Team RL--; ValueIteration++; #Vegas

Elliott Brossard (snowden@cs), Kellen Donohue (kellend@cs), Zach Stein (steinz@cs)

Goals

The goal of this project is to create an AI-based agent capable of winning at the standard configuration of blackjack.

System Design

Our system consists of a blackjack simulator and a variety of agents that can use it.

Terminology and Rules

In blackjack, cards with a count of two to nine have that as their value, ten through king have a value of 10, and the ace has a value of either 1 or 11. A *blackjack* is a hand of two cards in which one has a value of 10 and the other is an ace.

There are two types of *counts* of hands related to the sum of the values of the cards. The *soft count* of a hand is its maximum value; in blackjack, aces can be treated as having either a value of 11 or of 1, so a hand with an ace and a six has a soft count of 17. Conversely, the *hard count* of a hand is its minimum value, so the ace-six hand has a hard count of 7. When treating the ace as having a value of 11 would put the total count of the hand above 21, the soft count is taken to be the hard count, so in the case of a hand with a nine, a five, and an ace, the soft count and hard count would both be 15.

Prior to the commencement of a round in blackjack, each player (agent) makes a bet for the round, which determines how much the agent will win from or lose to the dealer at the conclusion of the round. Each agent initially receives two cards, which are revealed to the other agents. The dealer receives two cards, although only one is revealed to the agents. The agents take turns deciding which action to make next until all of them have finished the round. The actions that an agent may take include *hitting*, which is where the agent requests an additional card; *standing*, which is where the agent chooses to finish the round; *doubling down*, which is where the agent doubles its bet on a hand

containing two cards and requests an additional card, after which it finishes the round; and *splitting*, which is where the agent separates a hand of two cards of the same value into separate hands, each of which receives another card and has the same bet as the initial hand. If the hard count of an agents hand is greater than 21 after executing an action, the agent *busts* and finishes the round.

Once all agents have finished in the round, the dealer plays out its own hand in accordance with a small set of rules. If the soft count of the dealer's hand is greater than 21 the hard count is less than 17, or else if the soft count is less than 17, the dealer hits and takes an additional card. Otherwise the dealer stands, which concludes play in the round.

At the conclusion of a round, the status and soft counts of the agents' and the dealers' hands determine how chips, or whatever the underlying currency of betting is, are exchanged. If an agent's hand has bust, then the agent forfeits the bet associated with the hand; if an agent's hand is a blackjack, then the agent receives the amount of the bet associated with the hand from the dealer; and otherwise the soft counts of the hands determine whether the hand ends in a win, a tie, or a loss. In the event of a tie, the agent neither loses nor gains any currency.

Simulator

There are Deck, Hand, Card, and Agent data structures, which represent their real world equivalents. The Deck can be created with any number of sets of cards, suits, and values, and it supports taking and returning cards. When a card is returned to the deck, it enters the inactive card pile and is eventually shuffled with the rest of the cards. Hands track the money bet on the hand, which actions are available to the player, and the soft and hard count.

Agents

The blackjack simulator that we created contains a number of different agents, some of which do not rely on any form of artificial intelligence and some of which make decisions based on learning. The non-learning agents comprise DealerAgent, HumanAgent, ReflexAgent, StandingAgent, and NoBustAgent, while the learning agents comprise QLearningAgent, CountLearningAgent, and ValueIterationAgent.

DealerAgent

The DealerAgent acts in accordance with the dealer rules as outlined in the section on the game rules: If the soft count of the dealer's hand is greater than 21 the hard count is less than 17, or else if the soft count is less than 17, the dealer hits and takes an additional card. Otherwise the dealer stands.

HumanAgent

The HumanAgent encodes no logic on decision-making itself; instead, it relies on console input to decide how to act for a given hand. The performance of the HumanAgent depends entirely on the external agent controlling it.

ReflexAgent

The ReflexAgent is the result of encoding the table given on the blackjack Wikipedia page[1]. Few tables that we found online cited the origin of the actions that they suggested for states, so we simply picked one that looked promising and verified that it worked reasonably well in practice, which this one did. The ReflexAgent makes a decision based on whether its hand contains an ace, whether the hand is a pair, whether the hand contains only two cards (for doubling down), the hard count of the hand, and the value of the dealer's visible card.

StandingAgent

The StandingAgent simply stands whenever it is given the choice of action, no more and no less.

NoBustAgent

The NoBustAgent is a minor improvement on the StandingAgent. Rather than stand all the time, the NoBustAgent hits when it has a guarantee of not busting, i.e. when the hard count of a hand is 11 or less.

ValueIterationAgent

The blackjack ValueIterationAgent uses value iteration for training. Over a sequence of *n* iterations, the ValueIterationAgent visits all states in its state space and performs Bellman backup to update the q-value for each. This agent represents states in the form (isDone, isFirst, isDoubleDown, hasAce, hardCount, dealerSoftCount).The initial three elements are all boolean values, so there are $2^3 * 580 = 4640$ states, including some that are impossible to reach. Here isDone signifies whether the agent has finished playing a particular hand, which is the case if the agent stands, doubles down, or busts. isFirst signifies whether the agent's hand contains only two cards and hence an action will be the first action for the hand; this is necessary to determine whether doubling down is permitted. isDoubleDown signifies whether the agent's hand has been doubled down, hasAce signifies whether the hand contains an ace, hardCount is the hard count of the agent's hand, and dealerSoftCount is the soft count of the dealer's visible card (whether to store the soft or hard value of the dealer's card was arbitrary).

The reward function for a given (state, action, transitionState) tuple evaluates to 0 if isDone is false in transitionState and otherwise is non-zero. If isDone is true, the ValueIterationAgent simulates the dealer agent's play and returns a reward based on how the agent's state compares to the dealer's final state (win, loss, or tie) and the bet associated with the hand. The ValueIterationAgent reward function breaks from the standard reward model of blackjack in one respect, however, which is that the penalty for busting is doubled. In experiments, the ValueIterationAgent performed much better this modification than it did previously. The justification for the change is that busting guarantees a loss regardless of the dealer's final count, so the agent should prioritize not busting over maximizing the soft count of a hand.

[1] http://en.wikipedia.org/wiki/Blackjack#Blackjack_strategy

The ValueIterationAgent also does not support splitting, whereas the QLearningAgent does. It is not feasible to encode splitting into the ValueIterationAgent's state space and transition model without exploding the number of states to explore, so it makes more sense to rule out splitting entirely. In practice, too, the ValueIterationAgent performed well in comparison to the ReflexAgent even though its actions were limited to hitting, standing, and doubling down.

QLearningAgent

The blackjack QLearning Agent is similar to the Q learning agent implemented in the Pacman project, from which our project borrows some code. It learns an approximation of the expected reward of taking an action in a given state by making moves with a particular degree of randomness during many training rounds. During "real" rounds, it performs the action with the highest expected reward. The QLearningAgent represents states with tuples of the form (playerHardCount, playerHasAce, dealerCardValue), and the significance of each is self-explanatory. The elements can take on the values 2-31, true or false, and 1-10 respectively, so there are 29 * 2 * 10 = 580 states.

CountLearningAgent

The CountLearningAgent is an extension of the QLearningAgent. It operates by encoding additional information about the game state in the features used to group states. The idea is that additional features can offset the fact each state is seen less times. A common method human players use to increase their performance in blackjack is counting cards. This essentially is guessing the relative proportion of face cards left in the deck. When this proportion is high players are more likely to bust, and may alter their strategy accordingly. Using this inspiration the count learning agent utilizes the Zen Count method[2]. This method, while difficult for humans, is quite simple for a computer program such as this one.

The CountLearningAgent uses a state representation similar to that of the QLearning agent, adding a count variable to the tuple that is constrained to the range (-10, 10) based off the Zen Count of the deck. Since the QLearningAgent's state representation yielded 580 states, the CountLearningAgent's state representation yields 580 * 20 = 11,600 states.

AceCountLearningAgent

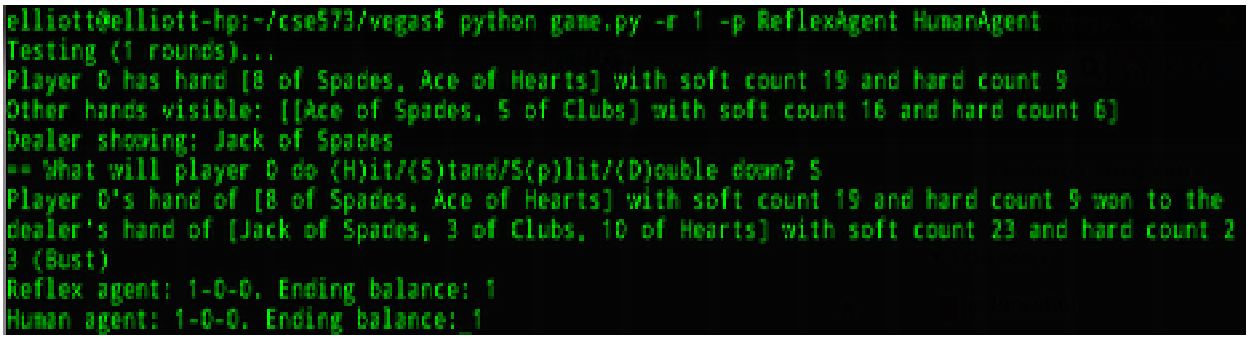
The AceCountLearningAgent is an extension of the CountLearningAgent, pushing the tradeoff in CountLearningAgent even further. The AceCountLearningAgent keeps a side count of the number of aces remaining in the deck, another popular strategy amongst humans[3]. While keeping multiple counts would be difficult for a human, it's easy for a computer. The real challenge in choosing additional features is not enlarging the state space too much. Since the CountLearningAgent has a state space with a size of 11,600 and the count of the number of aces can range from 0 to 4 on one deck, the AceCountLearningAgent has a state space of 11,600 * 5 = 58,000 states.

---

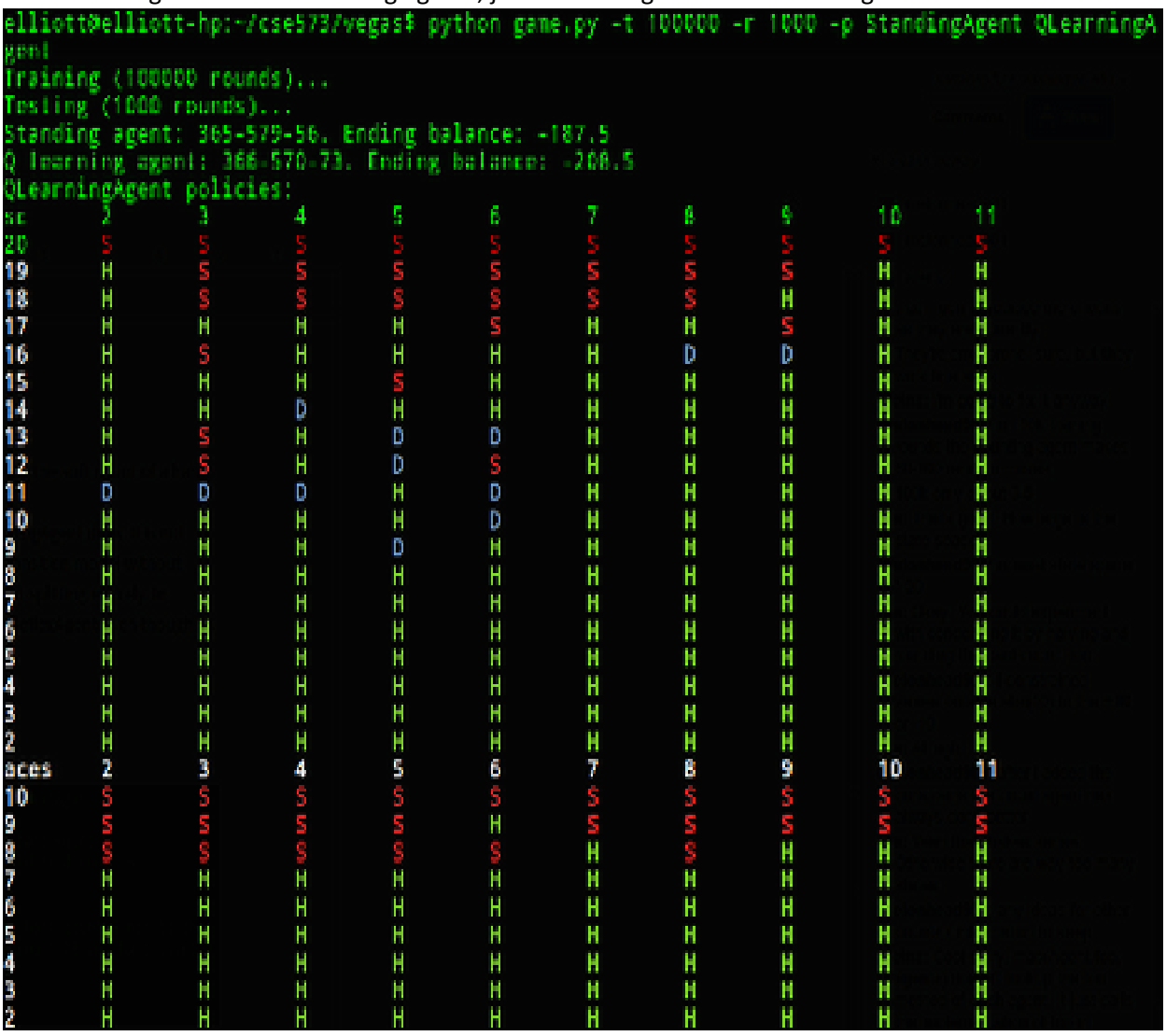
[2] http://www.blackjackchamp.com/card-counting/zen-count/

[3] http://www.qfit.com/blackjack-side-counts.htm

Sample Screenshots

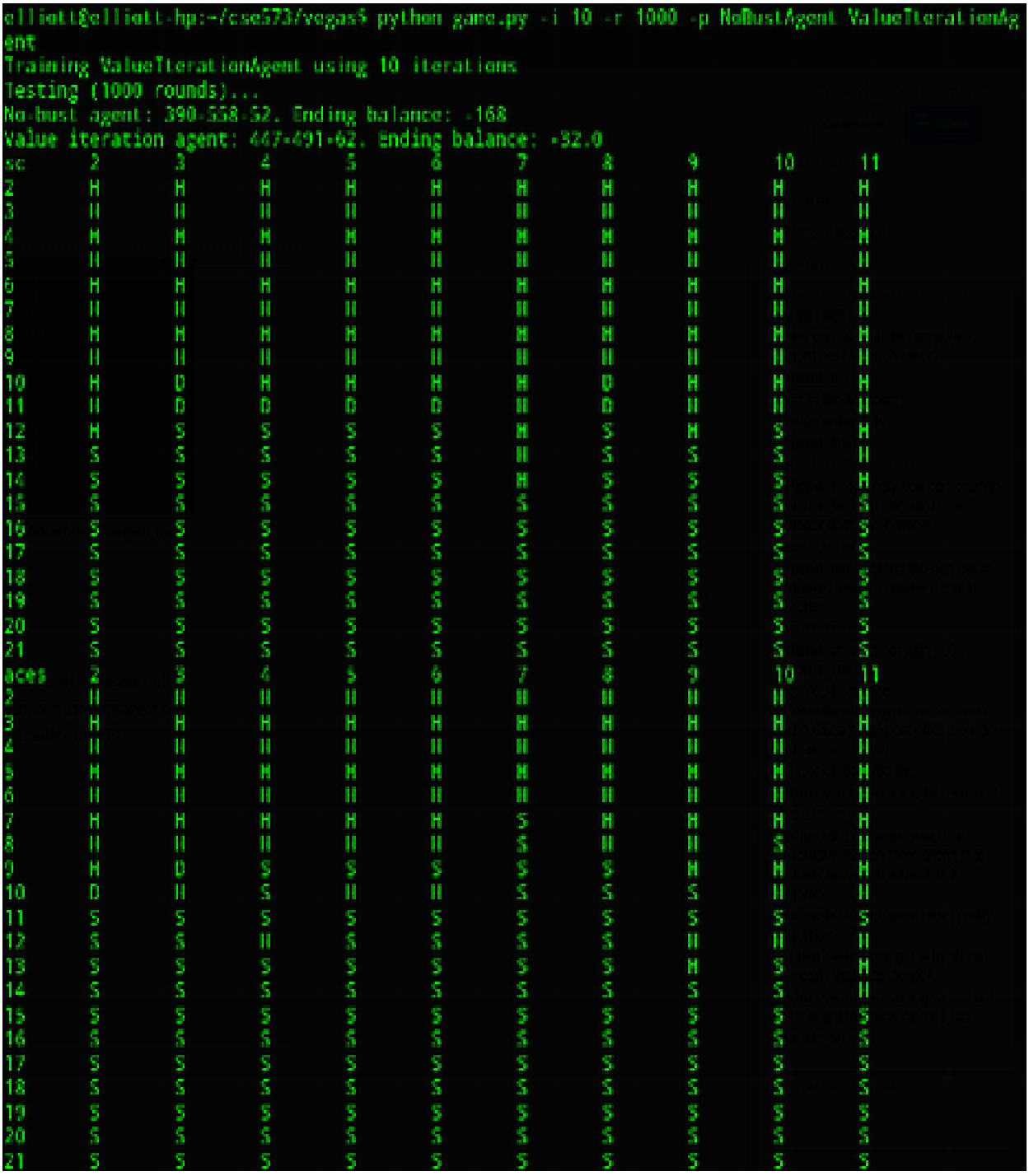


A one-round game with no learning agents, just a ReflexAgent and a HumanAgent.



A thousand-round game with a StandingAgent and a QLearningAgent. The learned policies for the QLearningAgent are also printed. The column on the left is the soft count for the hand, while the row at the top is the value of the dealer's visible card. The table shows "H" for hit, "S" for stand, and "D"

for double down. Splitting is not shown, although the QLearningAgent does encode that as a potential action when a hand contains a pair of cards.



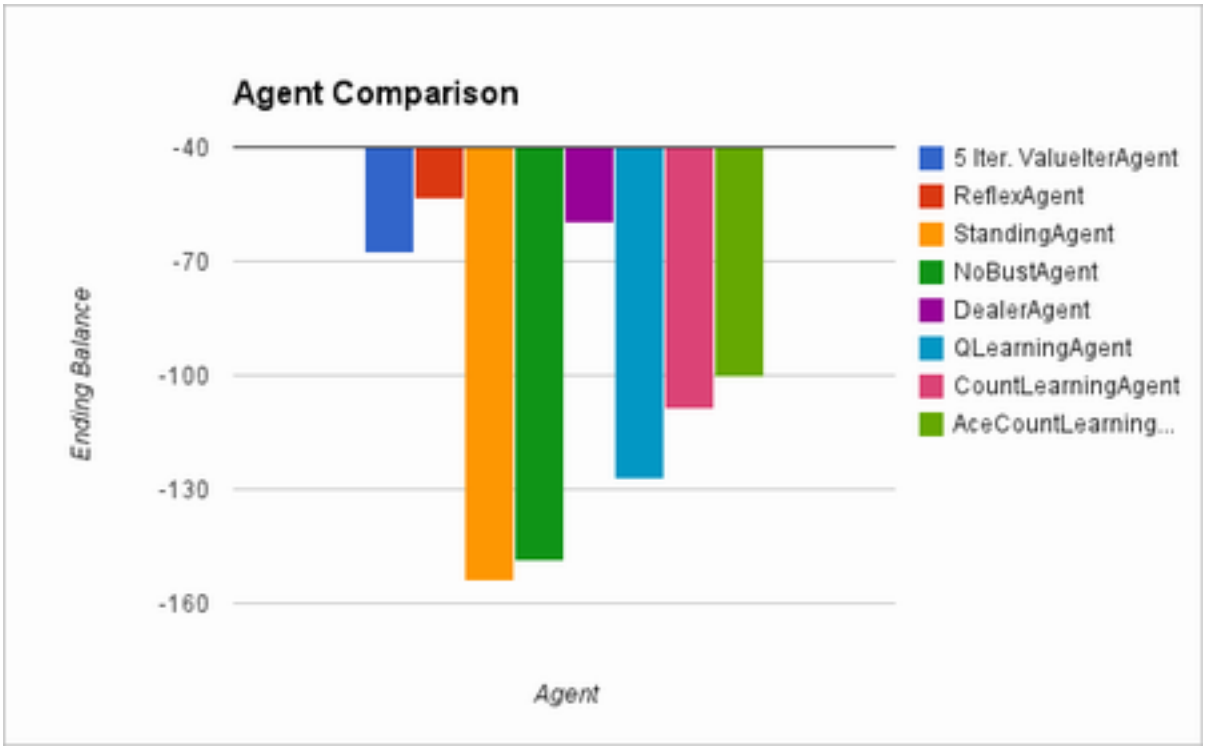
A thousand-round game with a NoBustAgent and a ValueIterationAgent. The ValueIterationAgent used 10 iterations over the state space for its initial training.

Experiments

Our experiments consisted of testing our agents against the ReflexAgent, which acted as the control. We typically ran 1,000 test trials, after any necessary training or value iteration, and used the agent's ending balance as a measure of success. Each trial consisted of a betting round with a bet of 1 associated with each hand.

Results

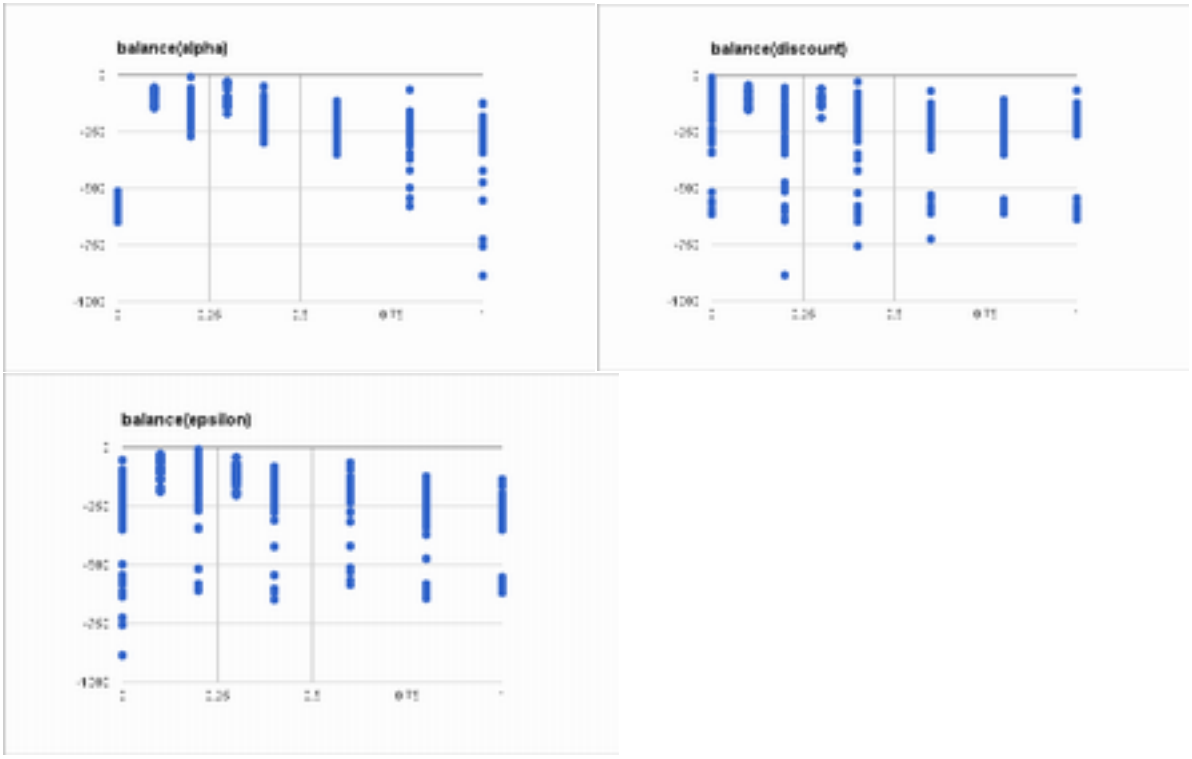
We compared the various non-learning and learning agents by conducting 100 series of 1000-round games for each agent. For the ValueIterationAgent, we ran the tests using a single-iteration training session, a three-iteration training session, and a five-iteration training session, but there was no statistical significance in the results, which is likely due to the infrequency of hands involving three or more cards. We chose alpha, discount, and epsilon factors for the QLearningAgent, CountLearningAgent, and AceCountLearningAgent in accordance with some optimizations that we ran to discover the values that maximized the agents' performance against the dealer.



Pictured in the chart above are the results of the experiments; the mean earnings across all 1000-round games are given for each agent. The ReflexAgent clearly performs best, which is not unexpected since it is likely the optimum encoding of decisions for the various states as determined by whichever group created it. The DealerAgent does surprisingly well for how simple its state-action mapping is, since one would think that its threshold-based hit-stand tradeoff would do nowhere near as well as the complex encoding of the ReflexAgent.

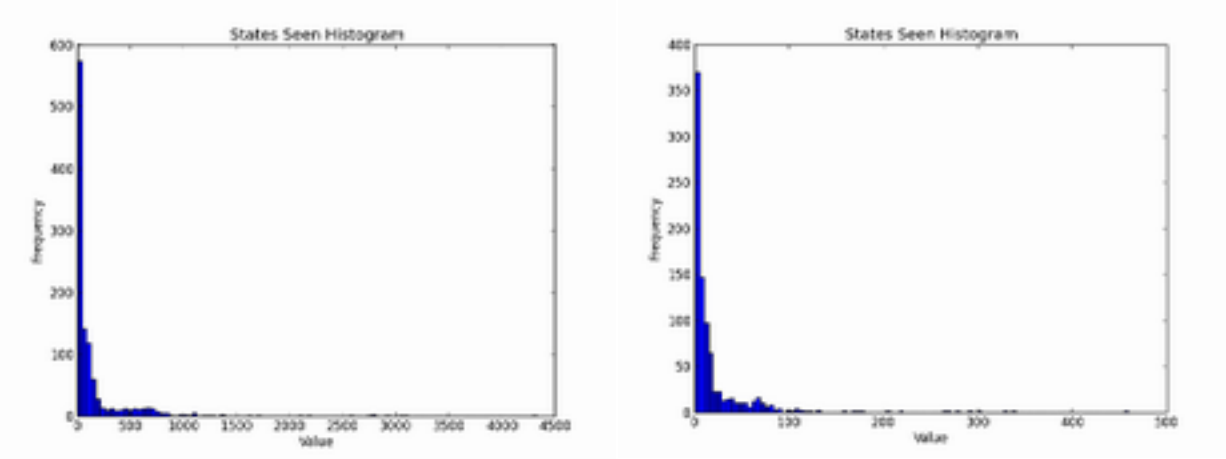
None of the QLearning agents performed especially well, however, which was disappointing. We wrote a script to compare different learning rates (alphas), discount factors (discounts), and random move

chances (epsilons) for our QLearningAgent, which is in compare_q_learners.py. Ending balances for 10,000 training rounds and 1,000 testing rounds are plotted below as a function of each parameter. The two parameters not included in each graph account for the vertical distribution seen at each x value.



Making random moves infrequently performs better than making random moves frequently, since this allows the agent to explore sequences of actions in a more directed manner. High discount factors don't seem to help much, and low learning rates tend to outperform higher learning rates. Values around 0.2 tend to do well for each parameter.

We were surprised how rarely some of our states were visited by the QLearningAgent. The following graphs show the frequency for the number of times a state was visited during the training process for 100,000 and 10,000 training rounds.



Exploring the most likely states more is probably a good idea, but hundreds of states are seen only a handful of times. This presents difficulty to our QLearningAgent because it must act based on situations it knows very little about. Additionally, some states are not seen at all during training. However, this doesn't seem to be much of a problem in practice, since those states don't tend to show up in testing either.

The ValueIterationAgent, conversely, visits all states during initialization, so it is better able to make a decision for any state it encounters.

Always standing (even when it's not possible to bust) is a surprisingly good strategy, yielding an ending balance of -149 in our experiments. In the end this was the worst by far of any agent, but while we were building and tuning our agents, it did outperform some versions.

With value iteration modeling using infinitely many decks one can get more accurate results in some cases, but additional information does not improve performance. This was possible, however, with the QLearningAgent, where adding the counting mechanism to the CountLearningAgent did noticeably improve performance. A similar change would not be possible to the ValueIterationAgent because it has no feasible way to add the concept of cards remaining in the deck without vastly enlarging the state space.

Future Work

In the future we would like to implement more sophisticated algorithms, such as UCT. It would be interesting as well to see what additional features could be used during training, if we had access to some long term computing cluster. While the AceCountLearningAgent needed over 100,000 trials to provide sufficient state coverage, if a cluster were always available perhaps counts could be kept for all ranks.

Conclusions

Our q learning agent was able to beat the trivial standing and no bust agents. Counting cards did even better, but was still significantly worse than our value iteration agent. Additionally, our dealer agent did slightly better than even our value iteration agent. The reflex agent (from Wikipedia) did outperform the dealer agent, but it still did not make a profit. Based on the QLearningAgent when counting was added perhaps we could improve it further by adding more features and using more computing power.

We will be staying away from casinos for the time being.

Appendices

Who Did What

Elliott wrote or collaborated on the implementations for the deck, cards, game logic, Markov decision process, HumanAgent, NoBustAgent, ReflexAgent, ValueIterationAgent, betting, doubling, and splitting.

Kellen wrote or collaborated on the implementations for the hand, game state, game logic, policy analysis, ReflexAgent, and CountLearningAgent. Kellen also made most of the slides for the presentation to the class.

Zach wrote or collaborated on the implementations for the win-loss-tie tracking, StandingAgent, QLearningAgent, QLearner comparison script, policy analysis, and team name mangling.

See also the commit history for the Github repository linked below for more detailed information about specific pieces of code.

Use of External Code

The only external code used in this project was some of the framework for Markov decision processes, value iteration, and q learning from the Pacman project. For the MDP and value iteration agent, no actual implementation was used, just the function signatures. We wrote everything else ourselves, including a suite of unit tests for the various classes.

Getting Started

The source code for our project is publicly available and can be obtained from https://github.com/melonhead901/vegas. The only requirement is an installation of Python, which is available for most major operating systems. To acquire the code using Git, simply execute `git clone https://github.com/melonhead901/vegas.git` from the command line.

To run the simulator, execute `python game.py` with the command line arguments that are shown by running the program with the --help flag. An example command to trigger a run involving a QLearningAgent that trains on 10,000 rounds and then tests on 1,000 is:

```
python game.py -t 10000 -r 1000 -p QLearningAgent ReflexAgent
```

To run a simulation with the ValueIterationAgent, one must specify the number of iterations to use with the -i parameter, e.g.:

```
python game.py -i 50 -r 1000 -p ValueIterationAgent ReflexAgent
```

The agents with which the simulation can run are DealerAgent, HumanAgent, ReflexAgent, StandingAgent, NoBustAgent, QLearningAgent, CountLearningAgent, and ValueIterationAgent.

All agents can be run and compared with:

```
python game.py -t 10000 -r 1000 -i 10 -p StandingAgent NoBustAgent
QLearningAgent ValueIterationAgent ReflexAgent CountLearningAgent
AceCountLearningAgent
```