Reinforcement Learning for Blackjack

Saqib A Kakvi
Supervisor: Dr Marco Gillies

April 30, 2009

Abstract

This report explores the development of an Artificial Intelligence system for an already existing framework of card games, called SKCards. This system was developed by the author as a way of learning Java. The old Artificial intelligence is highly flawed.

Reinforcement Learning was chosen as the method to be employed. Reinforcement Learning attempts to teach a computer certain actions, given certain states, based on past experience and numerical rewards gained. The agent either assigns values to states, or actions in states.

This will initially be developed for Blackjack, with possible extensions to other games. Blackjack is one of the simpler games and the only game in the SKCards package which needs an Artificial Intelligence agent. All the other games are single player.

Acknowledgements

I am hugely grateful to my project supervisor, Dr. Marco Gillies, and all the members of staff, for all the support and assistance they gave me. I would also like to thank my colleagues, whose help was invaluable. In particular I would like to thank David Jarret, Rohan Sherrard, Daniel Smith, Benoy Sen, Jusna Begum, Arron Donnely & Josephene Amirthanayagam. And a very special thank you to Nidhi Shah and Fatima Nassir for their help in proofreading this report.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The problem I am solving

This project has two goals. The major goal is the development of a Reinforcement Learning agent for Blackjack, using parts of an existing system. The second and less important goal is to improve the design of the existing system, so as to comply with software engineering norms. These goals may be solved independently and then combined to make the final system.

1.1.1 Reinforcement Learning Agent

When the initial system was built, it lacked a proper Artificial Intelligence Agent for the player to play against. This project aims to build an AI agent using Reinforcement Learning for Blackjack. This agent can be the dealer, or it can be another player in the game. Reinforcement Learning is very suited for the task of Blackjack due to the easily represented states. And the agent could learn to match a user's style of play, over time.

1.1.2 SKCards

SKCards was built a as framework for all card games. It provides the basic Cards, Packs of Cards and a standard interface. The concept is that any programmer could create their own card game and easily integrate it into SKCards. This, would give us an infinitely extendable suite of card games. This was created as an experiment with the Java programming language.

1.2 The layout of the report

This report will aim to break down the problem and present it in several smaller pieces. Each piece will logically follow from the next and will try to build on top of the last. In this way this report should give a smooth transition from theory to implementation and finally the results and conclusions.

The next chapter will discuss the separate parts of the project, namely, Blackjack and Reinforcement Learning. Then it discusses the two together. The following chapter will discuss the design aspects of the system. After that the development of the system will be looked at. This will detail the implementation of the system. Following that will the evaluation of the system. This will detail the experiments, the results and an analysis of these results. Finally, the last chapter will cover the self-evaluation.

Chapter 2

Background

2.1 Blackjack

2.1.1 Overview

Blackjack is a very simple game. The object is to get as close to, but not over 21 points in a maximum of 5 cards. This is achieved is by assigning values to each of the cards in the deck, irrespective of suit, as listed below:
2 - 10 $\Rightarrow$ Face value (i.e. a 2 is 2 points, a 3 is 3 points)
Courts (Jack, Queen, King) $\Rightarrow$ 10 points
A $\Rightarrow$ 1 or 11 points. The player decides if he wants to the ace "high" (11) or "low" (1). This can be changed in a situation where a "high" ace would cause a player to be bust.

2.1.2 Game play

The player is dealt 2 cards face down. The dealer then deals themselves 2 cards, 1st face up then face down. The player places a bet on his hand and turns over his cards. The player may then take any of several actions, listed below. These have been divided into core and extended rules, where the core rules make for the basis of the game, and the extended rules give additional gameplay.

Core Rules

HIT: The player may ask for another card from the pack to be added to their hand, this is known as a "hit" or "twist" (the term "hit" shall be used henceforth). The dealer "hits" the player's hand. The player's score is increased by the point value of the card added. If the player's score exceeds 21, they are "bust" and loses on that hand. If not, they can continue "hitting" until they are bust or have 5 cards in their hand.
STAND: If the player is satisfied with their hand they may "stand". This means that he is done with the current hand. The player then moves on to play any other hands they have(see Split in section 2.1.2). After "standing" on all their hands, the player ends their turn.

Extended rules

These rules are extensions of the rules in the previous section. They provide further functionality, however they are mostly based on the core rules.

BLACKJACK: The quickest way to win. A blackjack is an Ace and a Court Card (King, Queen or Jack) bringing the player up to a total of 21, known as a "natural". The player receives 2.5 times the money he was supposed to receive. This situation does not apply if the player has an Ace and a Ten. This rule was introduced in casinos to increase popularity of the game. Originally it only applied to an Ace and the Jack of Clubs, the "Black Jack", which is where the game derives it name from.

INSURANCE: If the dealer's exposed card is an ace , threatening a Blackjack, the player may take out "insurance" against a Blackjack. They place another bet on the table. If the dealer has Blackjack, they gets a 2-to-1 payout. If the dealer doesn't have a Blackjack, the player loses that money. As the name states, this serves as the player's backup, should they lose to a dealer blackjack.
SPLIT: If both the cards have the same face value (two Queens, two Fives, two Sevens…) the player may opt to "split" their hand into two separate hands. The cards are separated and one card is added to each hand to make two cards in each hand. The player then places the same bet on the second hand. A player may have up to 5 hands.
DOUBLE DOWN: If the total points of the players cards is greater than or equal to 11 the player may opt to "double down". In this case the player doubles their bet and the player's hand is hit, after which they are forced to stand, provided the player is not bust. This rule is basically a gamble by the player that they will need no more than 1 Card to reach a favourable score. This increased risk is why there is an increased reward.

After the player is finished, the next player plays in the same manner. After all players have finished playing, play moves to the dealer, who plays in a similar manner. After the dealer plays, the winner is the hand which is closest to 21. It must be noticed that the dealer generally has some restrictions imposed on them to make it fair on the players.

Dealer restrictions normally come in the form of a hitting threshold, a score above which they can no longer hit. It must be noted that a dealer can hit on high counting hands that are "soft" i.e. they have an Ace counting high. The standard policy in casinos tends to be a threshold score of 17.

Rule variations

The rules listed above are mostly in effect in the United States of America. There are some variations, specifically on the British version of the game, know as Pontoon. The details of this game are described in Parlett (1994). Although the basic principles of the game are the same, there are several rule variations. These variations are described below.

In this game, the role of dealer rotates between the players. The other players are called punters. The dealer also places a stake and plays as normal. Listed below are some of the other rule variations:
BUY: This is a variation on the hit rule. Under this rule, the player may request for a card to be dealt face-down to them. This allows the player to keep their score secret. To do this the player must pay an amount greater than what they paid for the last card, where applicable, and a less than their total stake. The fifth card of the hand may not be bought.
5-CARD TRICK: If a player has a hand of 5 cards and has not gone bust, then they automatically win. They receive a payment equivalent double to their stake.
PONTOON: In Pontoon Tens, Jacks, Queens & Kings are called tenths. A regular Pontoon is any tenth and an Ace. A Royal Pontoon is three sevens. The payouts for a Pontoon and Royal Pontoon are 2 times and 3 times respectively. Royal Pontoons can only be claimed by a punter. If the dealer holds a pontoon, then each punter pays the dealer double their stake. If the punter holds a Pontoon, they only pay an amount equal to their stake.
BANKER'S PLAY: The banker plays in a similar way as the punters, however there are some restrictions on their play. The banker is not allowed to split pairs. The players may only beat the dealer if they hold a score of 16 or higher. After the dealer has stood and is not bust, at a score of $s < 21$, they make a call of "pay $s + 1$". Any players holding hands scored at $s + 1$ or greater must reveal their hands and claim payment. A Royal Pontoon is unbeatable.

A sample game which illustrates most of the gameplay and the basic and extended rules can be found in Appendix E.

2.2 Reinforcement Learning

2.2.1 Introduction

Reinforcement learning is an unsupervised Artificial Intelligence learning method in which the agent teaches itself. The central principle is based on rewards for a sequence of actions taken or sequence of state transitions. Given a state *S*, the agent will take an action *A*, based on its knowledge of the environment. This will lead to a change in the environment, and eventually a reward, either positive or negative, will be given. The agent will then take this into consideration and accordingly adjust its knowledge. It must be noted that rewards are not given after every action, but may be after a series of actions.

The agent has a preference to each possible action for a given state. This is based on what is known as the action's value. This is a numeric value attached to each action *A*, given a state *S*. Every reward will adjust the action's value and hence make it more or less desirable. There are several ways to value action and the choice is mainly arbitrary. It depends mainly on the developers understanding of the system and their preference as to how the agent should behave.

Reinforcement Learning agents can act based on 3 things: values of states, values of actions or values of actions in states. The choice of which method to employ is left to the developer. This choice is made based on the developer's understanding of the environment and his ability to represent it accurately. It must be noted that an agent can learn in more than one way from the same environment. Action valuing is only applicable in very trivial tasks and hence is not used as much as state or state-action valuing.

The agent learns based on 2 things, namely the current reward and the previous rewards for *A*, *S*, or *A* & *S*. To simplify calculations and storage, we simply take the average of the previous rewards and update it. As the rewards are arbitrary, although mostly people use +1, -1 and 0, the agent will not know how good a reward is. By comparing it to the average of the past rewards, it can judge how high or low the reward was. A "high" reward would increase the value of the action and a "low" reward would reduce it.

How exactly rewards are allocated is a bit of a mathematical quandary. Some methods allow rewards to be "passed back" to previous actions. This follows directly from the fact that rewards are allocated after a series of actions. This is so a low valued action which can be followed by a high valued action seems more attractive. If this were not done then the action values would not be balanced and would only reflect immediate rewards.

2.2.2 Rewards

The basic value method is based on discounted expected future returns. As mentioned above, when an agent receives a reward, it passes back some of that reward. However this is not always back all the way and may just be to the previous state. By doing this, when the agent next encounters the newly rewarded state, it passes back to the previous state and so on and so forth. Based on these values, and the probability of the states occurring, we calculate the expected reward. So in effect we are summing the rewards of the given possible states over their probability distribution, from $0 \to \infty$.

This is not always possible to do, due the large problem space. Instead whenever a reward is received, all previous states or state-action pairs are updated by a fraction of that reward. This fraction is a constant call the discount rate and is denoted by γ. It follows that a state *S1* n steps from state *S2*, which receives a reward of r, will get a reward of $\gamma^n \times r$, where $0 < \gamma < 1$.

It is very important, especially for stochastic tasks, to minimise the effect noise will have on the learning algorithm. To do this, we use what is call the step-size parameter, denoted by α. This means that only a small portion of any reward will be passed back, mitigating any effects of noise in the data. valid data will be observed several times, which will eventually add up to its true value. The value of the step-size parameter is $0 < \alpha < 1$.

2.2.3 Policies

By assigning rewards to states or state-action pairs, the agent learns what is known as a policy, denoted by *P*. The policy is the set of actions the agent will take in a given state. The aim of Reinforcement Learning is to learn a policy which is optimal or as close to the optimal policy as possible, denoted by P^*. For some problems, such as the grid world, the optimal policy is defined. However for stochastic and/or non-stationary problems, this is much harder and some other statistic must be used to determine if the agent has learnt the policy it was required to.

Most Reinforcement Learning techniques have been mathematically proven to converge on P^* for the limit $n \to \infty$, where n is the number of runs. However as it is not always possible to learn the optimal policy in a reasonable amount of time, so the focus on is to learn a policy that is as close as possible to P^*, of not exactly P^*.

Reinforcement learning is a very powerful Artificial Intelligence tool. This very closely mimics the way humans learn by association. For every set of actions taken, there is a reward and this is associated to those actions, relative to past experience. However not every action is rewarded independently, but instead the entire series of actions receives a reward, which is mathematically calculated based on the whole reward. Hence it is not as powerful as human association, but it is none the less, very close to it.

What makes it even more powerful is that it is an unsupervised technique. So the agent can run through thousands of plays on its own and learn some policy *P*. With supervised learning, the user would be required to tell the agent something after every play. This way it saves time and completely eliminates any human error in the learning process. Assuming the implementation is done correctly, as stated above, it will, in the limit $n \to \infty$, converge on P^*.

2.2.4 Agent Behaviour

Agents can behave in several ways. The primary question with Reinforcement Learning is ”exploitation vs. exploration.” An agent can either exploit what it believes to be the best action, or it can explore new options, or ideally it can do some of both. The real trick is striking the correct balance between the two factors. Exploration is best for learning, as the agent will cover as much of the state space as possible, whereas exploitation is very useful after the agent ahs learnt a good policy.

The simplest exploitative and most intuitive behaviour is called a greedy agent. This agent will always choose the action with the highest reward value, known as exploitation. In this way it is guaranteed large immediate rewards, but possibly lower rewards in the long run. I believe given a small number of actions needed to reach a goal, this is the best approach. But it must still be said that some lowly valued actions may lead to much higher rewards than greedy actions.

In slight contrast to a greedy agent, we have what is known as an ε-greedy agent, which is more explorative. This agent explores other actions with lower reward values with a small probability ε. All non-greedy actions have an equal chance of being chosen. This allows the agent to explore more alternatives and possibly find a better action than the current greedy action. However in my opinion, this is not a very balanced way of having the agent behave as it may take a very poor action as likely as the next best action.

A more explorative agent behaviour is called a softmax selection agent. It takes all actions with a probability proportionate to their value. It creates a probability distribution according to the formula below:

$$\frac{e^{Q_t(a)\backslash\tau}}{\sum_{b=1}^{n} e^{Q_t(a)\backslash\tau}}$$

(Sutton and Barto, 1998)

This ensures that the agent picks better actions more often than not. This is, in my opinion, very close to human behaviour and can be perceived as the agent being able to take risks, as it will not always take the greediest action, but it tends to take good actions rather than bad. Also this strikes a fine balance between exploitation and exploration. The parameter τ is called the temperature and decides how much the agent explores. The values of the temperature are bounded as $0 < \tau < 1$

> High temperatures cause the actions to be all (nearly) equiprobable. Low temperatures cause a greater difference in selection probability for actions that differ in their value estimates. In the limit as $\tau \rightarrow 0$, softmax action selection becomes the same as greedy action selection. (Sutton and Barto, 1998)

Most agents start off with initial values all set to 0. However based on human knowledge of the system, optimistic initial values can be set. As they systems interacts with its environment, it will update the values based on formula defined by the user. This gives the agent a statistical bias. However Sutton and Barto (1998) make the point that it provides “an easy way to supply some prior knowledge about what level of rewards can be expected.”

2.3 Blackjack in Reinforcement Learning

Taking the existing Card/Pack system, which works very well, I intend to rebuild the functionality of the Player and Main classes. I will re-implement the basic GUI I had before, but the Player class will now be redefined. Each game will have 2 types of players, a human player, an improved version of the previous system, and a reinforcement learning agent. This will be the main concentration of my project.

Blackjack is a fairly simple game with minimal mathematical complexity both in terms of rules and betting. Also a relatively small number of actions are required to reach and outcome and hence a reward. Both these factors, make Blackjack suitable as a reinforcement learning problem.

After having made a successful agent for blackjack, it should be possible to abstract the basic functionality of the agent and hence create AIPlayer, which would integrate with my old framework and make an infinitely extensible suite of card games, with AI playing capabilities.

Blackjack is what is known as a Markov Decision Process (MDP). Blackjack is said to be have the Markov property because, given any state, you can easily trace back previous states, as well as the actions leading to those states. This is the case with a large majority of Reinforcement Learning problems.

The basic state representation will be the sum of the value of the cards the player has. This is the most important information, as far as the game is concerned. It would be wasteful to have a separate state for each combination, as they are irrelevant, a 6 and 7 is the same is a 5 and 8.

However there are some complexities involved in this. The most prominent case is the fact that an Ace can count high (11) or low (1). Also deriving from this is the fact that when a high ace would cause a player to be bust, it is automatically switched to a low ace. Another complexity would be enabling the agent to split. Considering the state is only the sum, we can miss on a split, as a 5 and 5 is treated the same as a 2 and 8. Also the double down rule would be a minor complication, but this would not be due to the state representation, but because of the inherent complexity of the rule itself.

Chapter 3

Project Description

3.1 Technologies involved

The system will be developed using the Java programming language, as the existing system is written in Java. From this fact, it follows that any additions to the system must be in Java. To change to any other programming language would require rebuilding the entire system, after it has been refactored. Due to the time constraints, it was decided to continue using Java.

In terms of Reinforcement Learning, the algorithms provided in Sutton and Barto (1998) are all in pseudo-code. Hence we can take these algorithms and implement them in any given programming language. As mentioned above, the existing system was developed in Java, hence the learning algorithms will all be implemented in Java to be added to the system.

3.2 System Design

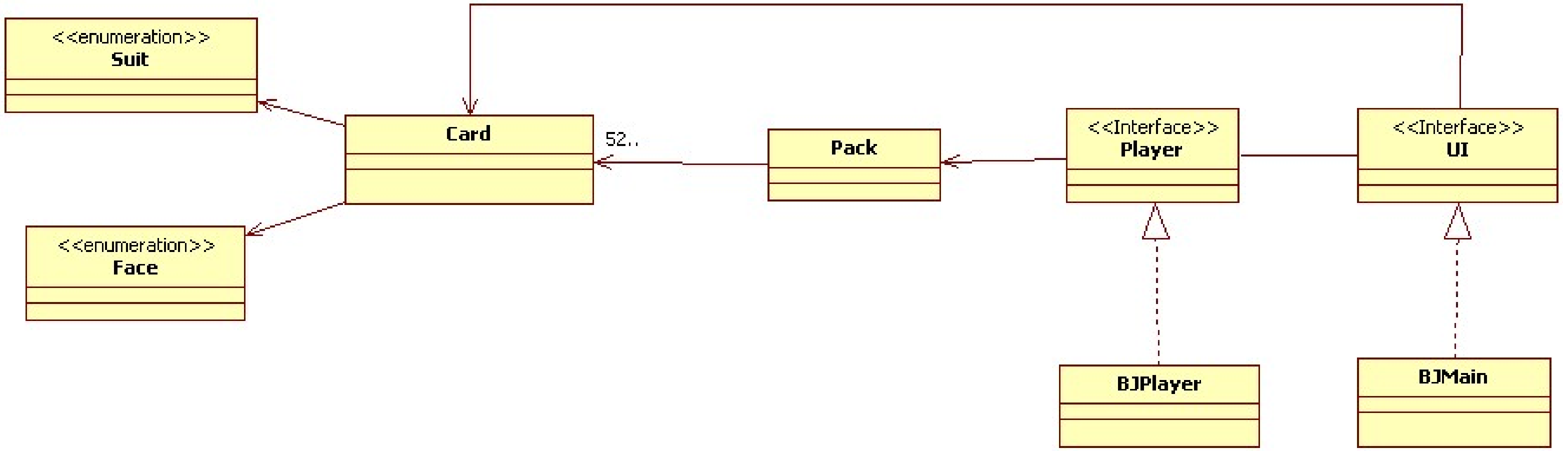
The old system is based on 3 main components. Firstly the Card/Pack system, which is currently the only fully operational part. This may still be modified to improve on the current functionality. The second part is Player interface, and its subclasses, which for the representation for the Player. Thirdly we have the Main interface, its subclasses and SKMain which provide the GUI elements. SKMain is the Main Menu of the program, but does not implement Main. This can be illustrated in the class diagram (see figure 3.1).

This system is not very stable, as the dependencies and interactions between Main and Player are very complicated and tend to cause unexpected errors. The new system uses a less densely connected system to ensure a more streamlined system and easier communication between objects. The interface "GameRules" will be introduced to act as a conduit for communication. This is illustrated in the class diagram (see figure 3.2).

There are two basic Use Cases for the system, which a Player playing the game. This case is implemented in the old system. However this case is specific for a Human Player. The second Use Case is the Reinforcement Learning agent actually learning how to play the game. This case actually includes the first Use Case. This can be illustrated by the Use Case Diagram (see figure 3.3).

Based on the reading and research, it has become apparent that Blackjack is an episodic task. The agent will learnt something each time it plays an episode, which in this case is a game of Blackjack. For Blackjack, there can be no more than 2 states between the initial state and the final state. This is because a Player may not have more than 5 cards in his hand. The initial state is 2 cards. Between these states we have 3 and 4 cards as the two possible intermediate states. The agent will operate as illustrated in the sequence diagram (see figure 3.4).

However it must be noted that in some problems, there may be a reward at an intermediate state. This is not true for Blackjack, hence this was not included in the diagram.



1

Figure 3.1: Old System Architecture

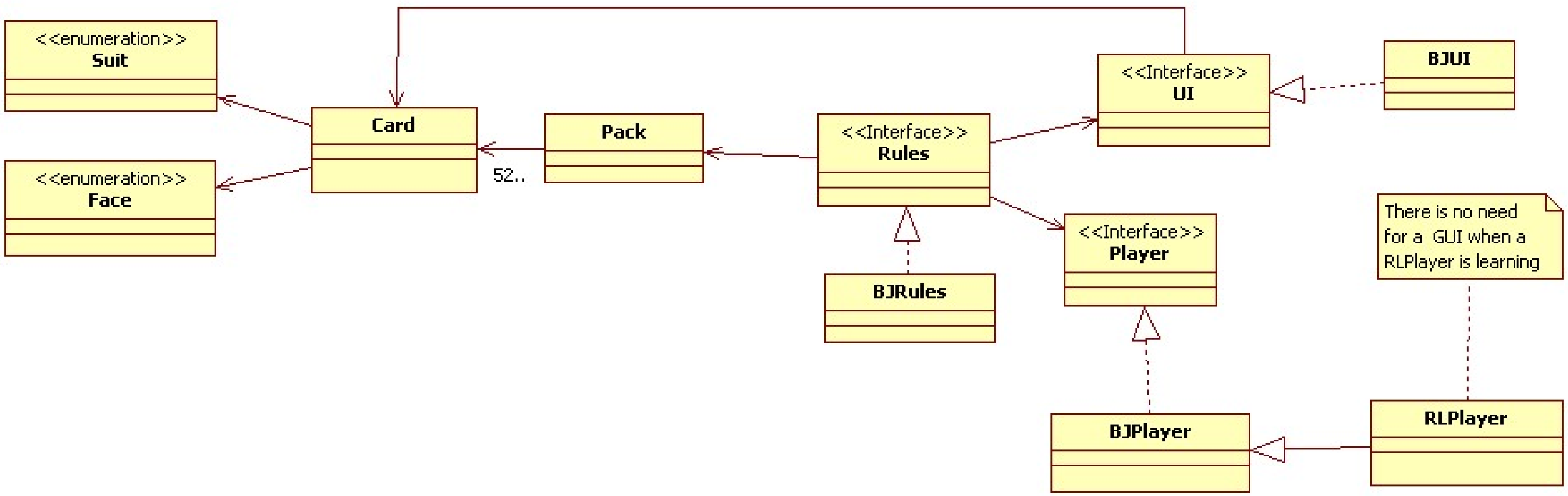


Figure 3.2: New System Architecture

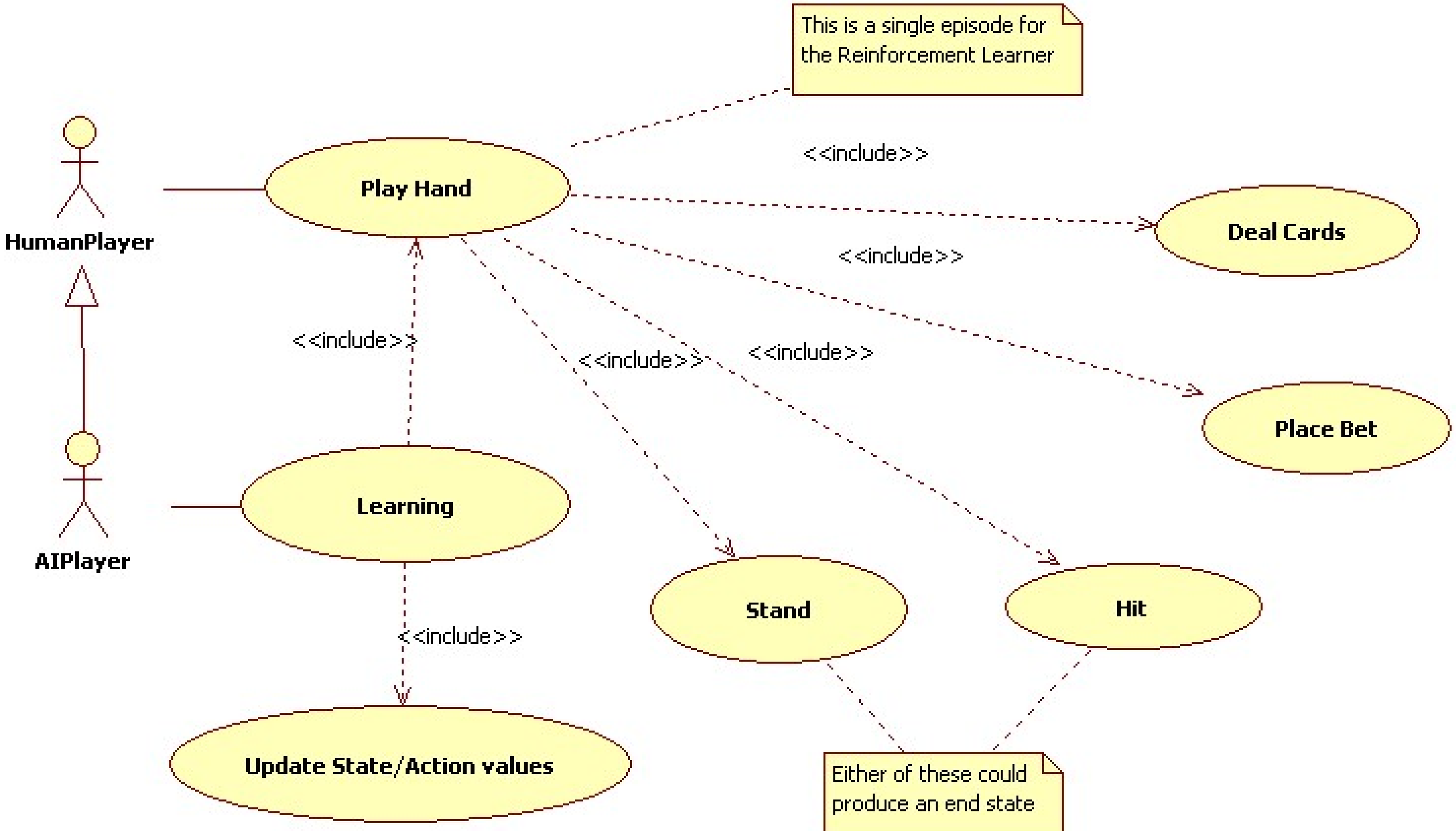


Figure 3.3: Use Cases

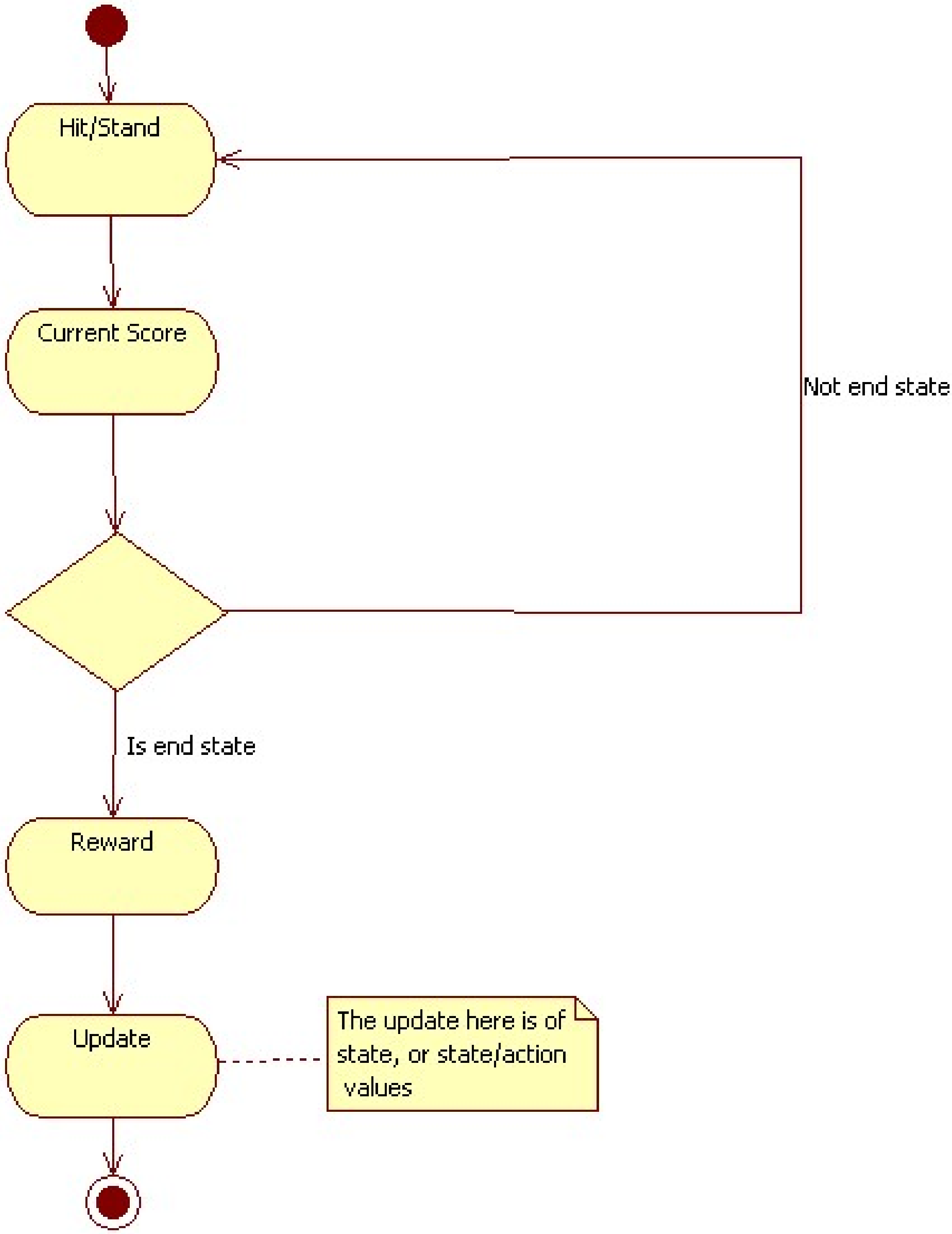


Figure 3.4: Single Episode

Chapter 4

System Development

4.1 Prototyping

To better understand Reinforcement Learning, two prototypes were built before the final system was built. These were built chosen in such a way that they shared some features with the final system, thus enabling code reuse. Both of these prototypes are discussed below.

4.1.1 Prototype 1 - Next card higher/lower

This system trains an agent to guess if the next Card is higher or lower than the current card. This was chosen so as to see how the Card/Pack system would be used in an agent. No proper Reinforcement Learning algorithm was used, but merely a sum of all past rewards.

The agent used the points() method in the Card class to determine of a card was higher or lower. This gives a Joker 0 points, Ace 1 point, number cards face value and Jack, Queen and King get 11, 12, and 13 respectively. The agent received a reward of +1 for guessing correctly, and -1 for guessing incorrectly. The source code for this prototype can be found in Appendix A.1.

The prototype did not take into account when two cards where equal and just counted this as an incorrect guess. This situation does not occur very often and was not considered for the prototype as it was simply a proof of concept as opposed to a proper system. Results from this prototype can be found in Appendix A.2.

This prototype learns a policy which is fairly consistent with what is expected. However as this does not use a proper Reinforcement Learning method, it is not guaranteed to converge on P^*. Despite this, it uses the softmax probability as described in section 2.2.3.

From this prototype, code that calculated softmax probabilities was made. This was carried over to the next prototype and the final system. Also it showed how to correctly use the Card/Pack system in a Reinforcement Learning Agent. Parts of this code was carried over to the final system.

4.1.2 Prototype 2 - GridWorld

This prototype solves what is known as the GridWorld problem. This problem has an agent in a grid-based environment. Each square is either a goal, a penalty or empty. The agent begins in a random square and must navigate its way to the goal(s) and avoid any penalty(ies). If the agent moves to a penalty or goal square, the episode ends. It receives a reward of +100 for a goal and -100 for a penalty. The source code for this prototype can be found in Appendix B.1.

For this prototype, the Monte Carlo method was used (see Sutton and Barto (1998) chapter 5). This is mathematically proven to converge on P^*, which this system does. For an easier graphical representation, states were valued instead of state/action pairs. As stated before either approach is valid and can be chosen based on the users requirements. Some results of this can be found in Appendix B.2.

From the previous prototype, code that calculated softmax probabilities was used. This prototype created the code for reward functions and value calculation. The code was modified, as the Monte Carlo method was abandoned in favour of Q-Learning for the final system. However, the learning parameters were carried over.

4.2 Final System

4.2.1 AI

As the Reinforcement Learning Agent was the primary objective of the project, it was built first. It was decided that the agent should be built using the Temporal Difference Q-Learning Algorithm. Although Sutton and Barto (1998) cover Blackjack under the MonteCarlo methods, Temporal Difference is a more powerful method in general.

For this agent, only hit and stand were considered as valid actions. The rest of the actions (as listed in 2.1.2) were not considered for the system, as it was thought that the agent would not learn anything of substance. These reasons are enumerated below:

- **BLACKJACK:** This would not help the agent learn anything except that a starting pair adding up to 21 is greatly desired. Hence the logical action would be to stand. The system will already learn it, as it will always get a negative reward for hitting at 21. Although the increased payout was added into the system, however the agent does not learn anything from this.
- **INSURANCE:** Taking out an insurance would not be beneficial in anyway to an AI agents policy, as the payout of this is based on probability. Also this occurs so infrequently, that it was not considered worth adding in.
- **SPLIT:** Splitting hands would only give the agent more hands to learn on. Although it would lead to more learning, it would not reduce the learning time enough to justify its inclusion. Also due to the complexity of being able to split many times it was not included in the final system.
- **DOUBLE DOWN:** Doubling down would lead to a hit and then stand. The agent would effectively learn a policy for hitting at a score s_n and standing at score s_{n+1}. This is would not lead to any additional learning and may also impede the learning of the agent. As a separate action, it would not add anything useful, as it is just a hit and a stand.

For further clarification, a graph indicating how frequently each action may have been taken is included below. As can be seen, all the actions occur less than 10% of the time. This would not have lead to any significant learning. Although double down occurs over 70% of the time, as stated above it would not add anything meaningful to the learning.

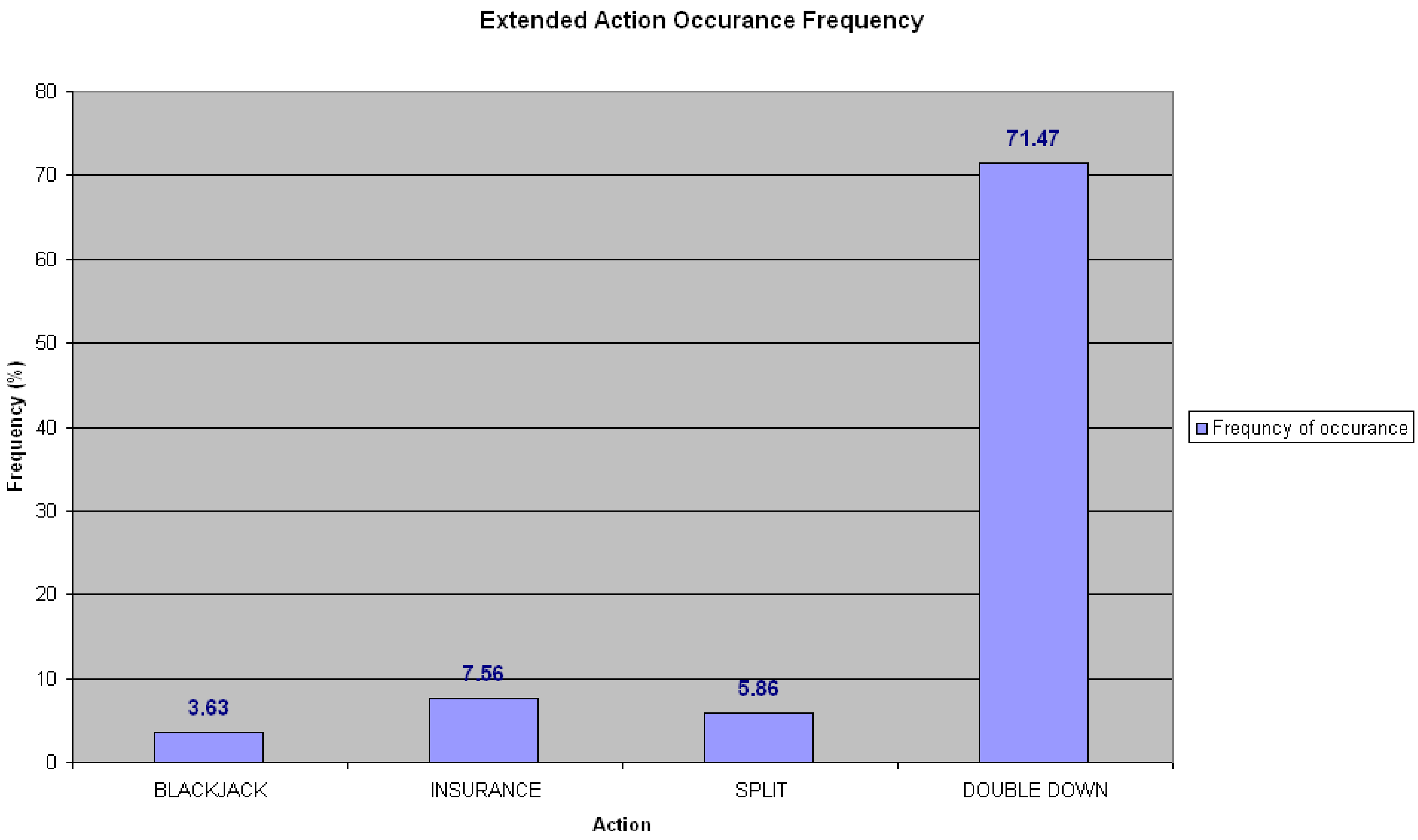


Figure 4.1: The frequency of the extended actions

```
double hit = Math.exp(val[HIT][score -1]/TAU);
double stand = Math.exp(val[STAND][score -1]/TAU);
double sum = hit + stand;
double hprob = hit / sum;
double sprob = stand / sum;
```

The first step was to build the graphical user interface. This was built with the same design as the previous system. This was to ensure that the maximum of code could be re-used in the new system, so as to avoid any complications of porting the code from one interface to another.

After the interface was built, the underlying game mechanics were built. For the AI, only hit and stand were considered for actions. The reasons for this have already been discussed above. To facilitate this, a class called BPlayer was created as an inner class, to facilitate easy access. This was used instead of using accessor methods to manipulate the object for simplicity and quick implementation.

The BPlayer was created with the basic functionality required for hit and stand. The BPlayer had no real functionality, apart from the Reinforcement Learning, as most of the gameplay is done in the Blackjack object. After this was established, the Reinforcement Learning methods were added. The player has an exploration method, which uses softmax, a greedy method and a fixed method, for the dealer.

Finally the Reinforcement Learning agent was developed. This mostly used code from the prototype, with modifications to fit into the system. The completion of the Reinforcement Learning agent makes both system equal in functionality. Although it must be noted that new AI works, where as the old AI does not.

To achieve this, we need a state representation. For Blackjack, this is simple as it just the score of the player minus 1. The actions were stored as numbers. The values of the state action pairs were stored in a 2-dimensional array called val. The value for action *A* in state (score) *S* is given by val[*A*][*S-1*]. The states are represented as *S-1* as Java uses 0 as the first array index and $n-1$ as the n^{th} index, hence the *S-1*.

The softmax method was fairly simple to implement as there are only 2 possible actions, hence only two probabilities. These were calculated using the following code:

After these probabilities are calculated, we update the last state-action pair, if any. For the case where it is the first action, *lastaxn* is set to DEAL(-1). Then a random r number is generated to choose the action. After the action is chosen, we set the values of *lastaxn* and *lastscore*. This is achieved with the code given below:

The above code is an implementation of the general formula:

$$Q_{(s_t,a_t)} = Q_{(s_t,a_t)} + \alpha(r_{t+1} + \gamma max_a Q_{(s_{t+1},a)} - Q_{(s_t,a_t)})$$

(Sutton and Barto, 1998)
Here the reward r_{t+1} is 0, as no reward is given.

After the learning method, a greedy method was devised. This is for use when the agent has learnt a policy $P \approx P^*$. As stated previously, after exploring, the agent should exploit what it learnt. This was implemented using the following code:

```
double maxval = Math.max(val[STAND][score - 1], val[HIT][score - 1]);
if(lastaxn != DEAL){
    double rd = A * ((G * maxval) - val[lastaxn][lastscore - 1]);
    val[lastaxn][lastscore -1] += rd;
}
double r = Math.random();
lastscore = score;
if(r > hprob){
    stand();
    lastaxn = STAND;
}
else{
    hit();
    lastaxn = HIT;
}
```

```
if(val[HIT][score -1] > val[STAND][score - 1]){
    hit();
}
else{
    stand();
}
```

Finally the reward function was created. This gives a reward to the agent, based on the game outcome. After the dealer stands, the outcome is calculated. There are 5 possible outcomes. Based on the outcome, the player was judged to have achieved one of the 4 results and based on that it was given a reward. Listed below are the 5 outcomes and their end state and reward in brackets:

1. Player Busts (Bust -10)
2. Dealer Busts (Win +10)
3. Player and Dealer bust (Bust -10)
4. Player Wins (Win -10)
5. Player Loses (Lose -10)
6. Push (Push +5)

This was achieved using the code below:

```
if(player.score > 21 && dealer.score <= 21){
    //player bust
    player.reward(-20);
}
else if(player.score <= 21 && dealer.score > 21){
    //dealer bust
    player.funds += bet * 2;
    player.reward(10);
}
else if(player.score > 21 && dealer.score > 21){
    //dealer and player bust
    player.reward(-10);
```

```
}
else if(player.score > dealer.score){
    //player wins
    player.funds += bet * 2;;
    player.reward(10);
}
else if(player.score < dealer.score){
    //player loses
    player.reward(-10);
}
else{
    //push
    player.funds += bet;
    player.reward(5);
}
```

The reward method uses the same formula as employed by the learning method, except the reward is explicitly passed to it as can be seen above. The code for this method is given below:

```
private void reward(double r){
    double maxval = Math.max(val[STAND][lastscore -1], val[HIT][lastscore -1]);
    if(lastaxn != DEAL){
        double rd = A * (r + (G * maxval) - val[lastaxn][lastscore -1]);
        val[lastaxn][lastscore -1] += rd;
    }
}
```

It must be noted that the final class has more code than listed above, but this code was for testing purposes only and does not interfere with the learning algorithm. For the final code see Appendix C.1.5.

4.2.2 SKCards

The first feature that was added to the SKCards was the Shoe class (see Appendix D.1.5). This class is used to represent what is known as a dealer's shoe. This is in effect a Pack, which is not fully used. A cut card is inserted at random in the Pack. Once this cut card is dealt, the Shoe is discarded and a new one is brought into play. This is used mainly in casinos for Blackjack to counter card counting. This was implemented for completeness of the system. The dealer does not distinguish between "soft" and "hard" scores. This is as dealer play was not the objective of the experiments.

Then the Player interface was reworked, to fit in the new system. This included eliminating all relations to the Main interface, which was renamed to UI. Simultaneously, the UI interface was modified to eliminate all relations to the Player interface. The Rules abstract class was then created and the two interfaces where joined using this. With this the framework was completed.

After completing the framework, the existing BJPlayer and BJUI (formerly BJMain) were built up. Where possible, code from the original system was used, in its original form or modified. Where it was not possible, code from the prototype was used, or it was made from scratch. This now makes the Old system, equivalent to the old system in functionality, barring the AI.

Chapter 5

System Evaluation

5.1 System testing

There are 4 basic tests that have been devised, all of which depend on the agent being able to react to a dealer policy effectively. As previously stated, we wish to come as close as possible to P^*, which in this case would be a policy better than its opponent (the dealer).

Each test uses different values of rewards to achieve a different purpose. The first test uses the normal values as listed below:

- Win = +10
- Push = +5
- Lose = -10
- Bust = -10

The subsequent tests involve doubling the reward for win, lose and bust and keeping the rest at their normal values. The push value was not modified as its low frequency of occurrence would mean its effect is minimal. These values were tested against 7 dealer policies, namely dealer hits below 11-17. As the dealer plays a more aggressive strategy, the agent should also adjust its policy, within the parameters provided.

This range of policies is that the two boundary values represent significant policies. The lower bound (11) represent the policy a player must play to avoid going bust. The upper bound (17) represents the policy that is employed by several casinos, making it the de facto standard. All other policies were included for completion.

The constants in the learning equation were kept the same as listed below:

- Softmax temperature $\tau = 0.5$
- Step size parameter $\alpha = 0.6$
- Discount rate $\gamma = 0.75$

5.2 Results

5.2.1 Table

This table provides the results of running the agent with different parameters. The data and therefore the results are highly susceptible to noise, due to the random element in the system. However separate data sets with the same parameters tend show a consistency of approximately $\pm 2.5\%$, giving a total error range of 5%. This error can be attributed to noise in the data, as well as the random factor within the game. Due to the nature of card games, even an optimal action can lead to a sub-optimal result, causing interference in the data. This is mitigated to a certain extent by the Law of Large numbers.

DEALER POLICY	WIN REWARD	LOSE REWARD	BUST REWARD	WIN(%)	LOSE(%)	BUST(%)	NET WINS(%)
11	10	-10	-10	44.62	17.6	30.66	-3.64
12	10	-10	-10	43.38	19.88	29	-5.5
13	10	-10	-10	37.58	21.98	29.06	-8.96
14	10	-10	-10	34.76	22.28	29.34	-10.76
15	10	-10	-10	27.86	23.08	30.22	-12.46
16	10	-10	-10	26.2	23.9	29.26	-11.74
17	10	-10	-10	19.08	17.4	38.74	-12.86
11	20	-10	-10	45.98	23.1	22.02	0.86
12	20	-10	-10	42.76	19.54	29.82	-6.6
13	20	-10	-10	37.62	20.96	29.12	-7.92
14	20	-10	-10	35.02	21.92	29.66	-11.04
15	20	-10	-10	28.14	24.06	28.72	-12.26
16	20	-10	-10	26.12	23.88	29.88	-12.82
17	20	-10	-10	17.92	25.46	29.54	-13.18
11	10	-10	-20	43.02	47.22	0	-4.2
12	10	-10	-20	37.1	52.5	0	-15.4
13	10	-10	-20	30.06	56.22	0	-22.02
14	10	-10	-20	27.04	57.42	0	-24.4
15	10	-10	-20	19.48	58.88	0	-26.26
16	10	-10	-20	18.3	58.26	0	-25.72
17	10	-10	-20	12.44	54.34	0	-16.76
11	10	-20	-10	29.26	3.92	60.08	-34.74
12	10	-20	-10	27.52	5.48	59.56	-37.52
13	10	-20	-10	25.28	5.02	59.86	-35.44
14	10	-20	-10	25.28	5.02	59.32	-32.9
15	10	-20	-10	20.66	5.86	60.84	-32.28
16	10	-20	-10	20.46	5.8	59.86	-31.12
17	10	-20	-10	16.14	6.04	59.02	-25.7

Table 5.1: Final Results

5.2.2 Graphs

All the data from table 5.1 is illustrated graphically in the graphs below. Each graphs show the comparison between the 4 separate testbeds. For each run the following values were calculated:

1. Final Policy
2. Winning Percentage
3. Losing Percentage
4. Bust Percentage
5. Net Winning Percentage

The final policy is the highest score at which a greedy agent will hit and will stand on any higher score, based on the values learnt. It is expected that as the dealer policy increases, the agent will also increase its policy to try and stay ahead of it. It is also expected as agent's reward values are adjusted, its policy will also be adjusted.

The winning percentage is the percentage of games in which neither dealer nor agent has gone bust and the player has won the game. Although when the dealer goes bust, it is considered that the agent has won, this was not considered as this situation arises from poor play from the dealer and not necessarily good play from the agent.

Similarly, the losing percentage is the percentage of the games in which the dealer has won. Again the case of the player going bust was not considered under here as it can be attributed to poor play by the agent and not necessarily good play from the dealer.

However, busting is an essential part of the game and must not be ignored. To this effect, the percentage of busts was recorded. This number is simply the percentage of times the player has been bust during the course of the learning.

When considering all of these figures, we must remember that there is a large random factor that affects the game and therefore the results. Although the step-size parameter can smooth the learning process, there is really no way to compensate for the noise in the final results data.

To consolidate all these figures, the net winning percentage was also calculated. This can be found using the formula below:

```
net wins = (wins + dealer busts) - (losses + player bust)
```

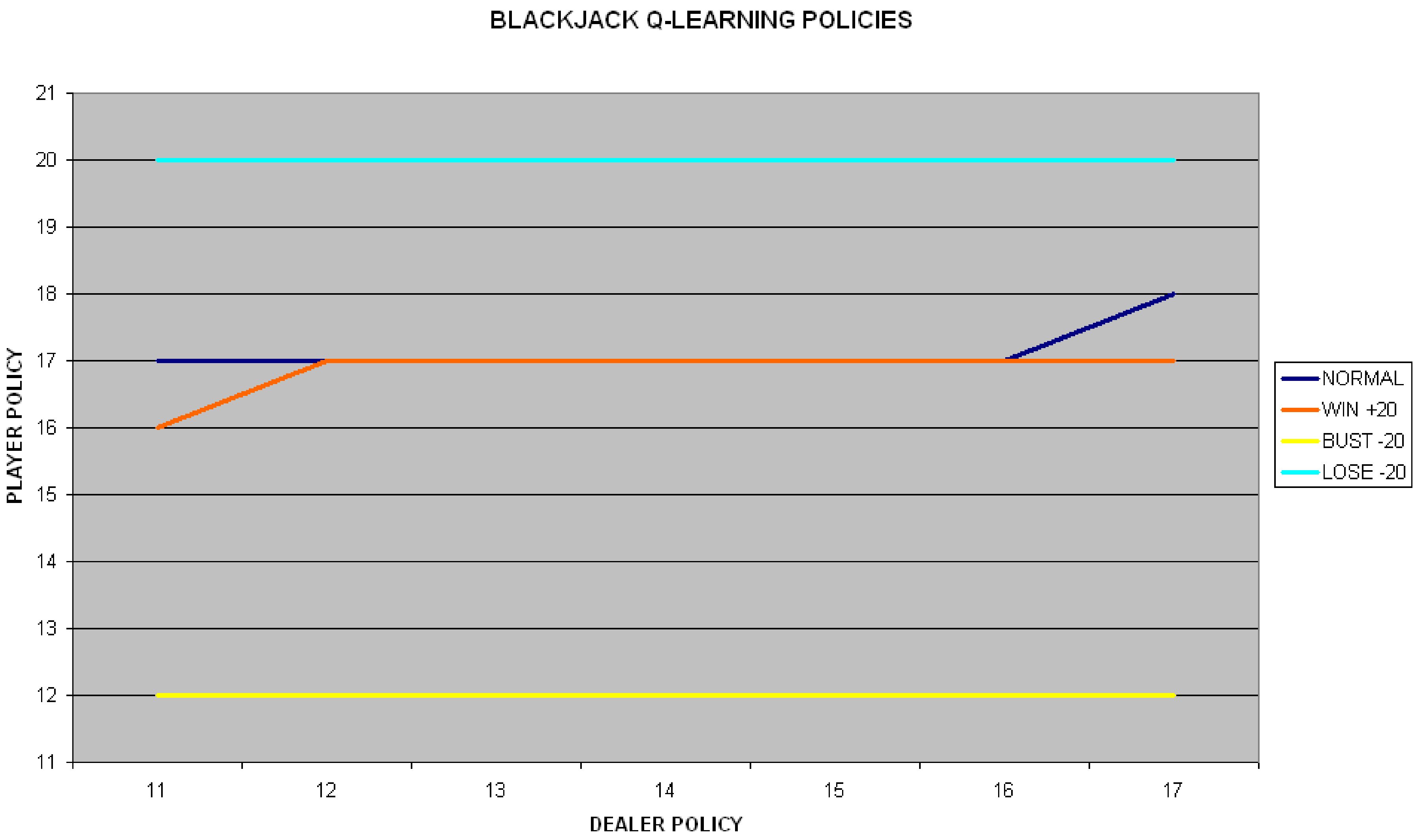


Figure 5.1: Q-Learing Policies

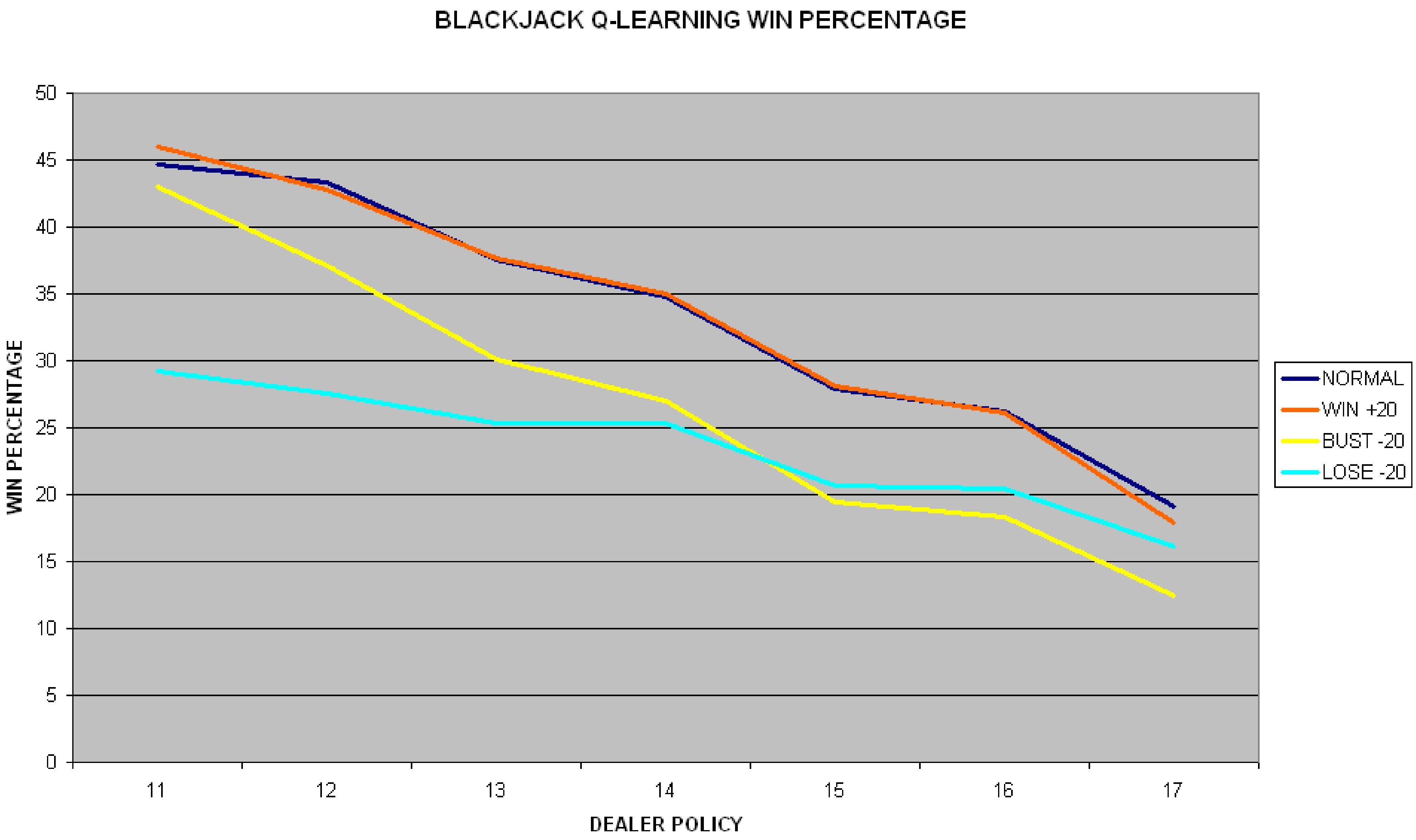


Figure 5.2: Win Percentages

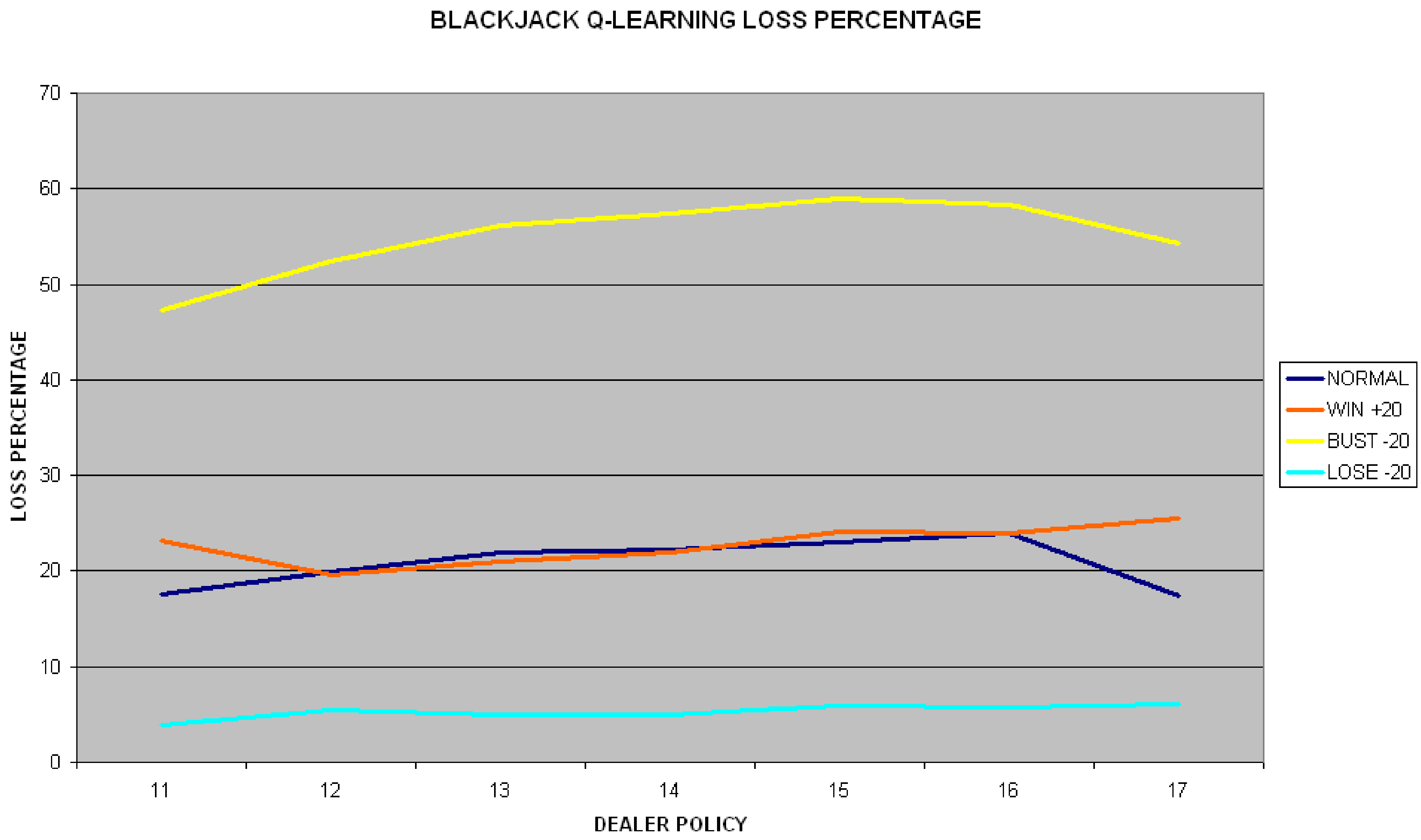


Figure 5.3: Loss Percantages

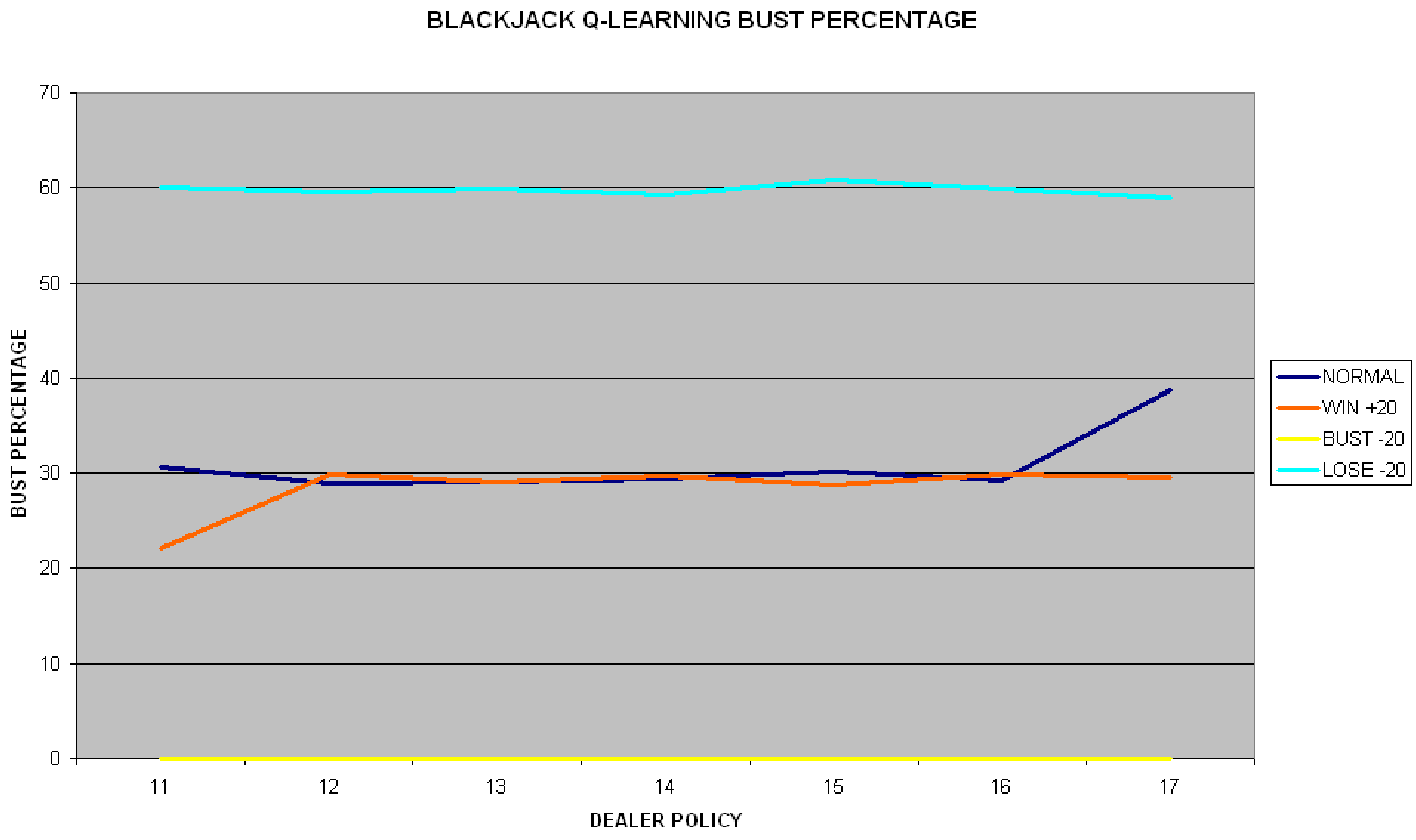


Figure 5.4: Bust Percantages

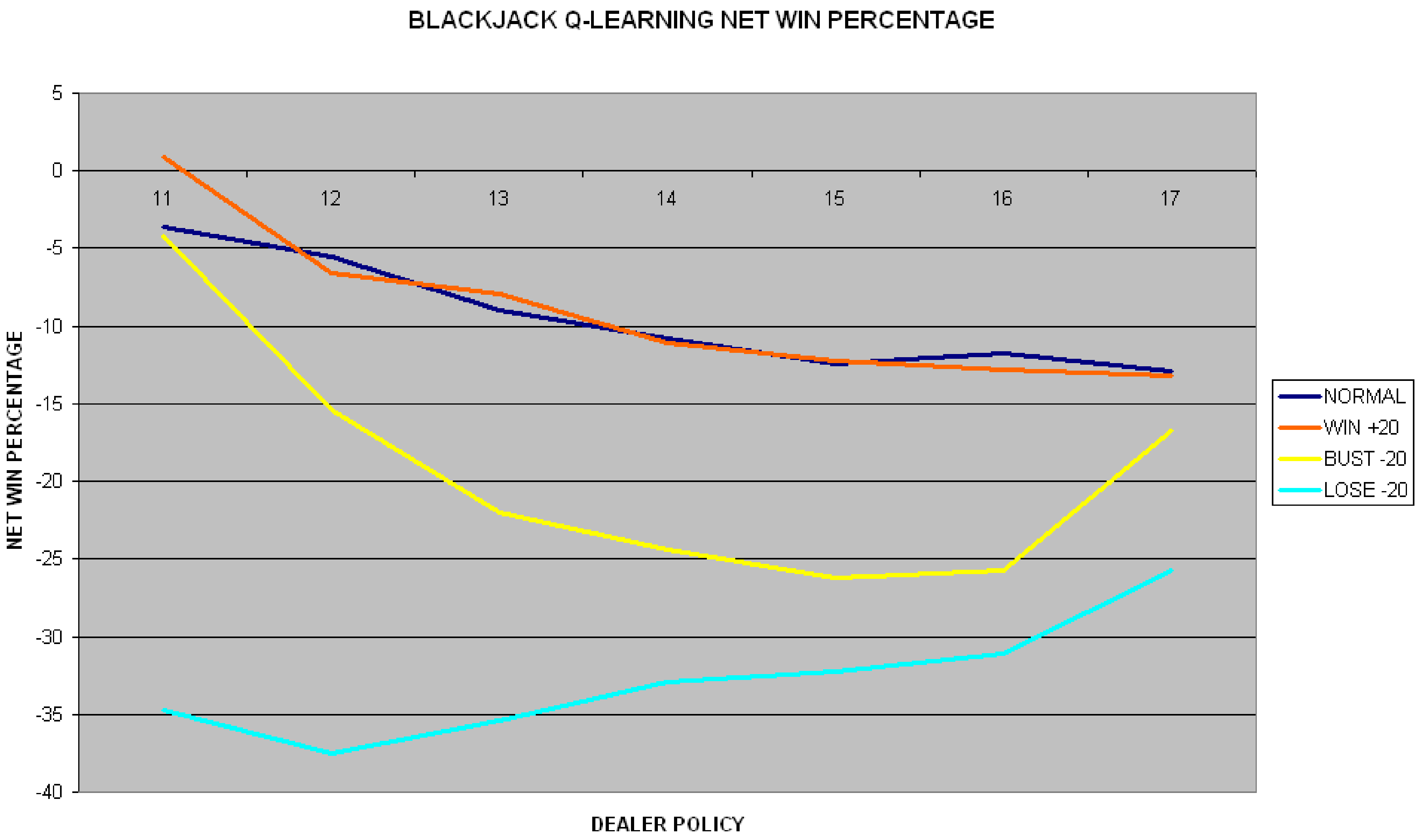


Figure 5.5: Net Win Percentages

5.3 Result Analysis

As can be seen from Figure 5.1, when the agent is given a reward of -20 for busting, it learns the most effective policy to counter it, that is hitting at 11 or below. Here we can see the agent prefers to lose (-10), rather than go bust. The best possible reward is +10 for it winning. However, it chooses not to pursue this, as it is simply tries to avoid going bust and receiving the -20 reward. However, when it is given a reward of -20 for losing, it plays a very aggressive strategy, specifically 20, hitting on everything but a 21. Here the agent tries to avoid losing, even at the risk of busting. The best way to do that is play a very aggressive strategy, which it does. By increasing the win reward to +20, the agent plays a slightly more defensive strategy. However, on several points, both agents pick the same strategy. Cleary, the negative rewards have a greater bearing on the learning than the positive rewards. Hence any tuning of the agent would concentrate more on the negative rewards than the positive.

One point that must be noted is that the agent consistently learns a threshold score at which to stop hitting. This is not specified on the algorithm. The algorithm merely provides the agent the ability to value the actions and hence build a policy based on its experience. This can be said to be one of the major successes of this project, in terms of the Reinforcement Learning Agent.

Furthermore, from Figure 5.2 it can clearly be seen that as the dealer plays a more aggressive strategy, the player will lose more often. This is as expected, because as the dealer plays a more aggressive policy, it will score higher, hence making it harder for the player to win. A point worth noting is that having a higher bust and lose penalty results in a lower winning percentage, where as increasing the win reward has a minimal effect.

As we proceed to Figure 5.3, we can see that this graph shows trends opposite to Figure 5.1. This is expected as the better policy the agent plays, it should lose less. The lines are not as smooth, due to the noise in the data, but the basic trend can be seen. Also it must be noted that the dealer policy has a minimal effect on the loss percentage of the agent.

Figure 5.4 shows the same trend we saw in Figure 5.1. The more aggressive policy the agent plays, the more it will bust. As it chooses to hit on higher scores, there is a larger probability of it going bust. Hence, the same trend repeats itself, as expected. However, it must be noted that the agent busts more than the dealer. The reason for this is not known, but it has been observed that the player busts about 5% more than the dealer playign the same policy.

From the above graphs and analysis it is clear that the best results are obtained with the winning set at +10 or +20. Changing the penalties may improve the performance in certain areas, but the overall performance is degraded. Hence the best agent under these circumstances would have a win reward of +10 or +20.

5.4 Possible Future Improvements

As can be seen from Figure 5.5 the agent tends to lose more than it wins when playing a policy of 12 or greater, no matter what the learning parameters. As this is the primary performance benchmark, any non-trivial improvement to the system would have to increase this figure. There are two possible ways this can be done, card counting and probability calculations. Both of these are discussed below.

5.4.1 Card Counting

Card counting is a tactic used by some players to try and “beat the system”. This tactic involves keeping a running total of what is know as the “count” of the deck. Each low card “counts” as +1 and each high card “counts” as +1. By adding the “count” of all the cards observed so far, the player can tell how “heavy” or “light” a deck is, indicating a large number of high or low cards respectively. Based on this, the player can adjust their strategy.

The best way to combat this is to use a Shoe, which has been implemented (see Appendix D.1.5). Once the shoe is changed out, the player no longer has a count of the deck. This would mean that they must now begin to count again. While they are counting, they would presumably play a fixed strategy. By keeping the cut-off point as low as possible, card counting can be countered.

5.4.2 Probabilities

Another possible way to improve the system is by using probabilities. This assumes knowledge of how many decks are in play. Normally this player information is kept hidden from the player. However, if the number of the decks is known, the number of each card is known. Based on cards already played out, the agent can calculate the probability of getting the required card(s). Using these values, the agent can adjust its policy.

The simplest counter to this is to simply not reveal the number of decks. Although a well programmed agent can calculate several probabilities at the same time. It could simply maintain multiple variables and update each with based on Cards observed. The number of decks could be inferred from the frequency of Cards appearing, e.g. 3 occurrences of the King of Spades would indicate at least 3 decks. Again, if we use a Shoe instead of a Pack, we can also quite effectively counter this.

5.4.3 Busting

It has been noted that the agent seems to bust too much. There is no apparent reason for this. Based on analysis of the code in Appendix C.1.5, the system should perform equally to the dealer. However this is not the case. The software needs to be thoroughly investigated to determine the cause of this error. Once found the error needs to be rectified. This would improve the net win percentage of the system, which as stated above is the measure of improvement in this system.

5.4.4 SKCards

There is scope for improvement in the SKCards framework. The Blackjack only implemnts the core rules. The extended rules can be implemented for the player and make it a complete Blackjack game. Also, the GUIs are very functional but do not look very attractive. These could be made more attractive. There is also the possibility of adding more games to the framework.

Chapter 6

Summary and Conclusions

6.1 Summary of the Project

6.1.1 Goals

There were two main aims of this project. The main goal was to develop a Reinforcement Learning agent for Blackjack. The second goal was to refactor the SKCards system and incorporate the Reinforcement Learning agent into it.

6.1.2 Outcomes

As we can see, the Reinforcement Learning agent was successfully developed. Although it was unable to beat the dealer, this is due to the nature of the game and not any shortcomings in the learning agent. However, it must also be noted that the large number of busts lead it perform lower than expected.

The refactoring was successful. The new system was created, and conforms the the model shown in Figure 3.2. However due the large amount of time spent on trying to implement the AI, there was little time for the final system. Althought the system was built, it has only the most basic functionality and supports only the core rules as stated in Section 2.1.2. Although it was planned to have all the rules included in the system, this was not possible.

6.2 Conclusion

Based on the results gained from the system, it can be said that the Reinforcement Learning Agent learns an effective policy. Although the algorithm does not specify that the agent should learn a threshold score for hitting, the agent does learn this. This can be seen a large success of the learning algorithm. From figure 5.1 we can see that modifying the values of the bust and lose reward has a much greater effect than the win parameter. It can be concluded that the agent does learn to play Blackjack and that Reinforcement Learning is suitable for Blackjack.

Also we notice that as the dealer plays more aggressively, even though the player is playing an equal or better policy, they tend to lose more. It can be inferred from here that through pure strategy alone, the casino policy is very difficult to beat.

Also we can see there is some noise affecting the system, as can be seen in Figure 5.4. The graph is fairly consistent with what is expected as the number of busts relates to the policy the agent plays. Both figure 5.4 and 5.1 show the same basic trend. However in Figure 5.4 the last point for the agent with -20 reward for loss is significantly higher than the others. This is the noise and the random element of the game clouding the results.

Although the Law of Large Numbers mitigates this effect to a certain extent, it is not sufficient. The Pack is shuffled using the Math.random() function in Java, which is not very effective at generating very large numbers of data. This factor is not very significant and does not affect most of the tests. However, this, combined with noise and the random element in Blackjack, can explain erroneous results such as this.

Chapter 7

Project Self-Evaluation

7.1 Introduction

This project aimed to build a Reinforcement Learning agent for Blackjack and build it into an existing framework. The framework had to be refactored to make it more usable and conform with software engineering norms. It can be said that this was very successful to a large extent.

7.2 Project outcomes

The first outcome, which was to develop a Reinforcement Learning agent, was successful. The agent does learn a policy of hitting until a threshold score. This score varies on the rewards it is given. However it can be seen from Figure 5.1 that the agent does learn a suitable policy based on what it is told to do. There is however, the problem of the agent busting more than the dealer.

The second outcome was to refactor the old system and include the Reinforcement Learning agent. This was succesfully achieved. However, it was only built to support the core rules of Blackjack. This was due to a planing oversight on my part, which meant that I spent too much time on the AI development.

7.3 Project planning

The initial planning of my project was quite challenging. There were several tasks to be completed and these all needed to be scheduled. The 3 main tasks were:

- Background reading and research
- Prototyping
- System development

Although I had allocated time for all these tasks, the prototyping stage spilled over into the system development stage. This gave me less time to work on the system. Barring this, the project went according to plan. There was a maximum deviation from plan of 1 or 2 days. I now realise that I should have allowed a larger buffer time in between tasks for spill over and unforeseen errors

7.4 Self-evaluation

During the course of this project, I learnt the limitations of my programming skills. Although I consider my self highly competent in terms of programming, developing graphical user interfaces is an area where there is great room for improvement. My interfaces tend to lean towards functionality at the expense of appearances. This makes the interface less aesthetically pleasing.

Also, I have realised that my planning tends to be to optimistic and does not account for errors and other spill over. This is due to my underestimation of buffer time between tasks, which would be taken up by error correction.

Finally I have learnt that I am quite competent at analysis of numerical data. Based on the prototypes I have built and the results thereof, (see Appendix A.2 & B.2) it has become apparent to me that my mathematical skills are better than I thought before the project. However, if I work with the same data for too long, I tend to miss the bigger picture.

Appendix A

Card High/Low prototype

A.1 Source Code

```java
//Adding HighLow to SKCards
package SKCards;

/*
 * Importing the ArrayList class
 * and the IO components
 */
import java.util.ArrayList;
import java.io.*;


/**
 * This is the 1st prototype for the Reinforcement Learining agent
 * for the SKCards systesm, specifically SKBlackJack. This agent
 * attempts to guess of the next card is higher or lower than the
 * current card. It acheiceves this by using a simple sum of all
 * the past rewards.
 *
 * @author Saqib A Kakvi
 * @version 2.0A
 * @since 2.0A
 * @see SKCards.Card Card
 * @SK.bug system does not consider the case when the Cards are equal.
 */
public class HighLow{

    public final static int HIGH = 1;
    public final static int LOW = 0;

    double[] highv, highp;
    double[] lowv, lowp;
    double tau;
    Pack pack;
    int cscore, hscore;


```

```
    /**
     * The main method. Creates a new HighLow.
     */
    public static void main(String[] args){
        new HighLow();
    }


    /**
     * Default constructor. Creates a new learning agent, with no previous
     * knowledge. It begins learning for 10,000 episodes. After it has
     * it has learnt, it plays against the user. As part of the testing,
     * the values of the learning are written to seperate files.
     */
    public HighLow(){

        pack = new Pack(2, true);
        highv = new double[14];
        highp = new double[14];
        lowv = new double[14];
        lowp = new double[14];
        tau = 0.1;

        for(int i = 0; i < highv.length; i++){
            highv[i] = 0.0;
            highp[i] = 0.0;
            lowv[i] = 0.0;
            lowp[i] = 0.0;
        }

        System.out.println("*****LEARNING*****");
        long start = System.currentTimeMillis();
        try{

            BufferedWriter[] wr = new BufferedWriter[14];

            for(int i = 0; i < wr.length; i++){

                String name = "values " + i + ".csv";

                wr[i] = new BufferedWriter(new FileWriter(name));
                wr[i].write("low" + Integer.toString(i));
                wr[i].write("\t");
                wr[i].write("high" + Integer.toString(i));
                wr[i].newLine();
            }
            for(int i = 0; i < 10000; i++){

                episode();

                for(int j = 0; j < wr.length; j++){

                    wr[j].write(Double.toString(lowv[j]));
                    wr[j].write("\t");
```

```
                        wr[j].write(Double.toString(highv[j]));
                        wr[j].newLine();
                    }
                }

                for(int i = 0; i < wr.length; i++){
                    wr[i].flush();
                    wr[i].close();
                }
            }
            catch(Exception ex){
            }
            long stop = System.currentTimeMillis();
            long time = stop - start;
            System.out.println("*****DONE LEARNING*****");
            System.out.println("*****TOOK " + time + " ms TO LEARN*****");
            try{
                BufferedWriter bw = new BufferedWriter(new FileWriter("values.txt"));

                bw.write("num" + "\t" + "high" +  "\t" + "low");
                bw.newLine();

                for(int i = 0; i < highv.length; i++){

                    bw.write(i + "\t" + highv[i] + "\t" + lowv[i]);
                    bw.newLine();
                }

                bw.flush();
                bw.close();
            }
            catch(Exception ex){
                ex.printStackTrace();
            }
            System.out.println();
            System.out.println("NOW LETS PLAY!!!");
            System.out.println("You need to guess if the next card is higher or" +
                "lower than the current card");
            System.out.println("Enter 1 for higher, 0 for lower, " +
                "equals is not an option!");

            cscore = 0;
            hscore = 0;

            try{

                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

                for(int i = 1 ; i < 10; i++){
                    Card card1 = pack.nextCard();
                    Card card2 = pack.nextCard();

                    int c1 = card1.getPoints();
                    int c2 = card2.getPoints();
```

```
                    int caxn;

                    if(highv[c1] > lowv[c1]){
                        caxn = 1;
                    }
                    else{
                        caxn = 0;
                    }

                    System.out.println("Number␣" + i + ":␣" + card1.toString());
                    System.out.print("Enter␣your␣guess:␣");
                    int haxn = Integer.parseInt(br.readLine());
                    System.out.println("I␣guessed␣" + caxn);
                    cplay(caxn,c1,c2);
                    hplay(haxn,c1,c2);
                    System.out.println("The␣actual␣card␣is:␣" + card2.toString());
                }

                System.out.println("YOU␣SCORED:␣" + hscore);
                System.out.println("I␣SCORED␣␣:␣" + cscore);
            }
            catch(Exception ex){
            }
        }


        /**
         * This method goes through a single episode of the learning. This
         * involves the agent picking an action based on the value of the
         * card. The selected action is then taken.
         */
        public void episode(){


            Card card1 = pack.nextCard();
            Card card2 = pack.nextCard();

            int c1 = card1.getPoints();
            int c2 = card2.getPoints();

            //Jo < A < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < T  < J  < Q  < K
            //0  < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < 11 < 12 < 13

            double hv = Math.exp(highv[c1]/tau);
            double lv = Math.exp(lowv[c2]/tau);
            double sum = hv + lv;

            highp[c1] = hv/sum;
            lowp[c1] = lv/sum;

            double r = Math.random();

            if(r > lowp[c1]){
```

```
                action(HIGH,c1,c2);
            }
            else{
                action(LOW,c1,c2);
            }
        }


        /**
         * Carries out a single action, high or low, based on the parameters
         * recieved from the episode method. The values of the two cards are
         * also passed for verification. The score is adjusted accordinly,
         * based on the values of the cards and the axn taken.
         *
         * @param axn the axn to be taken
         * @param c1 the value of Card1
         * @param c2 the value of Card2
         */
        public void action(int axn, int c1, int c2){

            if(axn == HIGH){
                if(c1 < c2){
                    highv[c1] += 1;
                }
                else{
                    highv[c1] -= 1;
                }
            }
            else if(axn == LOW){
                if(c1 > c2){
                    lowv[c1] += 1;
                }
                else{
                    lowv[c1] -= 1;
                }
            }
        }


        /**
         * This method is used for a single guess made by the computer in the
         * head to head game with the human. If the computer guessed correctly
         * it gets a point.
         *
         * @param axn the axn to be taken
         * @param c1 the value of Card1
         * @param c2 the value of Card2
         */
        public void cplay(int axn, int c1, int c2){

            if(axn == 1){
                if(c1 < c2){
                    cscore++;
                }
```

```
        }
        else if(axn == 0){
            if(c1 > c2){
                cscore++;
            }
        }
    }


    /**
     * This method is used for a single guess made by the human in the
     * head to head game with the computer. If the human guessed correctly
     * it gets a point.
     *
     * @param axn the axn to be taken
     * @param c1 the value of Card1
     * @param c2 the value of Card2
     */
    public void hplay(int axn, int c1, int c2){

        if(axn == 1){
            if(c1 < c2){
                hscore++;
            }
        }
        else if(axn == 0){
            if(c1 > c2){
                hscore++;
            }
        }
    }
}
```

A.2 Results

This section contains the results of one run of the High/Low prototype. Several runs were conducted and the results were found to be fairly consistent. The reason for variations in the results was the fact that no proper learning algorithm was used, hence there was no supporting proof of convergence. The following graphs illustrate the values of guessing high or low for a given card, with respect to the number of occurrences.

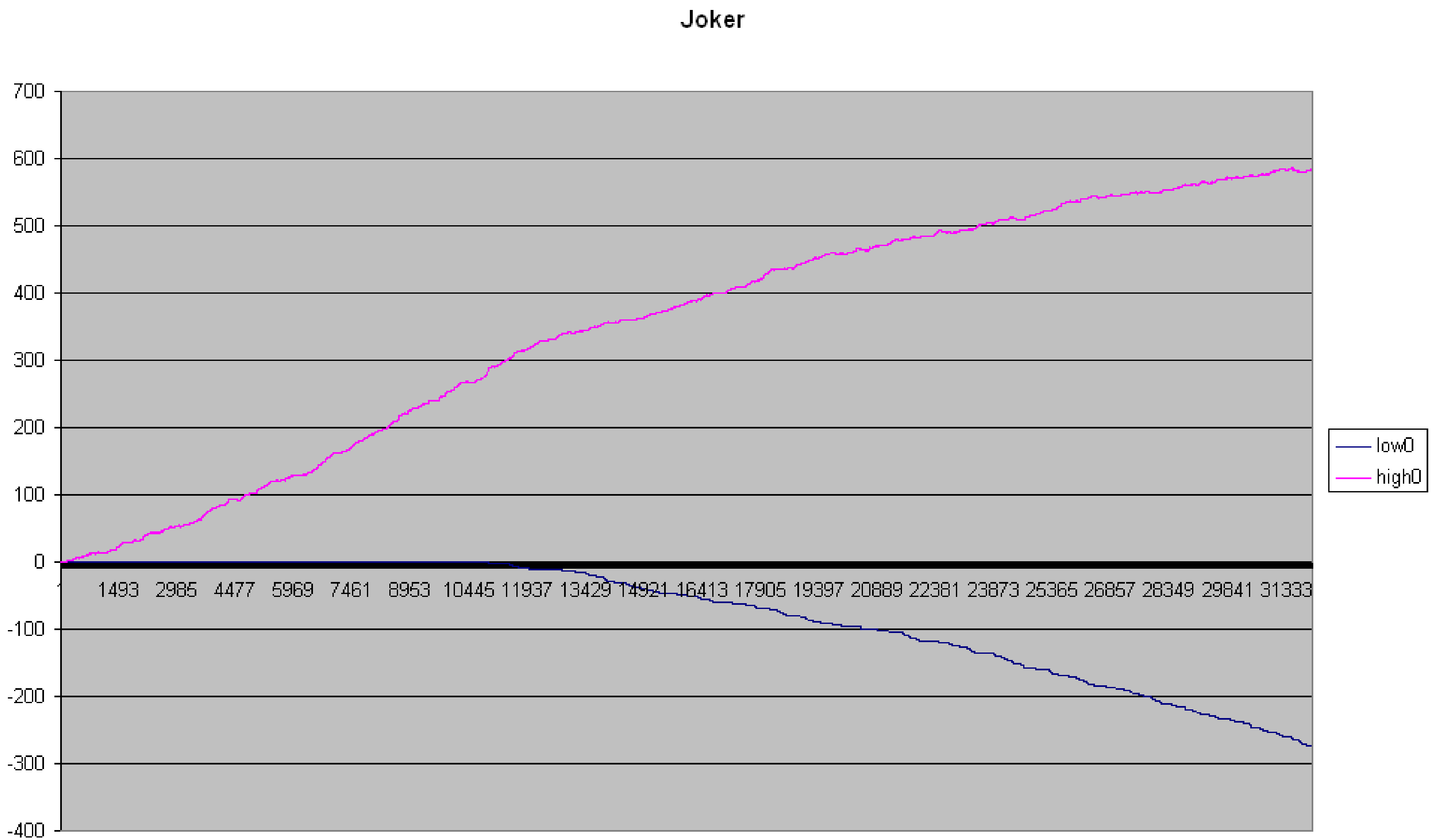


Figure A.1: Values for Joker

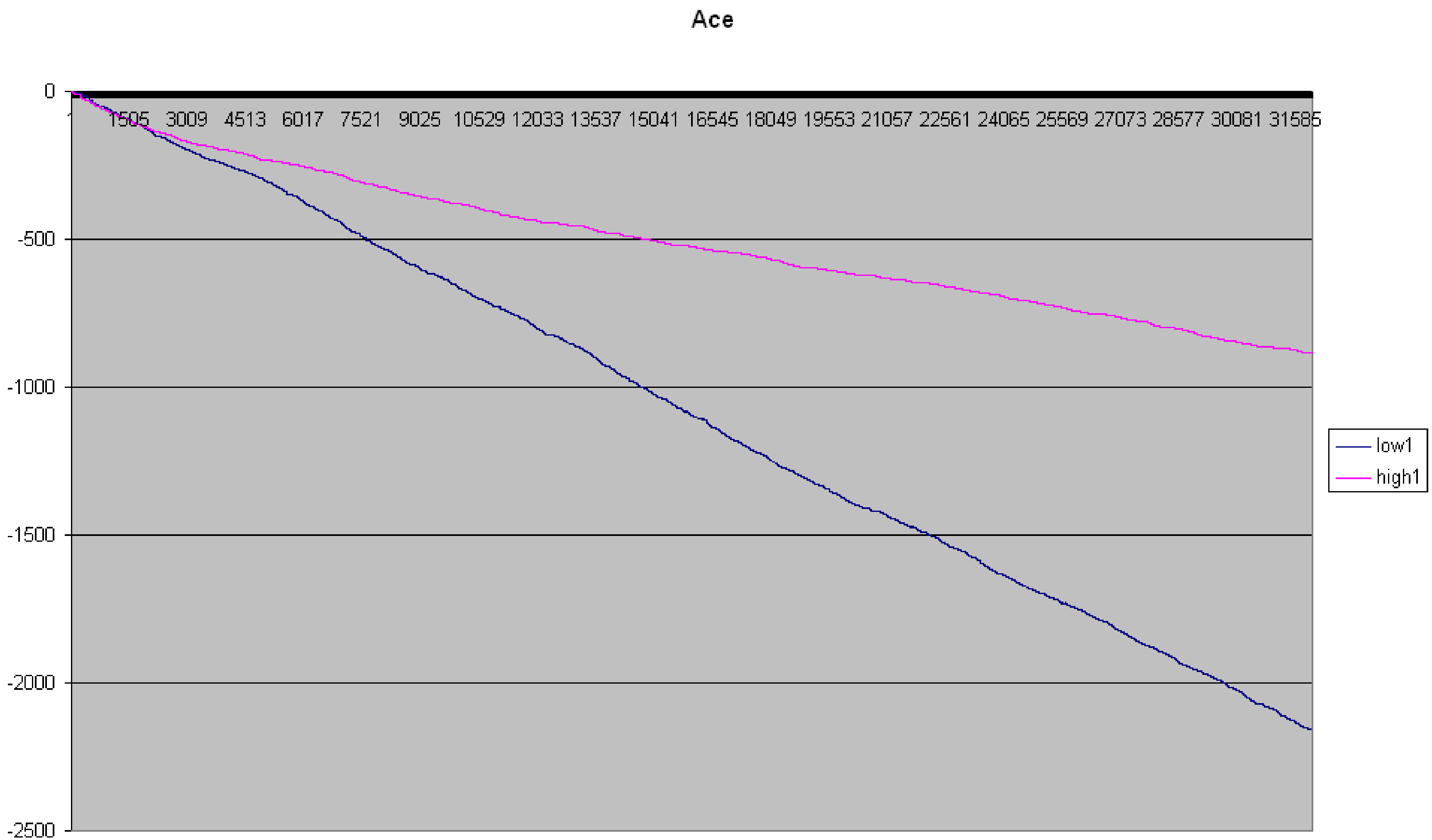


Figure A.2: Values for Ace

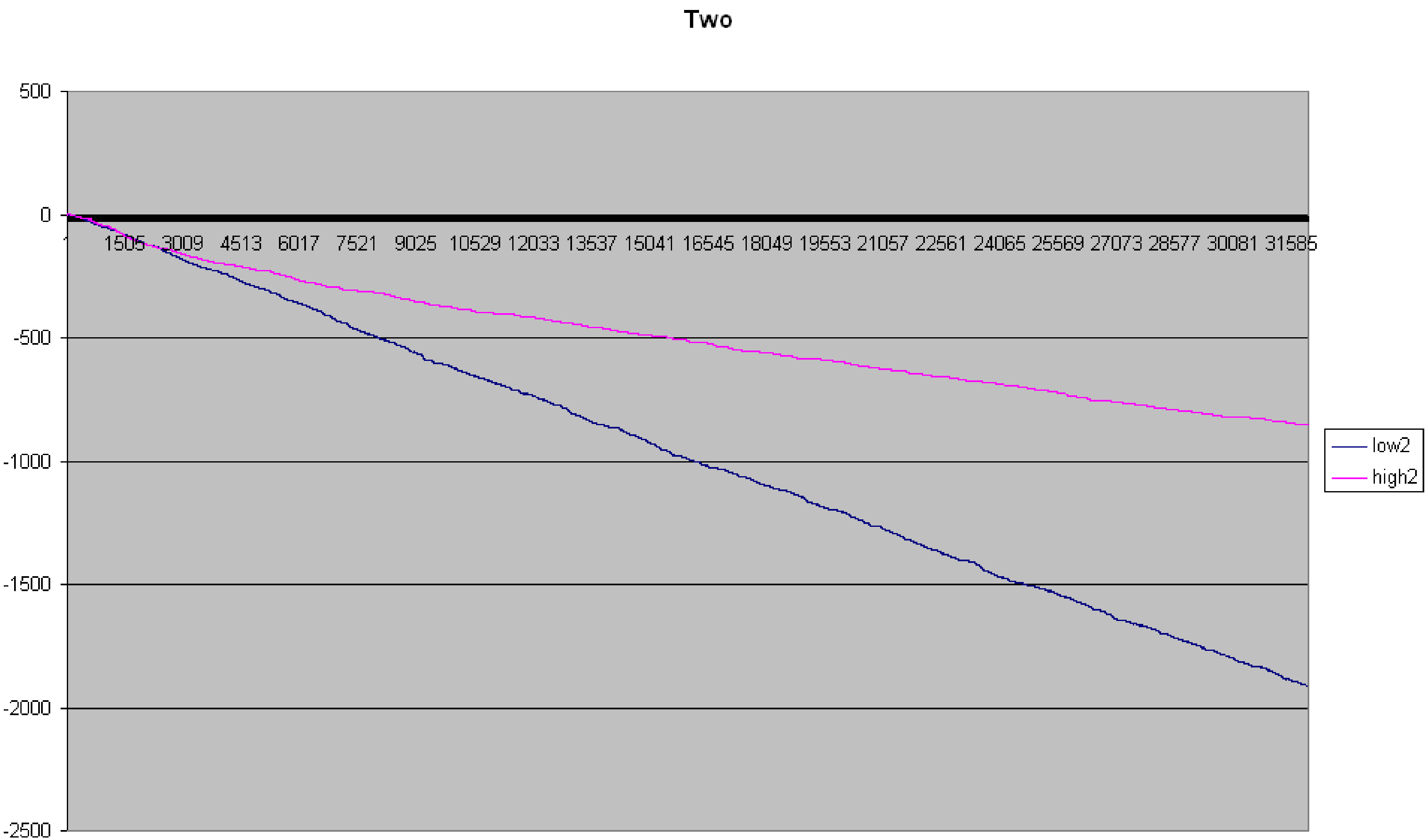


Figure A.3: Values for 2

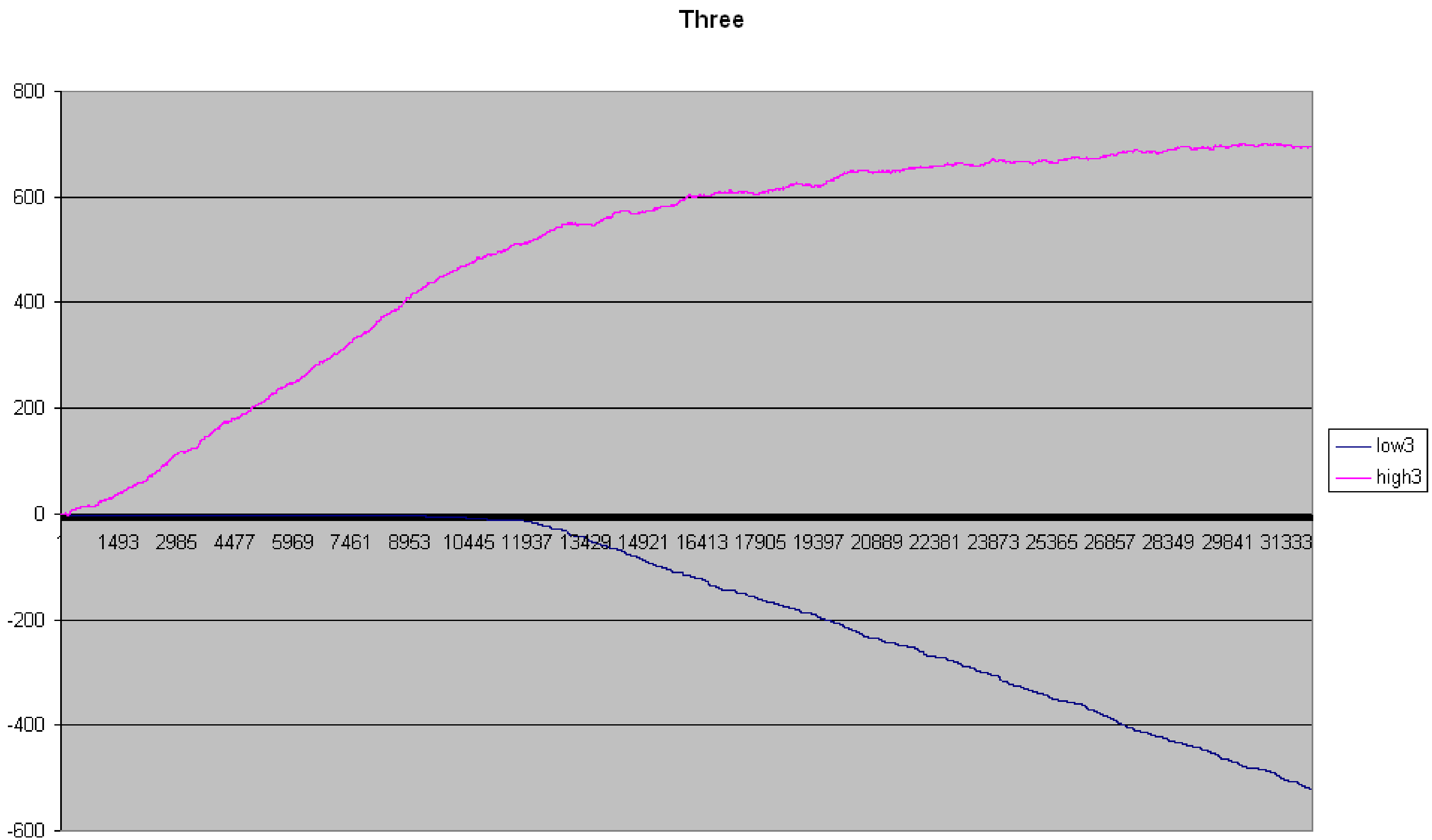


Figure A.4: Values for 3

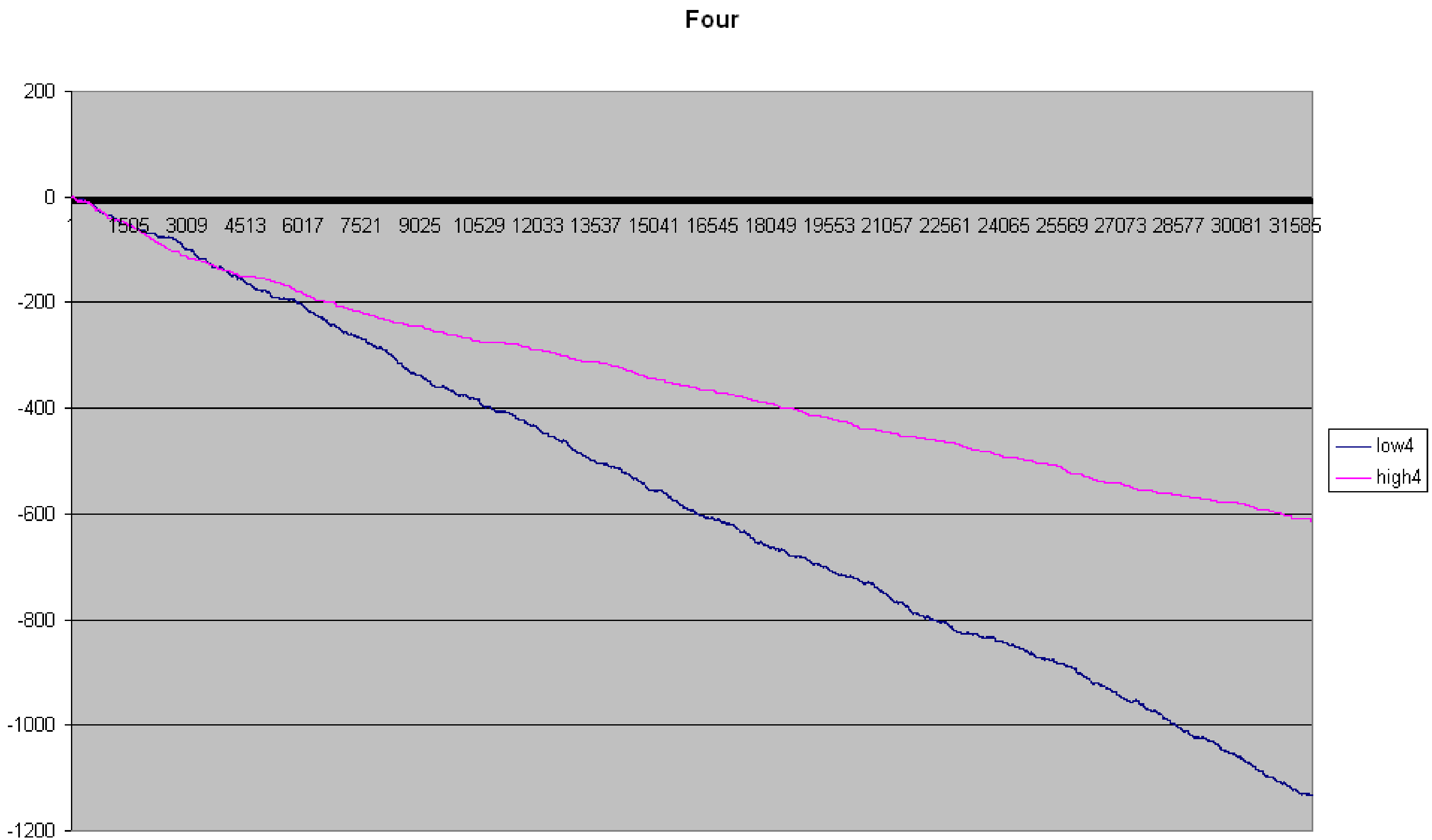


Figure A.5: Values for 4

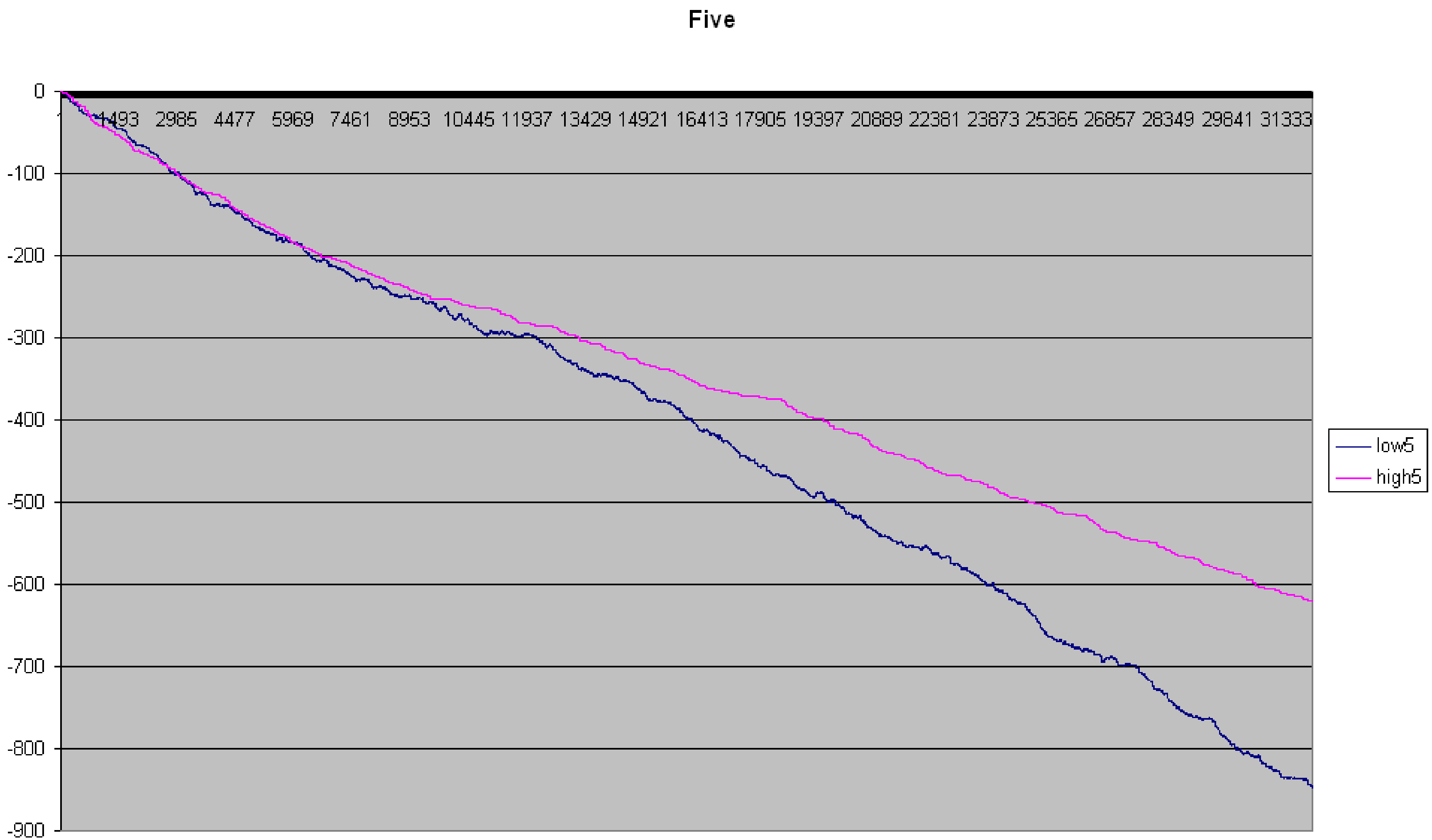


Figure A.6: Values for 5

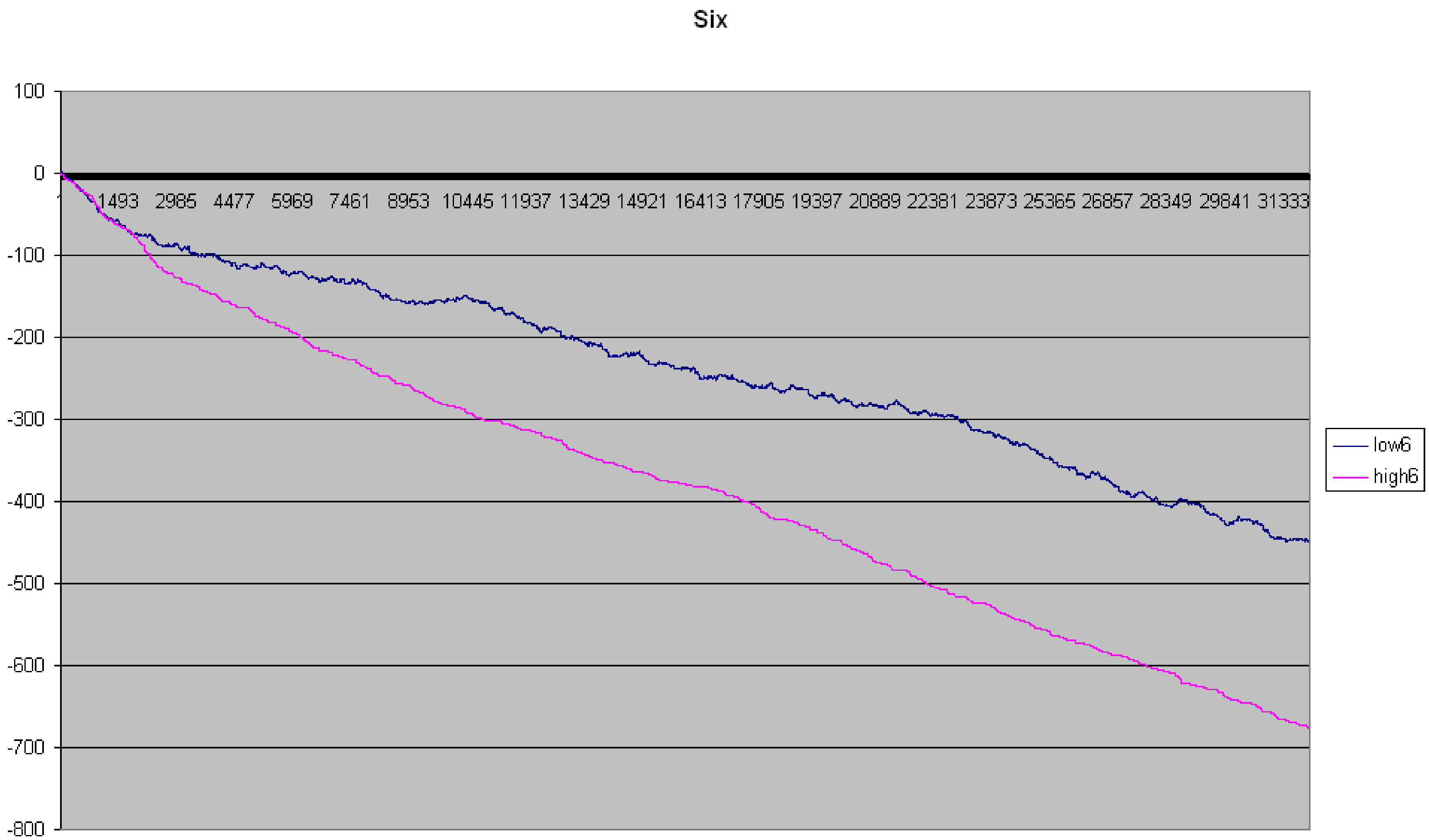


Figure A.7: Values for 6

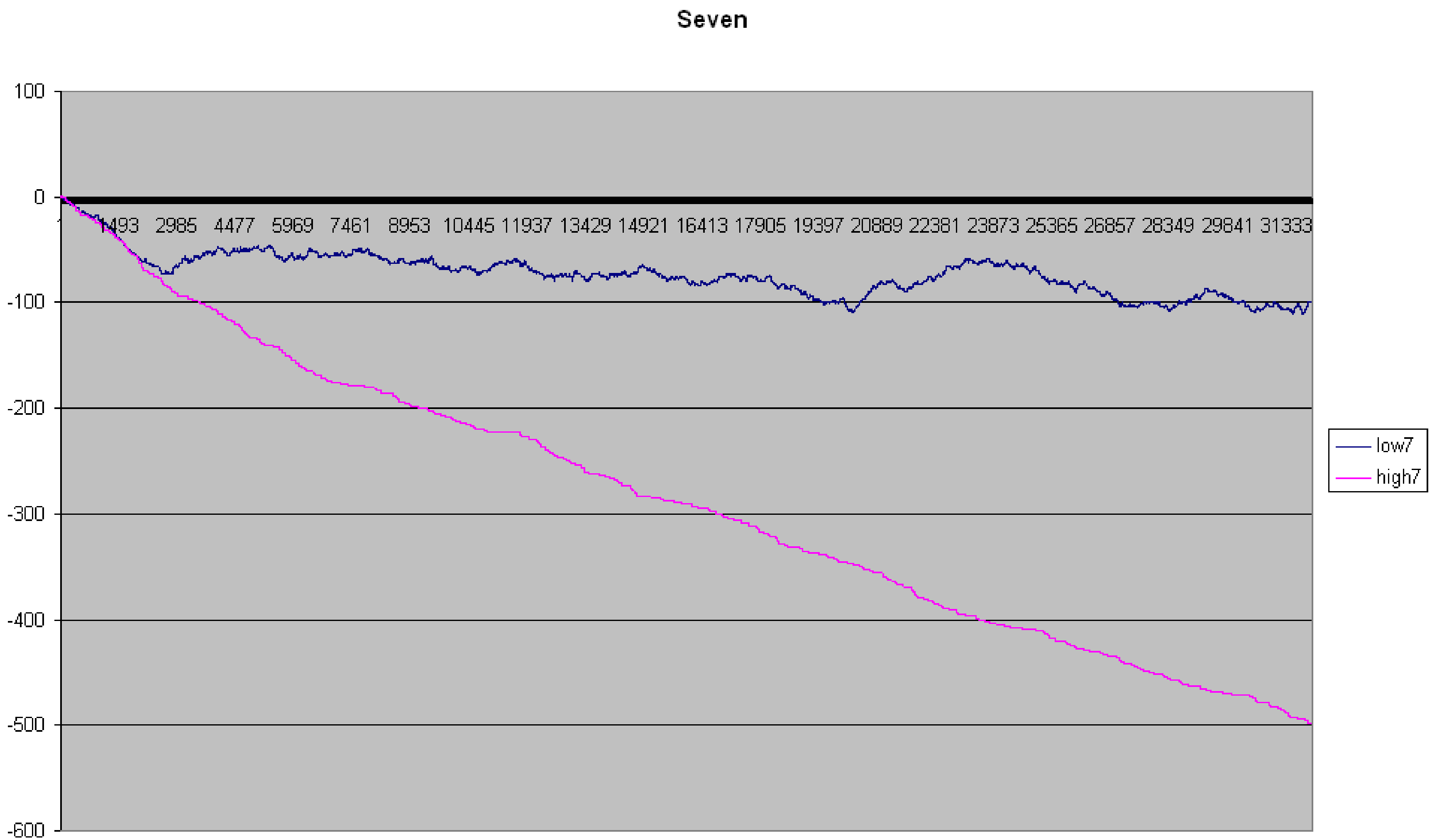


Figure A.8: Values for 7

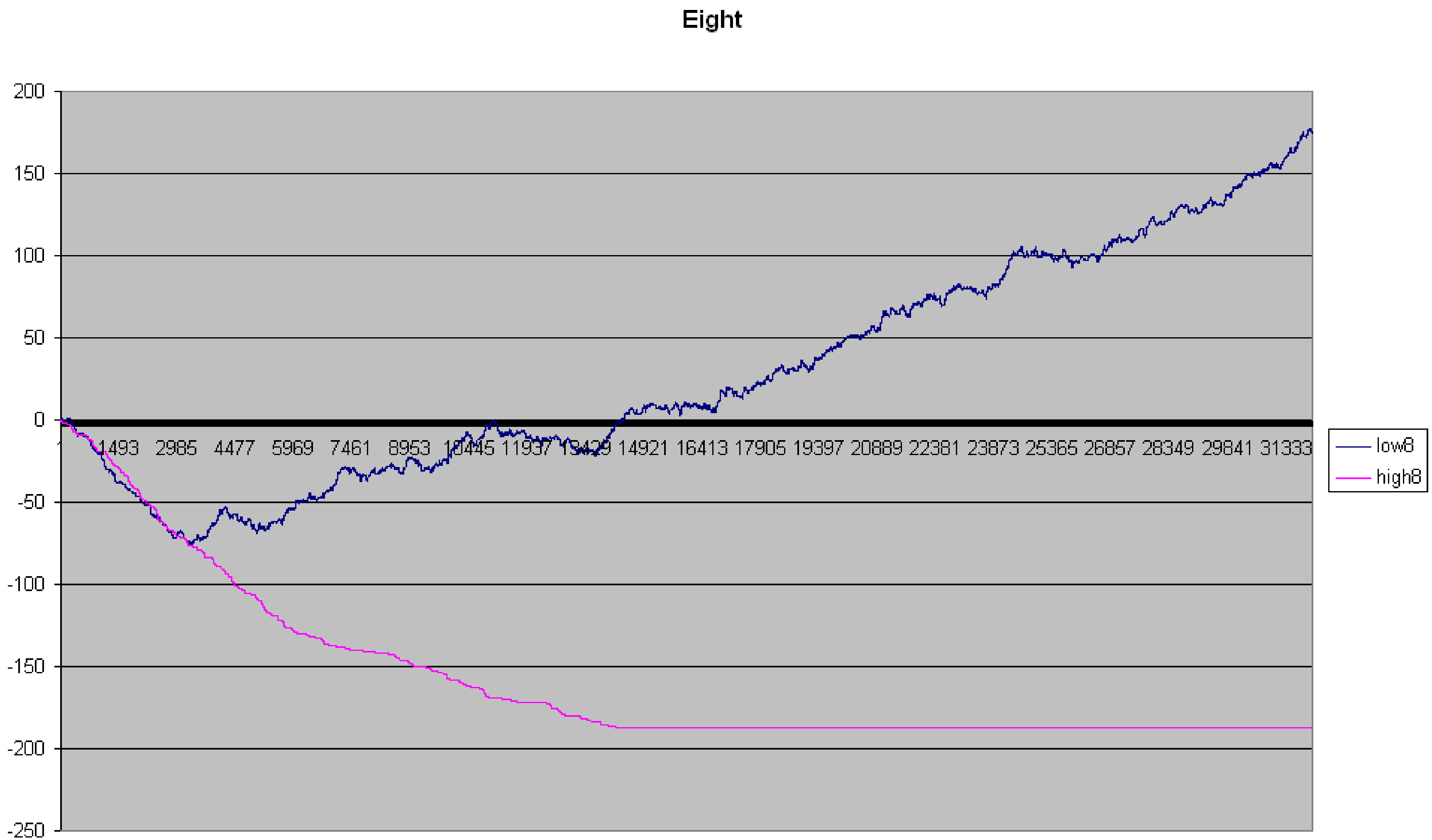


Figure A.9: Values for 8

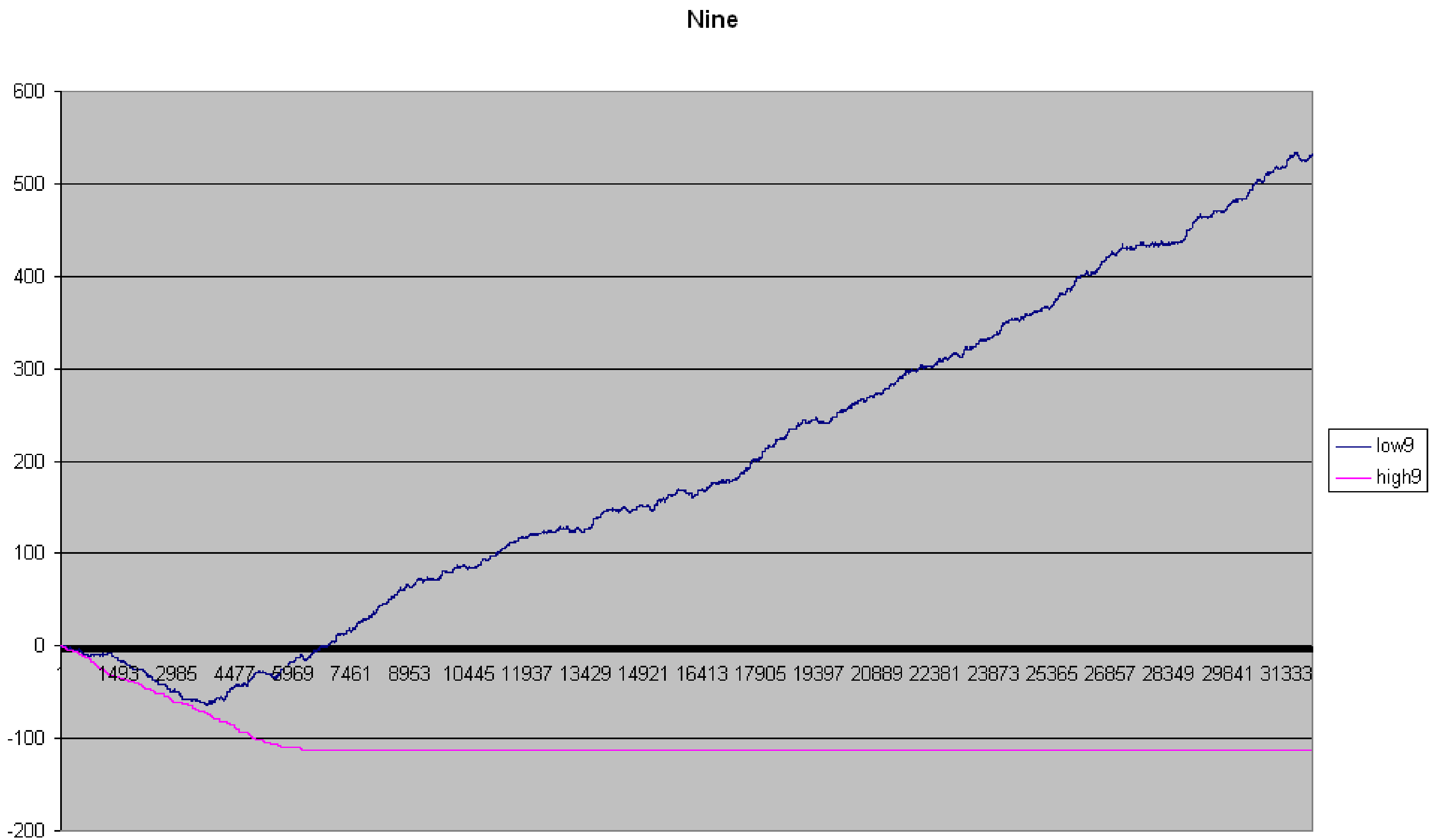


Figure A.10: Values for 9

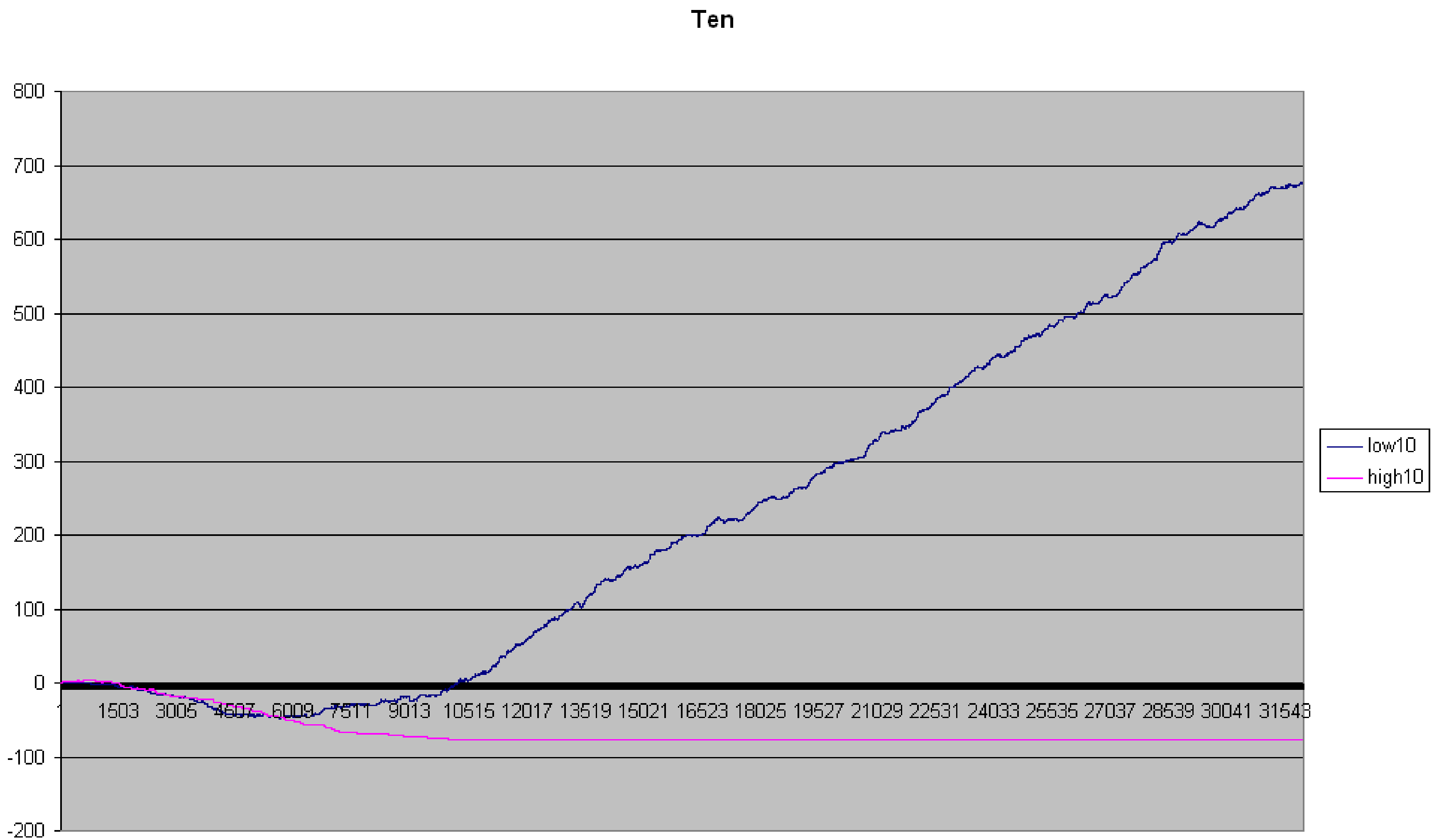


Figure A.11: Values for 10

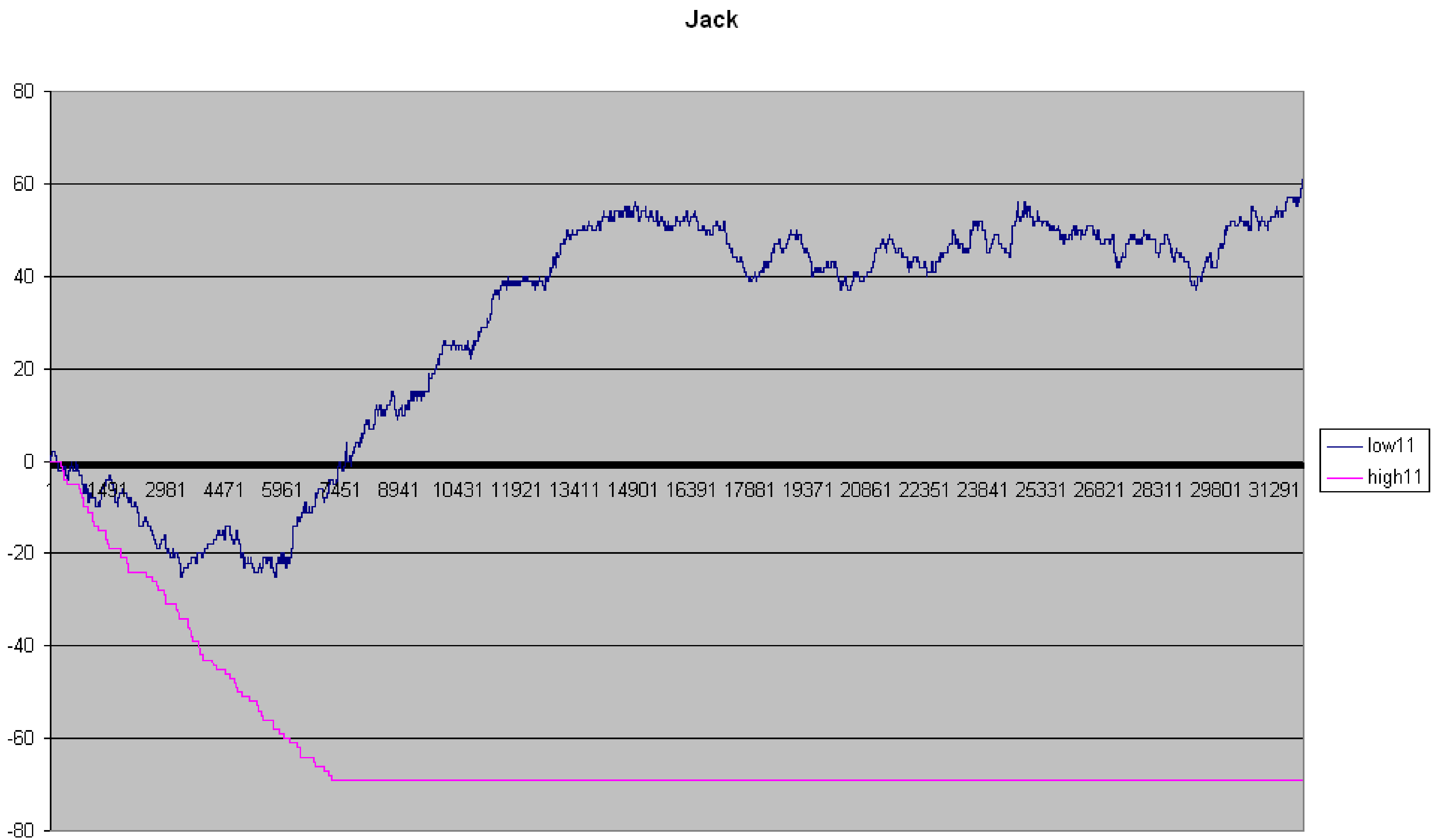


Figure A.12: Values for Jack

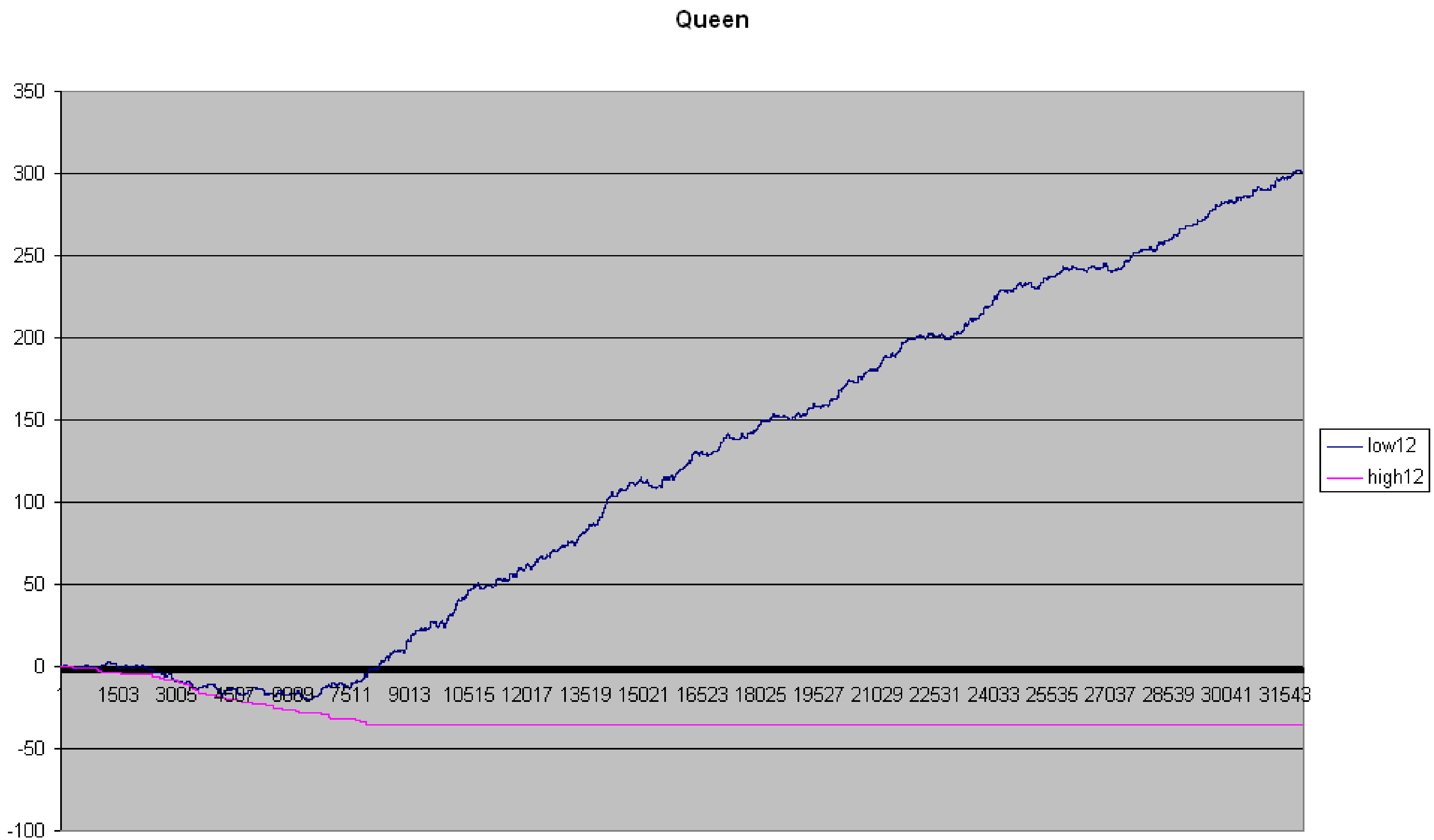


Figure A.13: Values for Queen

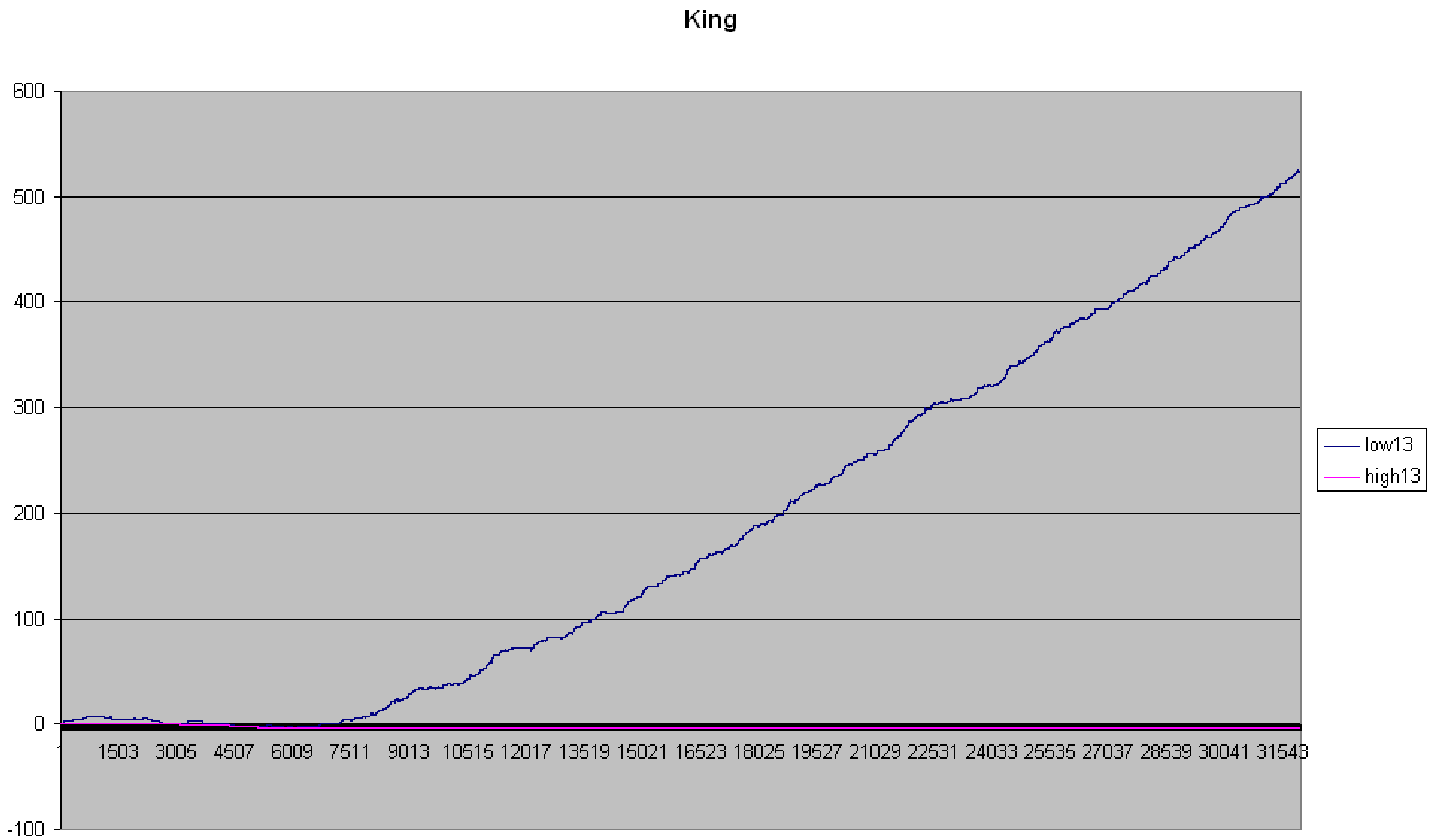


Figure A.14: Values for King

VALUE	FINAL POLICY
Joker	HIGH
2	HIGH
3	HIGH
4	HIGH
5	HIGH
6	HIGH
7	LOW
8	LOW
9	LOW
10	LOW
Jack	LOW
Queen	LOW
King	LOW

Table A.1: Final Policy

As we can see, The agent learns policies at the extremities (Joker and King) very easily. However with the middle values, it has some problems. Using a proper learning algorithm would help smooth out the learning in the middle values. It can bee seen from Table A.1 below that the that the agent has learnt a fairly good policy.

Appendix B

GridWorld prototype

B.1 Source Code

```java
//Adding GridWorld to SKCards
package SKCards;

/*
 * importing the ArrayList class
 * and the swing/awt components
 * and the border components
 * and the BufferedImage class
 * and the IO components
 * and the ImageIO class
 */
import java.util.ArrayList;
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.image.BufferedImage;
import java.io.*;
import javax.imageio.ImageIO;


/**
 * This is the second prototype for the SKCards system. This
 * class attempts to solve the GridWorld problem, using the
 * Reinforcment Learning Monte Carlo methods.
 *
 * @author Saqib A Kakvi
 * @version 2.0A
 * @since 2.0A
 * @SK.bug none
 */
public class GridWorld extends JFrame{

    private JLabel[][] grid;
    private Point loc;
    private double tau, gamma, alpha;
    private double[][] val, prob;
```

```
    private boolean[][] pen;
    private ArrayList<Point> path;
    private JFrame values;
    private JLabel[][] vlab;
    private String filename;
    private int h,w, p;

    /**
     * main method. Creates a new instance of GridWorld, using the default
     * constructor.
     *
     * @param args not used
     */
    public static void main(String[] args){new GridWorld().run();}


    /**
     * Default constructor. Creates a GridWorld instance b passing the
     * parameters 10 and 10 to the main constructor.
     *
     * @see GridWorld
     */
    public GridWorld(){ this(10,10);}


    /**
     * Main Constructor. Creates a grid of size <i>n</i>x<i>m</i>.
     * All the grid squares are created and penalties assigned to the
     * appropriate squares. The system is then run for 2500 episodes.
     *
     * @param n the length of the grid
     * @param m the width of the drid
     */
    public GridWorld(int n, int m){

        super("GridWorld");
        Dimension d = (Toolkit.getDefaultToolkit()).getScreenSize();
        h = (int)(d.getHeight()) / 4 * 3;
        w = (int)(d.getWidth()) / 2;

        setSize(w,h);
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);

        grid = new JLabel[n][m];
        tau = 0.75;
        gamma = 0.75;
        alpha = 0.5;
        val = new double[n][m];
        pen = new boolean[n][m];

        for(int i = 0 ; i < n; i++){
            for(int j = 0; j < m; j++){
                grid[i][j] = new JLabel();
                grid[i][j].setBorder(new LineBorder(Color.BLUE));
```

```
                    grid[i][j].setHorizontalAlignment(SwingConstants.CENTER);
                    int r = new Double(Math.random() * 10).intValue();
                    if((i+j == ((n+m)/2)-1) && (i != n/2)){
                        pen[i][j] = true;
                        grid[i][j].setForeground(Color.RED);
                    }
                    else{
                        pen[i][j] = false;
                    }
                }
            }

            Container cp = getContentPane();
            cp.setLayout(new GridLayout(n,m));

            for(int i = 0 ; i < n; i++){
                for(int j = 0; j < m; j++){
                    cp.add(grid[i][j]);
                }
            }

            grid[0][0].setForeground(Color.GREEN);
            grid[n-1][m-1].setForeground(Color.GREEN);

            for(int i = 0; i < val.length; i++){
                for(int j = 0; j < val[i].length; j++){
                    if(pen[i][j]){
                        val[i][j] = -100;
                    }
                    else{
                        val[i][j] = 0;
                    }
                }
            }

            val[0][0] = 100;
            val[n-1][m-1] = 100;

            values = new JFrame("VALUES");
            values.setSize(w,h);
            values.setLocation(w,0);
            values.setDefaultCloseOperation(DISPOSE_ON_CLOSE);

            Container c = values.getContentPane();

            c.setLayout(new GridLayout(n, m));

            vlab = new JLabel[n][m];

            for(int i = 0; i < vlab.length; i++){
                for(int j = 0; j < vlab[i].length; j++){

                    vlab[i][j] = new JLabel(Double.toString(val[i][j]));
                    vlab[i][j].setHorizontalAlignment(SwingConstants.CENTER);
```

```
                vlab[i][j].setOpaque(true);
                vlab[i][j].setForeground(Color.WHITE);
                vlab[i][j].setBackground(Color.BLACK);
                c.add(vlab[i][j]);
            }
        }

        reset();
        this.setVisible(true);
        values.setVisible(true);
        this.repaint();
        values.repaint();
    }

    /**
     * Runs the GridWorld program.
     */
    public void run(){

        for(int i = 1; i <= 2500; i++){
            filename = new String("TEST" + Integer.toString(i) + ".jpg");
            setTitle("Episode " + i);
            episode();
            reset();
            screenShot();
        }
        System.out.println(p + " PENALTIES");
    }


    /**
     * Sets the GridWorld and the values frame's visiblity using
     * the value passed.
     *
     * @param vis the visibility of the windows.
     */
    public void setVisibility(boolean vis){

        this.setVisible(vis);
        values.setVisible(vis);
    }


    /**
     * This method runs a sningle episode through the system. It
     * continues to move a single step until it reaches a terminal
     * sqaure. When this happens, a screenshot of the windows is taken.
     *
     * @see step
     */
    private void episode(){

        int x = (int)((Math.random() * 100) % grid.length);
        int y = (int)((Math.random() * 100) % grid[0].length);
```

```
        loc = new Point(x,y);

        while(checkloc() != 0){
            x = (int)((Math.random() * 100) % grid.length);
            y = (int)((Math.random() * 100) % grid[0].length);

            loc = new Point(x,y);
        }

        path = new ArrayList<Point>();

        grid[x][y].setText("X");
        while(checkloc() == 0){
            step();
        }

        reward();
    }


    /**
     * Testing method. Used to take a screenshot of the system
     * after each episode. This allows the progress and results
     * of the system to be documented.
     */
    private void screenShot(){
        //CODE TAKEN FROM
        //www.java-tips.org/java-se-tips/java.awt/how-to-capture-screenshot.html
        try {
            Robot robot = new Robot();
            // Capture the screen shot of the area of the screen
            // defined by the rectangle
            BufferedImage bi=robot.createScreenCapture(new Rectangle(w*2,h));
            ImageIO.write(bi, "jpg", new File(filename));
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }


    /**
     *
     */
    private void reset(){

        for(int i = 0; i < grid.length; i++){
            for(int j = 0; j < grid[i].length; j++){
                if(pen[i][j]){
                    grid[i][j].setText("PEN");
                }
                else{
                    grid[i][j].setText("");
```

```
                }
            }
        }

        grid[0][0].setText("GOAL");
        grid[grid.length-1][grid[0].length-1].setText("GOAL");

        for(int i = 0; i < vlab.length; i++){
            for(int j = 0; j < vlab[i].length; j++){
                vlab[i][j].setText(Double.toString(val[i][j]));
                if(val[i][j] > 0){
                    int v = (int)val[i][j] * 255 / 100;
                    vlab[i][j].setBackground(new Color(0, v ,0));
                }
                else{
                    int v = (int)val[i][j] * 255 / 100 * -1;
                    vlab[i][j].setBackground(new Color(v, 0 ,0));
                }
            }
        }

        repaint();
    }


    /**
     *
     */
    private int checkloc(){

        if(getx() == 0 && gety() == 0){
            return 1;
        }
        else if(getx() == grid.length-1 && gety() == grid[0].length-1){
            return 1;
        }
        else if(pen[getx()][gety()]){
            return -1;
        }
        else{
            return 0;
        }
    }


    /**
     * This method moves the agent a single step. The direction
     * is decided on using the softmax probability distributions.
     *
     * @see action(int)
     */
    private void step(){

        double lim = 0.0;
```

```
        double sum = 0.0;
        int x = getx();
        int y = gety();
        double[] prob = new double[4];

        double up,down,left ,right;

        boolean uv = true, lv = true, dv = true, rv = true;

        if(getx() == 0){
            uv = false;
        }
        else if(getx() == grid.length -1){
            dv = false;
        }

        if(gety() == grid[0].length -1){
            rv = false;
        }
        else if(gety() == 0){
            lv = false;
        }

        if(uv){
            up = Math.exp(val[x-1][y]/tau);
            sum += up;
        }
        if(rv){
            right = Math.exp(val[x][y+1]/tau);
            sum += right;
        }
        if(dv){
            down = Math.exp(val[x+1][y]/tau);
            sum += down;
        }
        if(lv){
            left = Math.exp(val[x][y-1]/tau);
            sum += left;
        }

        if(uv){
            prob[0] = Math.exp(val[x-1][y]/tau) / sum;
        }
        if(rv){
            prob[1] = Math.exp(val[x][y+1]/tau) / sum;
        }
        if(dv){
            prob[2] = Math.exp(val[x+1][y]/tau) / sum;
        }
        if(lv){
            prob[3] = Math.exp(val[x][y-1]/tau) / sum;
        }

        double r = Math.random();
```

```
        for(int i = 0; i < prob.length; i++){

            lim += prob[i];
            if(r < lim){
                action(i);
                break;
            }
        }

        repaint();
    }


    /**
     * Takes an action, that is moving 1 step in any of the
     * valid directions.
     *
     * @param axn the direction to move.
     */
    private void action(int axn){

        path.add(new Point(loc));

        switch(axn){
            case 0: //up
                loc.translate(-1,0);
                break;
            case 1: //right
                loc.translate(0,1);
                break;
            case 2: //down
                loc.translate(1,0);
                break;
            case 3: //left
                loc.translate(0,-1);
                break;
        }

        grid[getx()][gety()].setText("X");
    }


    /**
     *
     */
    private void reward(){

        if(checkloc() == -1) p++;
        double reward = val[getx()][gety()];

        for(int i = path.size()-1; i >=0; i--){

            reward *= gamma;
```

```java

                Point p = path.get(i);
                int x = (int)p.getX();
                int y = (int)p.getY();
                val[x][y] += alpha * (reward - val[x][y]);

                reward = val[x][y];
            }
        }


        /**
         * Returns the x co-ordinate of the agent's current
         * position, as an int so it can be used as an
         * index in the grid array.
         *
         * @return the x co-ordinate
         * @see gety
         */
        private int getx(){return (int)loc.getX();}


        /**
         * Returns the y co-ordinate of the agent's current
         * position, as an int so it can be used as an
         * index in the grid array.
         *
         * @return the y co-ordinate
         * @see getx
         */
         private int gety(){return (int)loc.getY();}
}
```

B.2 Results

The GridWorld problem can be said to be mathematically trivial, as the values can be calculated very easily. Given the final reward r and the discount rate γ, we can calculate the value v of a square n steps away from the goal using the formula:

$$v = r \times \gamma^n$$

Based on this formula, we can calculate the values of all the grid squares based on their distance from the goal square as given in the table below:

Included below are some screen shots taken during the execution of the program. These screen shots selected were before episodes 1, 25, 50, 100, 150, 200, 250, 500, 750, 1000, 1500 & 2500.

STEPS TO GOAL	VALUE
0	100
1	75
2	56.25
3	42.1875
4	31.6406
5	23.7305
6	17.7979
7	13.3484
8	10.0113
9	7.5085
10	5.6314
11	4.2235
12	3.1676

Table B.1: Predicted values for the GridWorld problem

Figure B.1: Episode 1

Figure B.2: Episode 25

Figure B.3: Episode 50

Figure B.4: Episode 100

Figure B.5: Episode 150

Figure B.6: Episode 200

Figure B.7: Episode 250

Figure B.8: Episode 500

Figure B.9: Episode 750

Figure B.10: Episode 1000

Figure B.11: Episode 1500

Figure B.12: Episode 2000

Figure B.13: Episode 2500

Appendix C

Blackjack AI

C.1 Source Code

C.1.1 Face

```
//Adding Face to SKCards
package SKCards;


/**
 * Defines an enum of face values. The values are 2-10(the words),
 * Jack, Queen, King, Ace and Joker.
 *
 * @author SK
 * @version 2.0A
 * @since 1.0A
 * @see SKCards.Suit Suit values
 * @SK.bug none
 */
public enum Face{TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
    JACK, QUEEN, KING, ACE, JOKER};
```

C.1.2 Suit

```
//Adding Suit to SKCards
package SKCards;


/**
 * Defines an enum of the playing card suits. The values are Diamonds, Clubs,
 * Hearts, Spades and Null for jokers.
 *
 * @author SK
 * @version 2.0A
 * @since 1.0A
 * @see SKCards.Face Face values
 * @SK.bug none
 */
public enum Suit{DIAMONDS, CLUBS, HEARTS, SPADES, NULL}
```

C.1.3 Card

```java
//Adding Card to SKCards
package SKCards;


/*
 * Importing the ArrayList class
 * and the swing/awt components
 */
import java.util.ArrayList;
import java.awt.*;
import javax.swing.*;


/**
 * Card represents a playing card and has two sates - suit and face value.
 * Suit is defined in an enum and can be Diamonds, Clubs, Hearts, Spades or
 * Null for Jokers. The face value is also defined in and enum and is the
 * string representation of the card's face value.
 *
 * @author SK
 * @version 2.0A
 * @since 1.0A
 * @SK.bug none
 */
public class Card{

    private Suit suit;
    private Face faceValue;

    /**
     * Constructs a new card with the values passed as arguments.
     * @param v the face value of the card.
     * @param s the suit of the card.
     * @see SKCards.Face Face Values
     * @see SKCards.Suit Suit Vaules
     */
    public Card(Face v, Suit s){

        faceValue = v;
        suit = s;
    }


    /**
     * Returns the face value of the active Card.
     * @return the code of the Face value.
     */
    public String getValue(){

        if(this == null){
            return " ";
        }
```

```
        else if(faceValue == Face.TWO){
            return "2";
        }
        else if(faceValue == Face.THREE){
            return "3";
        }
        else if(faceValue == Face.FOUR){
            return "4";
        }
        else if(faceValue == Face.FIVE){
            return "5";
        }
        else if(faceValue == Face.SIX){
            return "6";
        }
        else if(faceValue == Face.SEVEN){
            return "7";
        }
        else if(faceValue == Face.EIGHT){
            return "8";
        }
        else if(faceValue == Face.NINE){
            return "9";
        }
        else if(faceValue == Face.TEN){
            return "T";
        }
        else if(faceValue == Face.JACK){
            return "J";
        }
        else if(faceValue == Face.QUEEN){
            return "Q";
        }
        else if(faceValue == Face.KING){
            return "K";
        }
        else if(faceValue == Face.ACE){
            return "A";
        }
        else if(faceValue == Face.JOKER){
            return "Joker";
        }
        else{
            return "";
        }
    }


    /**
     * Returns the face value of the active Card.
     * @return the code of the Suit value.
     */
    public String getSuit(){

```

```java
        if(this == null){
            return "";
        }
        else if(suit == Suit.NULL){
            return "";
        }
        else{
            return (new Character(suit.toString().charAt(0))).toString();
        }
    }


    /**
     * Gets the String value of the card int the form FS(FaceSuit), except the
     * joker. eg the 2 of Hearts is 2H, the 10 of Diamonds is TD, the King of
     * Spades is KS, the Joker is Joker.
     * @return the String value of the card
     */
    public String toString(){

        if(this == null){
            return " ";
        }
        else if(faceValue == Face.JOKER){
            return "Joker";
        }
        else{
            return (getValue() + getSuit());
        }
    }


    /**
     * Gets the image associated with the given card, using the cardSet provided.
     * @param cs the cardSet
     * @return the card's image
     */
    public ImageIcon toImage(String cs){

        return new ImageIcon("images/cards/" + cs + "/"+ toString() +".jpg");
    }


    /**
     * Returns the image of the card face down, using the back provided
     * @param b the back
     * @return the card's back image
     */
    public ImageIcon toBackImage(String b){

        return new ImageIcon("images/backs/" + b);
    }


```

```
    /**
     * Prints out the description of the active card.
     */
    public void print(){

        System.out.println(this.toString());
    }


    /**
     * Returns the point value of the card for general uses. This methods counts
     * Ace as 1, number cards at face value, Jack as 11,Queen as 12, King as 13
     * and Joker as 0.
     * @return the point value of the card.
     */
    public int getPoints(){

        if(faceValue == Face.ACE){
            return 1;
        }
        else if(faceValue == Face.TWO){
            return 2;
        }
        else if(faceValue == Face.THREE){
            return 3;
        }
        else if(faceValue == Face.FOUR){
            return 4;
        }
        else if(faceValue == Face.FIVE){
            return 5;
        }
        else if(faceValue == Face.SIX){
            return 6;
        }
        else if(faceValue == Face.SEVEN){
            return 7;
        }
        else if(faceValue == Face.EIGHT){
            return 8;
        }
        else if(faceValue == Face.NINE){
            return 9;
        }
        else if(faceValue == Face.TEN){
            return 10;
        }
        else if(faceValue == Face.JACK){
            return 11;
        }
        else if(faceValue == Face.QUEEN){
            return 12;
        }
        else if(faceValue == Face.KING){
```

```
                return 13;
            }
            else{
                return 0;
            }
        }

        /**
         * Gets the point value of a face value card.
         * @param faceValue the face value of the cards.
         */
        public static int getPoints(Face faceValue){

            if(faceValue == Face.ACE){
                return 1;
            }
            else if(faceValue == Face.TWO){
                return 2;
            }
            else if(faceValue == Face.THREE){
                return 3;
            }
            else if(faceValue == Face.FOUR){
                return 4;
            }
            else if(faceValue == Face.FIVE){
                return 5;
            }
            else if(faceValue == Face.SIX){
                return 6;
            }
            else if(faceValue == Face.SEVEN){
                return 7;
            }
            else if(faceValue == Face.EIGHT){
                return 8;
            }
            else if(faceValue == Face.NINE){
                return 9;
            }
            else if(faceValue == Face.TEN){
                return 10;
            }
            else if(faceValue == Face.JACK){
                return 11;
            }
            else if(faceValue == Face.QUEEN){
                return 12;
            }
            else if(faceValue == Face.KING){
                return 13;
            }
            else{
                return 0;
```

```
        }
    }

    /**
     * Returns the points value of the Card for Blackjack. This
     * method is only used by the BlackJack prototype. The SKCards
     * system uses the BJRules.getBJPoints(Card) method, which is
     * almost identical.
     *
     * @return the Blackjack points value of the Card.
     */
    public int getBJPoints(){

        int points = getPoints();

        if(points == 10 || points == 11 || points == 12 || points == 13){
            return 10;
        }
        else{
            return points;
        }
    }
}
```

C.1.4 Pack

```
//Adding Pack to SKCards
package SKCards;


/*
 * Importing the ArrayList class
 */
import java.util.ArrayList;


/**
 * Represents a pack of cards. A pack is composed of a number of decks, with
 * or without jokers, both to be specified by the user.
 *
 * @author SK
 * @version 2.0A
 * @since 1.0A
 * @see SKCards.Card Card
 * @SK.bug none
 */
public class Pack{

    private final static Suit[] suits = Suit.values();
    private final static Face[] vals = Face.values();

    private ArrayList<Card> pack;
    protected int decks, currentCard;
    private boolean jokers;
```

```java


    /**
     * The default constructer. Creates a pack of Cards, with
     * 1 deck and no jokers
     */
    public Pack(){
        this(1,false);
    }

    /**
     * Creates a new pack of cards. This pack is defined by
     * the parameters passed to it.
     *
     * @param decks the number of decks in the pack.
     * @param jokers whether the decks have jokers or not.
     * @see SKCards.Card#makeDeck
     */
    public Pack(int d, boolean j){

        decks = d; jokers = j;

        pack = new ArrayList<Card>();

        ArrayList<Card> deck = new ArrayList<Card>();

        for(int i = 1; i <= decks; i++){

            deck = makeDeck(jokers);
            deck = shuffle(deck);

            for(Card c: deck){

                pack.add(c);
            }
        }

        pack = Pack.shuffle(pack);
        currentCard = 0;
    }


    /**
     * Creates a new ordered deck of cards, with or without jokers depending
     * on the arguments passed.
     * @param jokers a boolean which determines if the deck has jokes or not.
     * @return deck an ArrayList of the cards.
     */
    public static ArrayList<Card> makeDeck(boolean jokers){

        ArrayList<Card> deck = new ArrayList<Card>();

        for(int i = 0; i < suits.length - 1; i++){
            for(int j = 0; j < vals.length - 1; j++){
```

```java

                deck.add(new Card(vals[j], suits[i]));
            }
        }


        if(jokers){
            deck.add(new Card(Face.JOKER, Suit.NULL));
            deck.add(new Card(Face.JOKER, Suit.NULL));
        }

        return deck;
    }


    /**
     * Prints out all the cards in the deck.
     */
    public void print(){

        for(Card c: pack){

            c.print();
        }
    }


    /**
     * Takes a pack of cards and shuffles them.
     * @param pack the pack of cards to be shuffle.
     * @return the shuffledPack.
     */
    public static ArrayList<Card> shuffle(ArrayList<Card> pack){

        ArrayList<Boolean> shuffled = new ArrayList<Boolean>();
        ArrayList<Card> shuffledPack = new ArrayList<Card>();

        for(int i = 0; i < pack.size(); i++){

            shuffled.add(false);
        }

        while(shuffled.contains(false)){

            Double numb = Math.random();
            numb *= 125;
            int numbr = numb.intValue();
            int num = numbr % pack.size();

            if(!shuffled.get(num)){

                shuffledPack.add(pack.get(num));
                shuffled.set(num, true);
            }
```

```
        }

        return shuffledPack;
    }

    /**
     * Regenerates the Pack, by creating a new Pack. This is identical to the
     * previous only in the number of decks and the presence of jokers, but
     * the order of the Cards is different.
     */
    protected void newPack(){

        pack = new ArrayList<Card>();

        ArrayList<Card> deck = new ArrayList<Card>();

        for(int i = 1; i <= decks; i++){

            deck = makeDeck(jokers);
            deck = shuffle(deck);

            for(Card c: deck){

                pack.add(c);
            }
        }

        pack = Pack.shuffle(pack);
        currentCard = 0;
    }

    /**
     * Gives the next Card in the Pack. If the Pack has been used
     * up, a new Pack is genereated.
     *
     * @return the next Card
     */
    public Card nextCard(){

        if(currentCard >= size()){
            newPack();
        }

        currentCard++;
        return pack.get(currentCard-1);
    }


    /**
     * Returns the size of the Pack
     *
     * @return size of the Pack
     */
    public int size(){
```

```

        return pack.size();
    }

    /**
     * Returns the number of decks in the Pack
     *
     * @return the number of decks in the Pack
     */
    public int decks(){

        return decks;
    }


    /**
     * Returns a boolean to indicate if there are
     * Jokers in the Pack or not
     *
     * @rerurn true if there are Joers
     * @return false if there are no Jokers
     */
    public boolean jokers(){

        return jokers;
    }
}
```

C.1.5 Blackjack

```
//Adding BlackJack to SKCards
package SKCards;

/*
 * Importing the swing/awt components and the event models
 * and the I/O classes
 */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.io.*;


/**
 * Blackjack game with Temporal Difference AI.
 */
public class BlackJack extends JFrame{

    /**
     * This class represents a player in the BlackJack game. It can be either a human
     * player or a Reinforcement Learning player, using Temporal Differance.
     * @author Saqib A Kakvi (33106087)
     * @version 1.0
```

```java
     * @rubish yes it is
     */
    private class BPlayer{

        //ACTIONS
        private static final int HIT = 1;
        private static final int STAND = 0;
        private static final int DEAL = -1;

        //CONSANTS
        private final double TAU = 0.5;
        private final double G = 0.75;
        private final double A = 0.6;

        //the player's hand
        private Card[] hand;
        //the player's funds
        private int funds;
        //the players scores counting low/high when applicable
        private int score, hscore;
        //the values attached of the state-action pairs
        private double[][] val;
        //*****TESTING****** the number of times the agent takes a certain action
        private int[][] count;
        //the players last action and score
        private int lastaxn, lastscore;
        //flags to indicate if the player is playing an ace as high, is a grredy agen
        //or is playing a fixed heuristic policy
        boolean high, greedy, fixed;
        //*****TESTING****** writing all the action data to file
        BufferedWriter p;


        /**
         * Constructor.
         */
        public BPlayer(){
            hand = new Card[5];
            funds = 1000000;
            score = 0;
            hscore = 10;
            val = new double[2][21];
            count = new int[2][21];
            lastaxn = DEAL;
            lastscore = 0;
            greedy = false;
            fixed = false;
            try{
                p = new BufferedWriter(new FileWriter("playerdump.csv"));
                p.write("SCORE ");
                p.write("HITPROB ");
                p.write("STANDPROB ");
                p.write("RNDM ");
                p.write("AXN ");
```

```
                p.newLine();
            }
            catch(Exception ex){
                ex.printStackTrace();
            }
        }


        /**
         * This method uses the softmax method of exploration to pick an action in th
         * current state. After the action is picked, the previous state is updated b
         * on the Temporal-Difference Q-Learning algorithm.
         */
        private void explore(){

            greedy = false;
            fixed = false;

            if(high && hscore <= 21){
                score = hscore;
            }
            else if(high && hscore > 21 && score == hscore){
                high = false;
                score -= 10;
            }
            double hit = Math.exp(val[HIT][score-1]/TAU);
            double stand = Math.exp(val[STAND][score-1]/TAU);

            double sum = hit + stand;

            double hprob = hit / sum;
            double sprob = stand / sum;

            double maxval = Math.max(val[STAND][score-1], val[HIT][score-1]);

            if(lastaxn != DEAL){
                double rd = A * ((G * maxval) - val[lastaxn][lastscore-1]);
                val[lastaxn][lastscore-1] += rd;
            }

            double r = Math.random();

            lastscore = score;

            try{
                p.write(Integer.toString(score)); p.write("␣");
                p.write(Double.toString(hprob)); p.write("␣");
                p.write(Double.toString(sprob)); p.write("␣");
                p.write(Double.toString(r)); p.write("␣");
                if(r > hprob){
                    stand();
                    lastaxn = STAND;
                    p.write("STAND");
                }
```

```
            else{
                hit();
                lastaxn = HIT;
                p.write("HIT");
            }
            p.newLine();
        }
        catch(Exception ex){
            ex.printStackTrace();
        }
        count[lastaxn][lastscore-1]++;
    }


    /**
     * This method provides a greedy agent. This agent is defensive in it behavio
     * as it stands when the value of hitting is equal to the value of standing.
     */
    private void greedy(){

        greedy = true;
        fixed = false;

        if(high && hscore <= 21){
            score = hscore;
        }
        else if(high && hscore > 21 && score == hscore){
            high = false;
            score -= 10;
        }

        lastscore = score;

        if(val[HIT][score-1] > val[STAND][score-1]){
            hit();
            lastaxn = HIT;
        }
        else{
            stand();
            lastaxn = STAND;
        }
        count[lastaxn][lastscore-1]++;
    }


    /**
     * This method provides a fixed heuristic agent for the Reinforcement Learnin
     * agent to learn against.
     */
    private void fixed(){

        greedy = false;
        fixed = true;

```

```
            if(high && hscore <= 21){
                score = hscore;
            }
            else if(high && hscore > 21 && score == hscore){
                high = false;
                score -= 10;
            }

            if(score < 17){
                hit();
            }
            else{
                stand();
            }
        }


        /**
         * This method is used to alllocate final rewards to the last state visited.
         * agents do not recieve rewards, as they are not learning. This reward funct
         * based on the Temporal Differance Q-Learning algorithm
         */
        private void reward(double r){

            if(greedy || fixed){
                return;
            }

            double maxval = Math.max(val[STAND][lastscore-1], val[HIT][lastscore-1]);

            if(lastaxn != DEAL){
                double rd = A * (r + (G * maxval) - val[lastaxn][lastscore-1]);
                val[lastaxn][lastscore-1] += rd;
            }
        }
    }


    //the playing agents
    private BPlayer player, dealer;
    //the pack of card
    private Pack pack;
    //the players' hands
    private JLabel[] pcards, dcards;
    //display of the users funds
    private JLabel fundsLabel;
    //command buttons
    private JButton hit, stand, betButton, clear;
    //panels used to display user components
    private JPanel buttonPanel, playerPanel, dealerPanel;
    //the content pane
    private Container cp;
    //textfield for user to enter amount to bet
    private JTextField betField;
```

```
    //amount the user has bet
    private int bet;
    //used to indicate current position in the players hand
    private int currentCard;
    //flag to indicate if the player is playing or not
    private boolean play;
    //***TEST*** stores number of times each result is recorded.
    private int[] result;
    //***TEST*** stores number of times each result is recorded.
    private int[] deals;
    //***TEST*** records the scores of each player when the draw.
    private BufferedWriter push;
    //***TEST*** stores number of times the player busts at each score.
    private int[] busts;

    /**
     * The main method. Creates a new BlackJack interface.
     * @param args not used
     */
    public static void main(String[] args){new BlackJack();}


    /**
     * Sole constructor. Creates the interface and the players. Begins the learning
     * for the agents(s).
     */
    public BlackJack(){

        super("BlackJack");
        setSize(600,600);
        Dimension d = (Toolkit.getDefaultToolkit()).getScreenSize();
        int h = (int)(d.getHeight());
        int w = (int)(d.getWidth());
        int x = (w - 600) / 2;
        int y = (h - 600) / 2;
        setLocation(x,y);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        pack = new Pack();
        player = new BPlayer();
        dealer = new BPlayer();
        play = true;
        result = new int[6];
        deals = new int[21];
        busts = new int[21];
        currentCard = 0;
        player.score = 0;

        try{
            push = new BufferedWriter(new FileWriter("push.txt"));
        }
        catch(Exception ex){
            ex.printStackTrace();
        }
```

```

        cp = getContentPane();
        cp.setLayout(new BorderLayout());

        hit = new JButton("HIT");
        hit.addMouseListener(new MouseAdapter(){

            public void mouseClicked(MouseEvent e){
                hit();
            }
        });
        hit.setEnabled(false);

        stand = new JButton("STAND");
        stand.addMouseListener(new MouseAdapter(){

            public void mouseClicked(MouseEvent e){
                stand();
            }
        });
        stand.setEnabled(false);

        betField = new JTextField("10");
        betButton = new JButton("BET");
        betButton.addMouseListener(new MouseAdapter(){

            public void mouseClicked(MouseEvent e){
                bet();
            }
        });

        clear = new JButton("CLEAR");
        clear.addMouseListener(new MouseAdapter(){

            public void mouseClicked(MouseEvent e){
                clear();
            }
        });

        buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(5,1));

        buttonPanel.add(hit);
        buttonPanel.add(stand);
        buttonPanel.add(betField);
        buttonPanel.add(betButton);
        buttonPanel.add(clear);

        pcards = new JLabel[5];
        dcards = new JLabel[5];
        fundsLabel = new JLabel(Integer.toString(player.funds));

        playerPanel = new JPanel();
        playerPanel.setLayout(new GridLayout(1,6));
```

```
        dealerPanel = new JPanel();
        dealerPanel.setLayout(new GridLayout(1,6));

        for(int i = 0; i < 5; i++){
            pcards[i] = new JLabel();
            dcards[i] = new JLabel();
            playerPanel.add(pcards[i]);
            dealerPanel.add(dcards[i]);
        }
        playerPanel.add(fundsLabel);
        dealerPanel.add(new JLabel());

        cp.add(dealerPanel, BorderLayout.NORTH);
        cp.add(buttonPanel, BorderLayout.EAST);
        cp.add(playerPanel, BorderLayout.SOUTH);

        setVisible(true);

        int c = 0;

        for(int i = 0; i < 1000; i++){
            bet();
            while(play){
                player.explore();
            }
            while(!play){
                dealer.fixed();
            }
            System.out.println("played " + c);
            c++;
        }

        write(1);

        for(int i = 0; i < 9000; i++){
            bet();
            while(play){
                player.explore();
            }
            while(!play){
                dealer.fixed();
            }
            System.out.println("played " + c);
            c++;
        }

        write(2);

        for(int i = 0; i < 40000; i++){
            bet();
            while(play){
                player.explore();
            }
            while(!play){
```

```
                dealer.fixed();
            }
            System.out.println("played " + c);
            c++;
        }

        write(3);

        result = new int[6];
        busts = new int[21];
        player.count = new int[2][21];

        for(int i = 0; i < 5000; i++){
            bet();
            while(play){
                player.greedy();
            }
            while(!play){
                dealer.fixed();
            }
            System.out.println("played " + c);
            c++;
        }

        write(4);
        writecount();
        try{
            push.flush();
            push.close();
            player.p.flush();
            player.p.close();
        }
        catch(Exception ex){
            ex.printStackTrace();
        }
    }


    /**
     * *****TESTING METHOD****** <br/>
     * Writes all the data collected to .csv files, which can be imported into an
     * spreadsheet package for further data analysis.
     * @param x the number tag for the file.
     */
    private void write(int x){

        try{
            BufferedWriter bw = new BufferedWriter(new FileWriter("results" + x + ".c

            bw.write("Player Bust");
            bw.write("\t");
            bw.write("Dealer Bust");
            bw.write("\t");
            bw.write("Both Bust");
```

```
            bw.write("\t");
            bw.write("Player wins");
            bw.write("\t");
            bw.write("Player Loses");
            bw.write("\t");
            bw.write("Push");
            bw.newLine();

            for(int i = 0; i < result.length; i++){
                bw.write(Integer.toString(result[i]));
                bw.write("\t");
            }

            bw.flush();
            bw.close();

            bw = new BufferedWriter(new FileWriter("pvals" + x + ".csv"));

            bw.write("score");
            bw.write("\t");
            bw.write("stand");
            bw.write("\t");
            bw.write("hit");
            bw.write("\t");
            bw.newLine();

            for(int i = 0; i < player.val[0].length; i++){
                bw.write(Integer.toString(i));
                bw.write("\t");
                for(int j = 0 ; j < player.val.length; j++){
                    bw.write(Double.toString(player.val[j][i]));
                    bw.write("\t");
                }
                bw.newLine();
            }

            bw.flush();
            bw.close();

            bw = new BufferedWriter(new FileWriter("dvals" + x + ".csv"));

            bw.write("score");
            bw.write("\t");
            bw.write("stand");
            bw.write("\t");
            bw.write("hit");
            bw.write("\t");
            bw.newLine();

            for(int i = 0; i < player.count[0].length; i++){
                bw.write(Integer.toString(i));
                bw.write("\t");
                for(int j = 0 ; j < player.count.length; j++){
                    bw.write(Double.toString(player.count[j][i]));
```

```java
                    bw.write("\t");
                }
                bw.newLine();
            }

            bw.flush();
            bw.close();

            bw = new BufferedWriter(new FileWriter("busts" + x + ".csv"));
            for(int i = 0; i < busts.length; i++){
                bw.write(Integer.toString(i));
                bw.write("\t");
                bw.write(Integer.toString(busts[i]));
                bw.newLine();
            }

            bw.flush();
            bw.close();
        }
        catch(Exception ex){
            ex.printStackTrace();
        }
    }

    public void writecount(){

        try{
            BufferedWriter bw = new BufferedWriter(new FileWriter("count.txt"));

            for(int i = 0; i < 2; i++){
                for(int j = 0; j < 21; j++){
                    bw.write(new Integer(player.count[j][i]).toString());
                    bw.write(" ");
                }
                bw.newLine();
            }
            bw.flush();
            bw.close();
        }
        catch(Exception ex){

        }
    }

    /**
     * This method is used to update the GUI to the most current state. Every call o
     * methods redraws all elements on screen and then repaints the JFrame.
     * @see clear
     */
    private void refresh(){

        if(play){
            for(int i = 0; i < 5; i++){
                if(player.hand[i] != null){
```

```
                    pcards[i].setIcon(player.hand[i].toImage("Normal"));
                }
            }
            dcards[0].setIcon(dealer.hand[0].toImage("Normal"));
            dcards[1].setIcon(dealer.hand[1].toBackImage("Trickster"));

        }
        else{
            for(int i = 0; i < 5; i++){
                if(dealer.hand[i] != null){
                    dcards[i].setIcon(dealer.hand[i].toImage("Normal"));
                }
            }
        }

        fundsLabel.setText(Integer.toString(player.funds));
        ((JComponent)(cp)).revalidate();
        repaint();
    }


    /**
     * Bets money for the player. The amount to be bet is parsed from betField. betF
     * is set to 10 by default. This is so the AI does not have to be made to bet, as
     * can simply call the method.
     */
    private void bet(){

        bet = Integer.parseInt(betField.getText());

        if(bet <= player.funds){
            player.funds -= bet;
            deal();
            hit.setEnabled(true);
            stand.setEnabled(true);
            betButton.setEnabled(false);
        }
        refresh();
    }


    /**
     * Deals a new hand. The player/dealer hands are reinitialised and the pointer a
     * all reset. Both scores are then updated.
     * @see hit
     * @see stand
     */
    private void deal(){

        dealer.hand[0] = pack.nextCard();
        dealer.score += dealer.hand[0].getBJPoints();
        if(dealer.hand[0].getValue().equals("A")){
            dealer.high = true;
            dealer.hscore = dealer.score + 10;
```

```
        }
        dealer.hand[1] = pack.nextCard();
        dealer.score += dealer.hand[0].getBJPoints();
        if(dealer.hand[1].getValue().equals("A")){
            dealer.high = true;
            dealer.hscore = dealer.score + 10;
        }
        dealer.lastaxn = BPlayer.DEAL;
        if(dealer.high && dealer.hscore <= 21){
            dealer.score = dealer.hscore;
        }

        player.hand[0] = pack.nextCard();
        player.score += player.hand[0].getBJPoints();
        if(player.hand[0].getValue().equals("A")){
            player.high = true;
            player.hscore = player.score + 10;
        }
        player.hand[1] = pack.nextCard();
        player.score += player.hand[1].getBJPoints();
        if(player.hand[1].getValue().equals("A")){
            player.high = true;
            player.hscore = player.score + 10;
        }
        if(player.high && player.hscore <= 21){
            player.score = player.hscore;
        }
        deals[player.score -1]++;
        player.lastaxn = BPlayer.DEAL;
        currentCard = 2;
    }


    /**
     * This method adds a Card to the active BPlayer's hand. The pointer is moved to
     * next Card. The player is forced to stand if they exceed a score of 21 or have
     * cards.
     * @see deal
     * @see stand
     */
    private void hit(){

        if(play){
            player.hand[currentCard] = pack.nextCard();
            player.score += player.hand[currentCard].getBJPoints();
            if(player.hand[currentCard].getValue().equals("A")){
                player.high = true;
                player.hscore += player.hand[currentCard].getBJPoints();
            }
            refresh();
            currentCard++;
            if(player.score > 21 || currentCard == 5){
                stand();
            }
```

```
        }
        else{
            dealer.hand[currentCard] = pack.nextCard();
            dealer.score += dealer.hand[currentCard].getBJPoints();
            if(dealer.hand[currentCard].getValue().equals("A")){
                dealer.high = true;
                dealer.hscore += dealer.hand[currentCard].getBJPoints();
            }
            refresh();
            currentCard++;
            if(dealer.score > 21 || currentCard == 5){
                stand();
            }
        }
    }


    /**
     * Stands for the agent that called the method. If it is the player, then the pla
     * is passed on to the dealer. If it is the dealer, then the result is calculate
     * and the hand ends
     * @see deal
     * @see hit
     */
    private void stand(){
        if(play){
            play = false;
            currentCard = 2;
            refresh();
        }
        else{
            if(player.score > 21 && dealer.score <= 21){
                //player bust
                player.reward(-20);
                result[0]++;
                busts[player.lastscore -1]++;
            }
            else if(player.score <= 21 && dealer.score > 21){
                //dealer bust
                player.funds += bet * 2;
                player.reward(10);
                result[1]++;
            }
            else if(player.score > 21 && dealer.score > 21){
                //dealer and player bust
                player.reward(-10);
                result[2]++;
                busts[player.lastscore -1]++;
            }
            else if(player.score > dealer.score){
                //player wins
                player.funds += bet * 2;;
                player.reward(10);
                result[3]++;
```

```
            }
            else if(player.score < dealer.score){
                //player loses
                player.reward(-10);
                result[4]++;
            }
            else{
                //push
                player.funds += bet;
                player.reward(5);
                result[5]++;
                try{
                    push.write(Integer.toString(player.score));
                    push.write(" ");
                    push.write(Integer.toString(dealer.score));
                    push.newLine();
                }
                catch(Exception ex){
                    ex.printStackTrace();
                }
            }
            bet = 0;
            clear();
            play = true;
        }
    }


    /**
     * Clears all GUI elements and resets all values to thier intitial values.
     * @see refresh
     */
    private void clear(){

        playerPanel.removeAll();
        dealerPanel.removeAll();

        for(int i = 0; i < 5; i++){
            player.hand[i] = null;
            dealer.hand[i] = null;
            pcards[i] = new JLabel();
            dcards[i] = new JLabel();
            playerPanel.add(pcards[i]);
            dealerPanel.add(dcards[i]);
        }

        playerPanel.add(fundsLabel);
        bet = 0;
        currentCard = 0;
        player.score = 0;
        dealer.score = 0;
        dealer.high = false;
        player.high = false;
        fundsLabel.setText((new Integer(player.funds)).toString());
```

```
        betButton.setEnabled(true);
        hit.setEnabled(false);
        stand.setEnabled(false);
        repaint();
    }
}
```

Appendix D

Final System

D.1 Source Code

D.1.1 Face

See Appendix C.1.1

D.1.2 Suit

See Appendix C.1.2

D.1.3 Card

See Appendix C.1.3

D.1.4 Pack

See Appendix C.1.2

D.1.5 Shoe

```
//Adding Shoe to the SKCards package
package SKCards;


/**
 * This object represents a Dealer's shoe. This is a number of decks
 * of cards in a closed container. There is a specaial cut card, which
 * is inserted randomly into the Shoe. When this is dealt, the shoe is
 * discarded and a new one is used. Shoes are used to counter Card counting.
 *
 * @author Saqib A Kakvi
 * @version 2.0A
 * @since 2.0A
 * @see SKCards.Pack Pack
 * @SK.bug none
 */
public class Shoe extends Pack{

    //the position of the cuttoff
```

```
    int cutoff;
    //the size of the Shoe
    int size;

    /**
     * Constructor. Creates a pack with the specified parameters by calling
     * super(d,j). The cutoff Card's position is calculated and is stored.
     * No extra Card is added to the Pack, so the size remains the same.
     *
     * @param d the number of decks
     * @param j if there are jokers in the deck
     */
    public Shoe(int d, boolean j){

        super(d,j);
        size = super.size();
        calculateCutoff();
    }


    /**
     * Calculates the cutoff Card position. The Card will be in the range of
     * 0-85% of the Shoe size. Hence a maximum of of 85% of the Pack is
     * played.
     */
    private void calculateCutoff(){
        cutoff = new Double(Math.random() * 100 * 0.85).intValue() * size;
    }


    /**
     * Checks if the current Card is the cutoff Card. If it is then a new Pack
     * is created and the cutoff Card's position is recalculated. Then the
     * method returns the Card from super.nextCard().
     *
     * @override SKCards.Pack nextCard
     * @return the next Card in the Shoe
     */
    public Card nextCard(){

        if(currentCard == cutoff){
            newPack();
            calculateCutoff();
        }
        return super.nextCard();
    }
}
```

D.1.6 BJPlayer

```
//Adding Player to SKCards
package SKCards;


```

```
/*
 * Importing the ArrayList class
 * and the Arrays class
 * and the swing/awt components
 */
import java.util.ArrayList;
import java.util.Arrays;
import java.awt.*;
import javax.swing.*;


/**
 * This is a player in the BlackJack game
 *
 * @author SK
 * @version 2.0A
 * @since 1.0C
 */
public class BJPlayer extends Player{

    protected int score, hscore;

    protected Card[] hand;

    protected BJRules rules;

    boolean high;

    /**
     * Creates a new player object with a given details.
     *
     * @param id the player name
     * @param cash the amount of money the player starts with
     * @param setPack the pack of cards to be used in the game
     * @param bck the card back the player uses
     * @param cs the card set used by the player
     */
    public BJPlayer(String id, int cash, String bck, String cs,
        BJRules r){

        rules = r;
        funds = cash;
        name = id;
        back = bck;
        cardSet = cs;
        high = false;
        score = 0;
        hscore = 10;

        hand = new Card[5];
    }

    /**
     * Basic constructor. Simply takes in the rules. Used for
```

```
     * BJRLPlayer
     *
     * @param r the rules
     */
    public BJPlayer(BJRules r){

        rules = r;
    }

    /**
     * Creates a new hand of cards for the current player and one for the compter
     * and then deals two cards to each hand
     */
    public void newHand(){
        hand = new Card[5];
        score = 0;
    }

    /**
     * Hits the player's hand
     *
     * @param currentCard the current card
     * @param nextCard the card to be added
     */
    public int hit(int currentCard, Card nextCard){

        hand[currentCard] = nextCard;

        score += rules.getBJPoints(hand[currentCard]);

        if(hand[currentCard].getValue().equals("A")){
            high = true;
            hscore += rules.getBJPoints(hand[currentCard]);
        }

        if(high && hscore <= 21){
            score = hscore;
        }
        else if(high && hscore > 21){
            if(hscore == score){
                score -= 10;
            }
            high = false;
        }

        return score;
    }


    /**
     * Returns the players current score
     *
     * @return score
     */
```

```
    public int getScore(){
        return score;
    }

    /**
     * Ends the current hand
     */
    public void endHand(){

        int comp = 0;
        int player = 0;


        System.out.println("player --->" + player);
        System.out.println("computer --->" + comp);

        if(comp > 0 || player > 0){
            if(comp > player){
                JOptionPane.showMessageDialog(null, "HOUSE WINS!!!");
                rules.display();
            }
            else if(player > comp){
                JOptionPane.showMessageDialog(null, "PLAYER WINS!!!");
                funds += bet;
                rules.display();
            }
            else if(comp == player){
                JOptionPane.showMessageDialog(null, "PUSH!!!");
                funds += bet;
                rules.display();
            }
            else{
                JOptionPane.showMessageDialog(null, "NO RESULT FOUND");
            }
        }
        else{
            JOptionPane.showMessageDialog(null, "NO RESULT FOUND");
            rules.display();
        }
    }


    /**
     * Clears all the hands and clears the interface
     */
    public void clear(){

        for(int i = 0; i < hand.length; i++){
            hand[i] = null;
        }

        rules.clear();
    }

```

```java

    /**
     * Gets the player's current hands
     * @return the player's hands
     */
    public Card[] getHand(){

        return hand;
    }
}
```

D.1.7 BJUI

```java
//adding BJUI to SKCards
package SKCards;

/*
 *
 */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * This is the User Interface for the Blackjack Game.
 * It allows the human player to access the functions
 * ans shows the current state of the game.
 *
 * @author SK
 * @version 2.0A
 * @since 2.0A
 * @SK.bug none
 */
public class BJUI extends JFrame implements UI{

    //the players' hands
    private JLabel[] pcards, dcards;
    //display of the users funds
    private JLabel fundsLabel;
    //command buttons
    private JButton hit, stand, bet, clear, exit;
    //panels used to display user components
    private JPanel buttonPanel, playerPanel, dealerPanel;
    //the content pane
    private Container cp;
    //textfield for user to enter amount to bet
    private JTextField betField;
    //the game rules
    BJRules rules;

    /**
     * Creates a new UI. Takes in the rules for the current game.
     * This allows the two objects to communicate.
```

```
     *
     * @param r the Rules for the game
     */
    public BJUI(BJRules r){

        super(”BlackJack”);
        setSize(600,600);
        Dimension d = (Toolkit.getDefaultToolkit()).getScreenSize();
        int h = (int)(d.getHeight());
        int w = (int)(d.getWidth());
        int x = (w - 600) / 2;
        int y = (h - 600) / 2;
        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
        setLocation(x,y);

        rules = r;

        cp = getContentPane();
        cp.setLayout(new BorderLayout());

        hit = new JButton(”HIT”);
        hit.addMouseListener(new MouseAdapter(){

            public void mouseClicked(MouseEvent e){
                rules.hit();
            }
        });
        hitEnabled(false);

        stand = new JButton(”STAND”);
        stand.addMouseListener(new MouseAdapter(){

            public void mouseClicked(MouseEvent e){
                rules.stand();
            }
        });
        standEnabled(false);

        betField = new JTextField(”10”);
        bet = new JButton(”BET”);
        bet.addMouseListener(new MouseAdapter(){

            public void mouseClicked(MouseEvent e){
                rules.bet(Integer.parseInt(betField.getText()));
            }
        });
        betEnabled(true);

        clear = new JButton(”CLEAR”);
        clear.addMouseListener(new MouseAdapter(){

            public void mouseClicked(MouseEvent e){
                clear();
            }
```

```
        });

        exit = new JButton("EXIT");
        exit.addMouseListener(new MouseAdapter(){

            public void mouseClicked(MouseEvent e){
                BJUI.this.dispose();
            }
        });

        buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(6,1));

        buttonPanel.add(hit);
        buttonPanel.add(stand);
        buttonPanel.add(betField);
        buttonPanel.add(bet);
        buttonPanel.add(clear);
        buttonPanel.add(exit);

        pcards = new JLabel[5];
        dcards = new JLabel[5];
        fundsLabel = new JLabel();

        playerPanel = new JPanel();
        playerPanel.setLayout(new GridLayout(1,6));
        dealerPanel = new JPanel();
        dealerPanel.setLayout(new GridLayout(1,6));

        for(int i = 0; i < 5; i++){
            pcards[i] = new JLabel();
            dcards[i] = new JLabel();
            playerPanel.add(pcards[i]);
            dealerPanel.add(dcards[i]);
        }
        playerPanel.add(fundsLabel);
        dealerPanel.add(new JLabel());

        cp.add(dealerPanel, BorderLayout.NORTH);
        cp.add(buttonPanel, BorderLayout.EAST);
        cp.add(playerPanel, BorderLayout.SOUTH);

        setVisible(true);
    }


    /**
     * Refreshes the GUI to show the current situation.
     * @override display
     */
    public void display(){

        boolean play = rules.getPlay();

```

```java
        Card[] phand = rules.getPlayerHand(rules.PLAYER);

        Card[] chand = rules.getPlayerHand(rules.DEALER);

        String cs = rules.getPlayerCardSet(rules.PLAYER);
        String bk = rules.getPlayerBack(rules.PLAYER);

        for(int i = 0; i < phand.length; i++){
            if(phand[i] != null){
                pcards[i].setIcon(phand[i].toImage(cs));
            }
        }
        dcards[0].setIcon(chand[0].toImage(cs));

        if(play){
            dcards[1].setIcon(chand[1].toBackImage(bk));
        }
        else{
            for(int i = 1; i < chand.length; i++){
                if(chand[i] != null){
                    dcards[i].setIcon(chand[i].toImage(cs));
                }
            }
        }

        fundsLabel.setText(rules.getPlayerFunds(rules.PLAYER).toString());
        repaint();
    }


    /**
     * Displays a new hand on the GUI.
     * @override newDisplay
     */
    public void newDisplay(){

        Card[] phand = rules.getPlayerHand(rules.PLAYER);

        Card[] chand = rules.getPlayerHand(rules.DEALER);

        String cs = rules.getPlayerCardSet(rules.PLAYER);
        String bk = rules.getPlayerBack(rules.PLAYER);

        for(int i = 0; i < phand.length; i++){
            if(phand[i] != null){
                pcards[i].setIcon(phand[i].toImage(cs));
            }
        }

        dcards[0].setIcon(chand[0].toImage(cs));
        dcards[1].setIcon(chand[1].toBackImage(bk));

        fundsLabel.setText(rules.getPlayerFunds(rules.PLAYER).toString());
        repaint();
```

```
    }


    /**
     * Clears the GUI
     * @override clear
     */
    public void clear(){

        System.out.println();

        playerPanel.removeAll();
        dealerPanel.removeAll();

        for(int i = 0; i < 5; i++){
            pcards[i] = new JLabel();
            dcards[i] = new JLabel();
            playerPanel.add(pcards[i]);
            dealerPanel.add(dcards[i]);
        }

        playerPanel.add(fundsLabel);
        betEnabled(true);
        hitEnabled(false);
        standEnabled(false);
        repaint();
    }

    /**
     * Gets the amount bet from the betField.
     *
     * @return bet
     */
    public int getBet(){
        return(Integer.parseInt(betField.getText()));
    }

    /**
     * Enables or disables the hit button
     *
     * @boolean hit button state
     */
    public void hitEnabled(boolean en){
        hit.setEnabled(en);
    }

    /**
     * Enables or disables the stand button
     *
     * @boolean stand button state
     */
    public void standEnabled(boolean en){
        stand.setEnabled(en);
    }
```

```

    /**
     * Enables or disables the bet button
     *
     * @boolean bet button state
     */
    public void betEnabled(boolean en){
        bet.setEnabled(en);
    }
}
```

D.1.8 BJRules

```
//adding BJRules to SKCards
package SKCards;

/*
 * importing the swing classes.
 */
import javax.swing.*;

/**
 * The rules of the Blackjack game.
 *
 * @author SK
 * @version 2.0A
 * @since 2.0A
 * @SK.bug none
 */
public class BJRules extends Rules{

    public static final int PLAYER = 0;
    public static final int DEALER = 1;

    private boolean play;
    int currentCard, pscore, dscore, bet;

    public BJRules(String id, int cash, Pack setPack, String bck, String cs,
        SKMain m){

        main = m;

        ui = new BJUI(this);

        pack = setPack;

        players = new Player[2];

        players[PLAYER] = new BJPlayer(id, cash, bck, cs, this);
        players[DEALER] = new BJRLPlayer(this);

        play = true;
    }

```

```java
    public void newGame(){

        clear();
        ((BJUI)ui).hitEnabled(true);
        ((BJUI)ui).standEnabled(true);
        ((BJUI)ui).betEnabled(false);
        players[PLAYER].newHand();
        players[DEALER].newHand();
        ((BJPlayer)(players[PLAYER])).hit(0, pack.nextCard());
        pscore = ((BJPlayer)(players[PLAYER])).hit(1, pack.nextCard());
        ((BJPlayer)(players[DEALER])).hit(0, pack.nextCard());
        dscore = ((BJPlayer)(players[DEALER])).hit(1, pack.nextCard());
        currentCard = 2;
        display();
    }

    public void endGame(){

        Card[] phand = ((BJPlayer)players[PLAYER]).getHand();
        Card[] chand = ((BJPlayer)players[DEALER]).getHand();

        for(int i = 0; i < phand.length; i++){
            if(phand[i] != null){
                System.out.print(phand[i].toString());
            }
        }
        System.out.println(" PLAYER " + pscore);

        for(int i = 0; i < chand.length; i++){
            if(chand[i] != null){
                System.out.print(chand[i].toString());
            }
        }
        System.out.println(" DEALER " + dscore);

        if(pscore > 21 && dscore <= 21){
            JOptionPane.showMessageDialog(null, "Player is bust!");
            ((BJRLPlayer)(players[DEALER])).reward(10);
        }
        else if(dscore > 21 && pscore <= 21){
            JOptionPane.showMessageDialog(null, "Dealer is bust!");
            players[PLAYER].setFunds(bet*2);
            ((BJRLPlayer)(players[DEALER])).reward(-10);
        }
        else if(pscore > 21 && dscore > 21){
            JOptionPane.showMessageDialog(null, "Both bust!");
            players[PLAYER].setFunds(bet);
            ((BJRLPlayer)(players[DEALER])).reward(-10);
        }
        else if(pscore > dscore){
            JOptionPane.showMessageDialog(null, "Player wins!");
            players[PLAYER].setFunds(bet*2);
            ((BJRLPlayer)(players[DEALER])).reward(-10);
        }
```

```java
        else if(pscore == dscore){
            JOptionPane.showMessageDialog(null, "Push!");
            players[PLAYER].setFunds(bet);
            ((BJRLPlayer)(players[DEALER])).reward(5);
        }
        else{
            JOptionPane.showMessageDialog(null, "Player loses!");
        }
        clear();
        play = true;
    }

    public void hit(){
        if(play){
            pscore = ((BJPlayer)(players[PLAYER])).hit(currentCard, pack.nextCard());
            currentCard++;
            if(pscore > 21 || currentCard == 5){
                display();
                stand();
            }
        }
        else{
            dscore = ((BJPlayer)(players[DEALER])).hit(currentCard, pack.nextCard());
            currentCard++;
            if(pscore > 21 || currentCard == 5){
                display();
                stand();
            }
        }
        display();
    }

    public void stand(){
        System.out.println(play);
        if(play){
            play = false;
            display();
            currentCard = 2;
            while(!play){
                ((BJRLPlayer)players[DEALER]).learn();
            }
        }
        else{
            play = true;
            endGame();
        }
        display();
    }

    public void bet(int amt){

        if(amt <= players[PLAYER].getFunds()){
            bet = amt;
            players[PLAYER].setFunds(-1 * bet);
```

```
        }
        else{
            JOptionPane.showMessageDialog(null, "INSUFFICENT FUNDS!!!");
        }
        newGame();
        display();
    }

    public int getBJPoints(Card c){

        int points = c.getPoints();

        if(points == 10 || points == 11 || points == 12 || points == 13){
            return 10;
        }
        else{
            return points;
        }
    }

    public boolean getPlay(){
        return play;
    }

    public Card[] getPlayerHand(int p){

        return ((BJPlayer)players[p]).getHand();
    }
}
```

D.1.9 RLPlayer

```
//adding BJRLPlayer to SKCards
package SKCards;

/*
 * importing the IO classes
 */
import java.io.*;


/**
 * The RL Player for the Blackjack game.
 *
 * @author SK
 * @version 2.0A
 * @since 2.0A
 * @SK.bug none
 */
public class BJRLPlayer extends BJPlayer{

    private final double ALPHA = 0.6;
    private final double GAMMA = 0.75;
    private final double TAU = 0.5;
```

```java

    private static final int HIT = 1;
    private static final int STAND = 0;
    private static final int DEAL = -1;

    private double[][] val;
    private int lastaxn, lastscore;

    /**
     * Sole constructor. Takes in the rules and passes it to
     * super(r).
     *
     * @param r the rules
     */
    public BJRLPlayer(BJRules r){

        super(r);

        val = new double[2][21];

        try{
            BufferedReader br = new BufferedReader(new FileReader("rlvals.txt"));

            String data = br.readLine();
            int i = 0;

            while(data != null && i < val[0].length){

                System.out.println(data);

                String[] ds = data.split(",");

                val[STAND][i] = Double.parseDouble(ds[0]);
                val[HIT][i] = Double.parseDouble(ds[1]);

                data = br.readLine();
                i++;
            }
        }
        catch(Exception ex){
            ex.printStackTrace();
        }
    }

    /**
     * Starts a new hand.
     */
    public void newHand(){

        lastaxn = DEAL;
        super.newHand();
    }

    /**
```

```
     * Plays a greedy policy, but updates the values as it plays
     */
    public void learn(){

        if(score > 21){
            return;
        }

        System.out.println("HIT = " + val[HIT][score -1]);
        System.out.println("STAND = " + val[STAND][score -1]);

        double maxval = Math.max(val[STAND][score -1], val[HIT][score -1]);

        if(lastaxn != DEAL){
            double rd = ALPHA * ((GAMMA * maxval) - val[lastaxn][lastscore -1]);
            val[lastaxn][lastscore -1] += rd;
        }

        lastscore = score;

        if(val[HIT][score -1] > val[STAND][score -1]){
            rules.hit();
            lastaxn = HIT;
            System.out.println("HIT");
        }
        else{
            rules.stand();
            lastaxn = STAND;
            System.out.println("STAND");
        }
    }


    /**
     * Allocates the final reward to the agent.
     *
     * @param r the reward
     */
    public void reward(double r){

        double maxval = Math.max(val[STAND][lastscore -1], val[HIT][lastscore -1]);

        if(lastaxn != DEAL){
            double rd = ALPHA * (r + (GAMMA * maxval) - val[lastaxn][lastscore -1]);
            val[lastaxn][lastscore -1] += rd;
        }
    }
}
```

Appendix E

Sample Blackjack Game

The following notation is used in the sample game:
[] - a face down card
— - Separation of two hands
Cards - Face values - 2 -9 ⇒ 2 - 9, T ⇒ 10, J ⇒ Jack, Q ⇒ Queen, K ⇒ King
Suits - D - Diamonds, C - Clubs, H - Hearts, S - Spades
e.g. KS ⇒ King of Spades, 2D ⇒ 2 of Diamonds, TH ⇒ 10 of Hearts

```
Dealer AC []
Bet
Player [] []
Player bets $10, and takes an insurance
Dealer AC []
Bet $10 $10
Player AD AH
Player splits
Dealer AC []
Bet $10 $10 $10
Player AD QS | AH AS
(BLACKJACK)
Player stands on hand 1, splits hand 2
Dealer AC []
Bet $10 $10 $10 $10
Player AD QS | AH 2C | AS 4S
Player hits hand 2
Dealer AC []
Bet $10 $10 $10 $10
Player AD QS | AH 2C 4D | AS 4S
Player hits hand 2
Dealer AC []
Bet $10 $10 $10 $10
Player AD QS | AH 2C 4D KD| AS 4S
(Ace now counts ``low" (1))
Player hits hand 2
Dealer AC []
Bet $10 $10 $10 $10
Player AD QS | AH 2C 4D KD 5S | AS 4S
(BUST!)
Player doubles down on hand 3 (counting ace ``high")
```

```
Dealer AC []
Bet $10 $10 $10 $20
Player AD QS | AH 2C 4D KD 5S | AS 4S 6C
Player stands on hand 3, dealer plays by turning over his second card
Dealer AC JC (BLACKJACK!)
Bet $10 $10 $10 $20
Player AD QS | AH 2C 4D KD 5S | AS 4S 6C
```

The dealer has hit a Blackjack, so the player receives their $10. The player loses the $10 from hand 2 because they went bust. As for hands 1 and 3, the dealer and player have tied. However some casinos enforce the rule the “House wins a push”. Under this rule, the player would also lose the $30. If the house treats pushes normally, then the player will get his $30 back. Although there are many different variations on the rules, known as “house rule”, they are all well known and accepted by the players.

Bibliography

Parlett, D. (1994). *Teach Yourself Card Games*. Hodder Headline Plc, Euston Road, London, NW1 3BH.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA.