 then block number is the block position to be matched
    // if type = 0 then block number is the number of blocks to be matched
    SubAgent = 0; //hold the id of the creator, initialized to zero
    wealth = 0;
    id = oid;
    oid ++;

    for (int s=0;s<m;s++)
    {
       estimation [s] = 0;              // the updated value function
       frequency[s] = 0;                // frequency  of the rule used

       stacknumber = (rand() % 4);
          exp1[s][0] = stacknumber;       // fill the stack number in exp1

       operation[s] = (rand() % 2);       // 0 for '='  and 1 for '!='

       do
       {
          stacknumber = (rand() % 4);       // mahe sure that stack in exp1 !=
stack in exp2
       }while (stacknumber == exp1[s][0]); // in order for the grab-drop rule
make sense
       exp2[s][0] = stacknumber;           // fill the stack number in exp1


       l = (rand() % 2);       // with 50% probability
       if (l == 0)
       {
          l = (rand() % (n/2)) +1 ;
          if (l == n/2)                     // the block number = *
          {
```

```
                exp1[s][1] = exp2[s][1] = n;  // *
                exp1[s][2] = exp2[s][2] = 0;  // type = 0 (ALL)
             }
             else
             {
                exp1[s][1] = exp2[s][1] = l;  // block number
                exp1[s][2] = exp2[s][2] = 0;  // type = 1
             }
          }
          else
          {
            l = (rand() % n);
            exp1[s][1]  = l;              // block number
            l = (rand() % n);
            exp2[s][1]  = l;              // block number
            exp1[s][2] = exp2[s][2] = 1;  // type = 1

          }

          nb1 = (rand() % 3) +1;          // fill the stack number in the grab
part of the rule
          rule[s][0] = nb1;

          do
          {
             nb2 = (rand() % 3) +1;        // make sure grab is different from
drop
          }while ((nb1 == nb2));
          rule[s][1] = nb2;                // fill the stack number in the drop
part of the rule
          //cout<<" "<<nb1<<" "<<nb2<<endl;getchar();
     }
}


// copy constructor
AGENT::AGENT(const AGENT& a)
{
    world = a.world;

    SubAgent = a.SubAgent;
    wealth = a.wealth;
    AverageWealth = a.AverageWealth;
    id = a.id;
    for (int s=0;s<m;s++){
       for (int i=0;i<3;i++)
       {
           exp1[s][i] = a.exp1[s][i];
           exp2[s][i] = a.exp2[s][i];

       }
       rule[s][0] = a.rule[s][0];
       rule[s][1] = a.rule[s][1];
       estimation[s] = a.estimation[s];
       operation[s] = a.operation[s] ;
       frequency[s] = a.frequency[s];
    }

}

// assignement operator
const AGENT& AGENT::operator = (const AGENT& a)
{
```

```
    world = a.world;

    SubAgent = a.SubAgent;
    wealth = a.wealth;
    AverageWealth = a.AverageWealth;
    id = a.id;
    for (int s=0;s<m;s++)
    {
       for (int i=0;i<3;i++)
       {
           exp1[s][i] = a.exp1[s][i];
           exp2[s][i] = a.exp2[s][i];

       }

       rule[s][0] = a.rule[s][0];
       rule[s][1] = a.rule[s][1];
       estimation[s] = a.estimation[s] ;
       operation[s] = a.operation[s] ;
       frequency[s] = a.frequency[s];
    }
    return *this;
}


// NumCorrect Function
int AGENT::NumCorrect ()
{
    int numberOfCorrectBlocks=0;

    for (int i=1; i<=n; i++)
    {
       if (world->AXE[1][i] == world->AXE[0][i])
       {
          numberOfCorrectBlocks++;
       }
    }
    return numberOfCorrectBlocks;
}

// Calculate A, of the function A * NumCorrect + B
int AGENT::A()
{
    int value=0;

    for (int i=1; i<=n; i++)
    {
       if (world->AXE[1][i] == world->AXE[0][i])
       {
          value = value + n - i + 1 ;
       }
    }
    return value;
}


//Update the estimation function
double AGENT::UpdateB()
{
    double epsilon=0.3;
    return (world->lastAgentPayement - world->currentPayement) * epsilon;
}
```

```
void AGENT::B(double value)
{
    estimation[world->lastUsedRuleNumber] += value;

}



// changing a mutation functions
void  AGENT::changerules()
{
    int r,l,stacknumber,nb1,nb2;
    for (int i=0;i<5;i++)
    {
        r = (rand() % 100 ) + 1;
        if (r >= 70)                                //with 30 % change one rule
randomly
        {
            r = rand() % m;
            stacknumber = (rand() % 4);
               exp1[r][0] = stacknumber;                //fill the stack number in
exp1
            operation[r] = (rand() % 2);            // o for '='  and 1 for '!='

            do
            {
               stacknumber = (rand() % 4);          //make sure that stack in exp1
!= stack in exp2
            }while (stacknumber == exp1[r][0]);
            exp2[r][0] = stacknumber;               //fill the stack number in
exp2

            l = (rand() % 2);
            if (l == 0)
            {
               l = (rand() % (n/2)) +1 ;
               if (l == n/2)
               {
                  exp1[r][1] = exp2[r][1] = n;    //if l > n/2, this is the *
symbil
                  exp1[r][2] = exp2[r][2] = 0;    //type of assignement 0 = ALL,
1 = ONE
               }
               else
               {
                  exp1[r][1] = exp2[r][1] = l;
                  exp1[r][2] = exp2[r][2] = 0;  //type of assignement 0 = ALL,
               }
            }
            else
            {
               l = (rand() % n);
               exp1[r][1]  = l;
               l = (rand() % n);
               exp2[r][1]  = l;
               exp1[r][2] = exp2[r][2] = 1;

            }
```

```
            nb1 = (rand() % 3) +1;     // fill the grab
            rule[r][0] = nb1;
            do
            {
               nb2 = (rand() % 3) +1; // fill the drop, make sure that grab !=
drop
            }while ((nb1 == nb2));
            rule[r][1] = nb2;
        }
        else if ((r <= 70) && (r >= 40))
        {                                    // with probability 30% exchange right
side rule symbols
            int temp;
            r = rand() %m;
            temp = rule[r][0] ;
            rule[r][0] = rule[r][1];
            rule[r][1] = temp;
        }
        else if ((r >= 10) && (r <= 40)) // reshuffle rule stacks positions
        {
            for (int j=0;j<5;j++)
            {
               r = rand() % n;
               l = rand() % n;
               exp1[r][0] = exp2[l][0];
            }
        }
        else if (r <= 10)
        {
            l = rand() % 2;
            if (l == 0)     //left side rule
            {
                l = rand() % 2;
                if (l == 0)        //exp1 selected
                {
                   l = rand() % m;
                   r = rand() % n;
                   exp1[l][1] = r+1;
                }
                else
                {
                   l = rand() % m; // exp2 has been selected
                   r = rand() % n;
                   exp2[l][1] = r+1;
                }
            }
            else
            {
                l = rand() % 2;
                if (l == 0)        //grab has been selected selected
                {
                   l = rand() % m;
                   r = (rand() % 3) + 1;
                   rule[l][0] = r;
                }
                else                  // drop has been selected
                {
                   l = rand() % m;
                   r = (rand() % 3) + 1;
                   rule[l][1] = r;
                }
            }
        }
```

```
    }
}




// this function displays the rule on the screen
void AGENT::view()
{
   for (int s=0;s<m;s++)
   {
      cout<<" agent["<<id<<"] rule["<<s+1<<"]"<<endl;
      cout<<"IF (stack[" ;
      if (exp1[s][0] == 0)
          cout<<"Goal";
      else
          cout<<exp1[s][0];

      if (exp1[s][1] == n)
          cout<<"],"<<"*"<<",";
      else
          cout<<"],"<<exp1[s][1]+1<<",";

      if (exp1[s][2] == 1)
          cout<<"ONE)";
      else
          cout<<"ALL)  ";


      if (operation[s] == 0)
          cout<<"=  ";
      else
          cout<<"!=";

      cout<<"(stack[" ;
      if (exp2[s][0] == 0)
          cout<<"Goal";
      else
          cout<<exp2[s][0];

      if (exp2[s][1] == n)
          cout<<"],"<<"*"<<",";
      else
          cout<<"],"<<exp2[s][1]+1<<",";

      if (exp2[s][2] == 1)
          cout<<"ONE)";
      else
          cout<<"ALL)  ";


      cout<<" THEN  GRAB from stack["<<rule[s][0]<<"] DROP in
stack["<<rule[s][1]<<"]"<<endl;
   }
   cout<<"================================\n\n";
   getchar();
}
```

```
#ifndef AgentNodeIncluded
#define AgentNodeIncluded

#ifndef AGENTIncluded
#include "agent.h"
#endif

#include <stdlib.h>

// agent node class, freind of agent list class
class AGENTNODE
{
    friend class AGENTLIST;

    private:
        AGENT agent;
        AGENTNODE *next;     //next node
        AGENTNODE *previous; //previous node
    public:
        AGENTNODE (const AGENT& agent, AGENTNODE* next = NULL, AGENTNODE*
previous = NULL );
};

#endif
#ifndef AgentNodeIncluded
#include "agentnode.h"
#endif

#ifndef AGENTIncluded
#include "agent.h"
#endif


AGENTNODE::AGENTNODE (const AGENT& a, AGENTNODE* n, AGENTNODE* p ): agent( a )
{
    next = n;      //constructor
    previous = p;
}
```

```
#ifndef AgentListIncluded
#define AgentListIncluded

#ifndef AgentNodeIncluded
#include "agentnode.h"
#endif


// class that represent the linked list having all the active agents
class AGENTLIST {
private:
    AGENTNODE *head;        // head of the linked lisr
    AGENTNODE *current;     // current position of the linked list
public:
    AGENTLIST();            // constructor
    ~AGENTLIST();           // destractor
    void reset ();          // go to the first position in the list
    AGENT & get();          // get an agent from the current position
    bool next();            // move one node in the list
    void remove();
    bool currentIsNull(); // test if the current node is null
    void insert(const AGENT& agent);  // insert a node in the list
};

#endif
```

```
#ifndef AgentListIncluded
#include "agentlist.h"
#endif

#include <stdio.h>

AGENTLIST::AGENTLIST()
{
    head = NULL;          // constructor
    current = NULL;
}

void AGENTLIST::insert(const AGENT& agent)
{
    AGENTNODE * newagentnode= new AGENTNODE (agent);

    if (head == NULL )
        head = current = newagentnode;
    else
    {
       AGENTNODE *temp = current;  // insert agent on the list
       current = newagentnode;
       current->previous = temp;
       temp->next = current;
    }
}


void AGENTLIST::remove()
{
    // pre-condition: current != NULL

   if (current->previous != NULL) // current is not the head
   {
      AGENTNODE *temp = current->previous;
      temp->next = current->next;
      delete current;              // remove agent from list
      current = temp->next;
   }
   else
   {
      AGENTNODE *temp = head->next;
      delete head;
      if( temp != NULL )
          temp->previous = NULL;    // remove agent from list
      head = current = temp;
   }

}



AGENTLIST::~AGENTLIST()
{
    AGENTNODE *p = head ;
    while (p != NULL){         // distructor
        head = p->next;
        delete p;
        p = head;
    }
}
```

```
// test if the currrent node is null
bool AGENTLIST::currentIsNull()
{
   return (current == NULL)     ;
}


// go to the first node in the linked list
void AGENTLIST::reset()
{
   current = head;
}


// get the current agent
AGENT & AGENTLIST::get()
{
    return current->agent;
}


// move the current agent to the next node (if not null)
bool AGENTLIST::next()
{
    if (current == NULL )
        return false;
    current = current->next;
    return true;
}
```

```
#ifndef BlockWorldIncluded
#define BlockWorldIncluded

//int const n = 4; // number of blocks

#ifndef AgentIncluded
#include "agent.h"
#endif



// class that represent the word configuration
class BlockWorld
{
    friend AGENT;

    enum color {red,blue,green};      // the color used in the game
    int AXE[4][n+1];                  // an instance of the goal stack
    double lastAgentPayement;         //keep the last agent payement (the last
value of the world)
    double currentPayement;           //current value of the world
    int lastUsedRuleNumber;           //last reule used on the world
    long int nbOfCreatedAgent;        //number of created agent
    long int nbOfRemovedAgent;        //number of removed agent

public:
    BlockWorld();

    void printcolor (int x);          // graphic world representation function
    void putcolor (int x);            // graphic world representation function
    void printworld ();               // graphic world representation function
    int match (AGENT a);              // test the matching
    bool fire (int g, int d);         // function that fire the agent rule and
make changes tot he world
    bool finished ();                 // test if the gane was over
    int run();                        // simulate the block world game
    void setnumberofremovedagent(); // count the removed agent
};

#endif
```

```
#ifndef BlockWorldIncluded
#include "blockworld.h"
#endif

#ifndef AgentIncluded
#include "agent.h"
#endif

#ifndef AgentListIncluded
#include "agentlist.h"
#endif


#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
#include <time.h>


BlockWorld::BlockWorld()
{
    //WOELD[n] = {1,0,1,2}
    AXE[0][0]=1;    AXE[0][1]= 0;    AXE[0][2]= 1;    AXE[0][3]= 2; // an
instance of the world

    //AXE1[n+1]  = {1,-1,-1,-1,-1};
    AXE[1][0]=1;    AXE[1][1]=-1;    AXE[1][2]=-1;    AXE[1][3]=-1;
AXE[1][4]=-1;

    //AXE2[n+1]  = {1,2,-1,-1,-1};  //the first world distrubution
    AXE[2][0]=1;    AXE[2][1]=2;     AXE[2][2]=-1;    AXE[2][3]=-1;
AXE[2][4]=-1;

    //AXE3[n+1]  = {0,-1,-1,-1,-1};
    AXE[3][0]=0;    AXE[3][1]=-1;    AXE[3][2]=-1;    AXE[3][3]=-1;
AXE[3][4]=-1;

    nbOfRemovedAgent = 0;
}


void BlockWorld::setnumberofremovedagent()
{
    nbOfRemovedAgent++;
}


// this function print the world status
void BlockWorld::printworld ()
{
   int i=0;

   for (i=0;i<n;i++)
   {

       printcolor(AXE[0][n-i-1]);
       if (AXE[1][n-i-1] != -1)
          printcolor(AXE[1][n-i-1]);
       else
          cout<<"    |         ";
          if (AXE[2][n-i-1] != -1)
          printcolor(AXE[2][n-i-1]);
       else
```

```
            cout<<"     |          ";
        if (AXE[3][n-i-1] != -1)
            printcolor(AXE[3][n-i-1]);
        else
            cout<<"     |          ";

        cout<<endl;
    }

      cout<<"____|_____________|_____________|_____________|____\n\n";
      cout<<endl;
}


// function to replace number on the stacks by their colors
void BlockWorld::printcolor (int x)
{
    if (x == 0)
        cout << " (RE|D)       ";
    else if (x == 1)
        cout << " (BL|UE)      ";
    else if (x == 2)
        cout << " (GR|EEN)     ";
    else cout<<x;

}


void BlockWorld::putcolor (int x)
{
/*    FILE *f = fopen ("data.txt","a");
    if (x == 0)
        fputs(" (RE|D)       ",f);
    else if (x == 1)
        fputs(" (BL|UE)      ",f);
    else if (x == 2)
        fputs(" (GR|EEN)     ",f);
    fclose(f);
 */
}


// this function take an agent as aragument and test if and if its rules match
// if so it return the matched rule number, otherwise it return -1
int BlockWorld::match (AGENT a)
{
    int i = 0;
    bool cond=true;
    int nbOfBlocks1,nbOfBlocks2,stack1,stack2,type;

    for (int s=0;s<m;s++)                 //m is the number of rules
    {

        nbOfBlocks1 = a.exp1[s][1];       //store the nb of blocks in the rule,
exp1
        nbOfBlocks2 = a.exp2[s][1];       //store the nb of blocks in the rule,
exp2
        stack1 = a.exp1[s][0];            //the first stack nb to bematched
        stack2 = a.exp2[s][0];            //the secong stack nb to bematched
        type = a.exp2[s][2];              //matching type, if 0 match all, if 1
match 1
```

```
        // this test if the rule match
        cond=true;
        if ((nbOfBlocks1 == nbOfBlocks2) && (a.operation[s] == 0) && (type ==
0))
        {
            for (i=0;i<nbOfBlocks1;i++)
                if (AXE[stack1][i] != AXE[stack2][i])
                    cond=false;
            if(cond == true)
               return s+1;
        }

        // this test if the rule match
        cond=true;
        if ((a.operation[s] == 1) && (type == 0))     //operation is (= or !=)
        {
            for (i=0;i<nbOfBlocks1;i++)
                if (AXE[stack1][nbOfBlocks1] == AXE[stack2][nbOfBlocks2])
                    cond=false;
            if(cond == true)
               return s+1;
        }

        // the rule is of the form (if exp1[stacknumber, blocknumber, type) OP
exp2[stacknumber, blocknumber, type)
        // OP is (= or !=)
        //type, is the type of comparision.
        //if type = 1, we match all the block from 0 till blocknumber
        //if type = 0, we only match only the stacknumber block
        if ((a.operation[s] == 1) && (type == 0))
        {
            for (i=0;i<nbOfBlocks1;i++)
                if (AXE[stack1][nbOfBlocks1] == AXE[stack2][nbOfBlocks2])
                    cond=false;
            if(cond == true)
               return s+1;
        }


    }
    return -1;

}


//this function take the as argument the rule parameter (grab and drop)
//make the changement to the world.
bool BlockWorld::fire (int g, int d) //this function test if
{
    int temp,t,f;

    if (g == d)              //grab stack = drop stack, useless function
        return false;

    if (g == 1)
    {
        if (AXE[1][0] == -1)         //if grab = 1, and stack[1] is empty,
useless rule
            return false;
        t=0;
        while (AXE[1][++t] != -1)
            f=0;
        t--;
```

```
        temp = AXE[1][t];
        AXE[1][t] = -1;

        if (d == 2)                     //else if drop is on stack 2 make the
changenemt to the world
        {
           t=0;
           while (AXE[2][t++] != -1)
               f=0;
            AXE[2][--t] = temp;
        }
        else
        {
           t=0;
           while (AXE[3][t++] != -1)
               ;
            AXE[3][--t] = temp;
        }
        return true;
    }


    if (g == 2)                             //if grab = 2, and stack[2] is empty,
useless rule
    {
        if (AXE[2][0] == -1)
            return false;
        t=0;
        while (AXE[2][++t] != -1)
            ;
        t--;
        temp = AXE[2][t];
        AXE[2][t] = -1 ;//grab

        if (d == 1)
        {
           t=0;
           while (AXE[1][t++] != -1) //else if drop is on stack 1 make the
changenemt to the world
               f=0;
            AXE[1][--t] = temp;
        }
        else
        {
           t=0;
           while (AXE[3][t++] != -1)
               ;
            AXE[3][--t] = temp;
        }
        return true;
    }


    if (g == 3)                             //if grab = 3, and stack[3] is empty,
useless rule
    {
        if (AXE[3][0] == -1)
            return false;
        t=0;
        while (AXE[3][++t] != -1)
            ;
        t--;
        temp = AXE[3][t];
```

```
        AXE[3][t] = -1; //grab

        if (d == 2)
        {
           t=0;
           while (AXE[2][t++] != -1)
                ;
            AXE[2][--t] = temp;
        }
        else
        {
           t=0;
           while (AXE[1][t++] != -1)
                ;
            AXE[1][--t] = temp;
        }
        return true;
    }
    return false;

}



//this function test if the game has finished
bool::BlockWorld::finished ()
{
    for (int i=0;i<n;i++)
    {
        if (AXE[1][i] != AXE[0][i])   //if stack[goal] = stack[1] ---> finish
game
            return false;
    }
    return true;
}



int BlockWorld::run ()
{
    double espilon = 0.1;                            //used for discount factor
    double taxvalue = 10;                            //10 percent for taxes
payed by the agent
    bool cond,newAgentExists,previousagentexists;    //boolean value used to
track some condition
    int winningrule,newagentrule;                    //keep the last winning
agent and its winning ruke
    double bid,biggest = 0;                          //hold the aution value

    int tempworld[4][n+1];                           //used to hold old world
representation
    double estimation=0;                             //hold the value estimation

    AGENT tempagent (this) ;          //temporary agent
    AGENT winningagent (this) ;       //winning agent
    AGENT newagent  (this) ;
    AGENT lastagent (this) ;


    long int itiration=0;           //use to remove agent after 10000 itiration
    int rulenb,g,d;                 //used to store grab and drop values
```

```
    int NbOfActifAgent = 0;
    AGENTLIST allagent;

    printworld();

    newAgentExists = false;
    previousagentexists = false;   //at the begining no previous owner exists
    nbOfCreatedAgent = 0;          //initialize the number of created agent to
0


    while (1) //the first while
    {

      itiration++;
      AGENT a(this);
      allagent.reset();              //create new agent

      allagent.insert(a);            //add it to the world
      nbOfCreatedAgent++;
      allagent.reset();
      cond = false;

      while( !allagent.currentIsNull()) //the second while
      {
         tempagent = allagent.get();        //start the auction
         rulenb =match(tempagent);          //if any agent match
         if (rulenb != -1 )
         {
             cond = true;                   //save the world representation
             for (int i=0;i<4;i++)
                for (int j=0;j<=n;j++)
                   tempworld[i][j] = AXE[i][j];


              g = tempagent.rule[rulenb][0];
              d = tempagent.rule[rulenb][1];
              fire(g,d);                    //change the world by firing his
rule
              estimation = tempagent.A() * tempagent.NumCorrect() +
tempagent.estimation[rulenb - 1];
              estimation = estimation/ (n*n); //calculate the estimation useing
A*NumCorrect + B
              bid = estimation * tempagent.wealth; //calculatr the BID

              if (tempagent.wealth == 0)  //if this is a nre agent mark that
              {
                  newagent = tempagent;
                  newagentrule = rulenb;
                  newAgentExists = true;
              }

              for (i=0;i<4;i++)                      //restore the world
presentation
                 for (int j=0;j<=n;j++)              //since it was an
estimation and not
                    AXE[i][j] = tempworld[i][j];    //a real rule firing


              if (bid> biggest)
              {
                  biggest = bid;                     //keep the highest bidder
                  winningrule = rulenb;
```

```
                    winningagent = tempagent ;
                }
            }
            allagent.next(); //repest till all the agent has try if they match
        }//end the second while


      if (cond == false)        //if no agent match
      {
         while (cond == false)
         {
            AGENT a(this);      //keep creating agent till one of the new
created match
            allagent.reset();
            allagent.insert(a);
            nbOfCreatedAgent++;
            tempagent = allagent.get();
            rulenb =match(tempagent);
            if (rulenb != -1 )
            {
               cond = true;              //the first new created agent who match
               tempagent.wealth = 1;     //initialize his wealth to zero
               winningrule = rulenb;
               winningagent = tempagent; // and make him the winning agent

               estimation = tempagent.A() * tempagent.NumCorrect();
               estimation = estimation / (n*n);
               bid = estimation;
            }
         }
      }// enf if(cond == false)
      else if (newAgentExists == true) //if there is a match, and one of the
matched agent suppose
      {                                //to be a new agent
          if (biggest > 0 )
          {
              newagent.wealth = biggest + espilon; //make his bid value =
highest bid plus epsilon
              bid = newagent.wealth;               // and his wealth = his bid
          }
          else
          {
              newagent.wealth = 1;        //if he is the only agent match,
wealth =1
              estimation = newagent.A() * newagent.NumCorrect();
              estimation = estimation/ (n*n);
              bid = estimation;
          }
          winningagent = newagent;
          winningrule  = newagentrule;
          newAgentExists = false;
      }//end else if newAgentExists == true)

      tempagent = winningagent;          //store the winning agenet
      rulenb = winningrule;              //store the winning rule

      if (rulenb != -1 )
      {
         g = tempagent.rule[rulenb][0];
         d = tempagent.rule[rulenb][1];

         if (fire(g,d))                           //real fire of the rule
         {
```

```
            ++tempagent.frequency[rulenb];             //increment the frequency
of the rule
            NbOfActifAgent = NbOfActifAgent+1;
            printworld();                              //print the world after
change
           // getchar();
            currentPayement = bid;
            if (previousagentexists)
            {
               lastagent.wealth += bid;                //pay the previous agent
               lastagent.UpdateB();                    //previous owner update his
estimation function
            }
            lastAgentPayement  = bid ;
            lastUsedRuleNumber = rulenb;
            lastagent = tempagent;
            previousagentexists = true;
                                                        //Update the average
wealth
            tempagent.AverageWealth = (tempagent.AverageWealth+
tempagent.wealth)/ (nbOfCreatedAgent-nbOfRemovedAgent);

            tempagent.wealth -= bid;
            tempagent.wealth -= ((bid * taxvalue) / 100);  //pay taxes for the
world


            if (tempagent.SubAgent != 0)               //pay agent creator if
exists
            {
                allagent.reset();
                while( !allagent.currentIsNull())
                {
                    winningagent = allagent.get();
                    if (winningagent.id == tempagent.SubAgent)
                    {
                        winningagent.wealth += (tempagent.wealth / 10); //10 of
the wealth
                        tempagent.wealth -= (tempagent.wealth / 10);
                    }
                    allagent.next();
                }
            }


            if (rand() % 3 == 0 )
               tempagent.changerules();                   //change and mutate
agent rule


            if (tempagent.wealth > (10* tempagent.AverageWealth))
            {
                AGENT child (this) ;                      //create children
                child = tempagent;
                child.changerules();
                child.wealth = tempagent.wealth / 10;
                child.SubAgent = tempagent.id;
            }


            if (finished())
            {
```

```
                cout<<"  \n\n     Finished\n\n  "<<NbOfActifAgent<<" agent
participated ";
                getchar();
                return 0;
            }
        }//end if (rulenb != -1 )
      }
      if (rand() % itiration == 10)              //remove agent from the game
      {
         allagent.reset();
         while( !allagent.currentIsNull())
         {
            tempagent = allagent.get();
            if (tempagent.wealth < 0.1)          //if wealth < 0.1
            {
                allagent.remove();
                setnumberofremovedagent();      //increase the number of removed
agent
            }
            allagent.next();
         }
      }
   } // end the first while
   return 0;
}
```

```
#ifndef BlockWorldIncluded
#include "blockworld.h"
#endif

#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
#include<time.h>

main()

{
    BlockWorld theWorld;
    srand(time(NULL));
    theWorld.run();
    return 0;
}
```