Blackjack Benchmark

Using a classic, strategy-oriented card game to test performance of various programming languages.

Karamchandani, Sidhant skaramch@ucsc.edu
Chauhan, Prajan pchauhan@ucsc.edu
Hamilton, Ian iahamilt@ucsc.edu
Lieu, Benjamin blieu@ucsc.edu
Singh, Yugraj ysingh@ucsc.edu

March 21, 2014

Abstract:

Blackjack is a classic card game played at casinos worldwide. In this game, all players sitting a table compete against the house (or casino, or the dealer, etc.), instead of competing with each other. Each player is dealt two cards to begin with, and they take turns one by one either "hitting", "standing", or "splitting". The goal is to reach as close as possible to the total card sum of 21, but not any greater. All face cards count as a value of 10, and Aces can be either an 11 or a 1. If a player beats the dealer's score, he wins whatever amount of money he bet. If a player loses to the dealer's score OR gets a total value higher than 21 (a bust), the house wins and takes the player's bet money. A "hit" move means the player asks for another card. The "stay" move means the player stands at the score where he currently is and play proceeds to the next player. The "split" move can be called only when the player has two of the same card. His hand splits into two and he resumes normal play, but with two hands now. Basic strategy for this game involves considering the number of decks in play, looking at the cards currently on the table (everything is face up except one of the dealer's cards), and deciding whether the next card drawn will be low, high, or in between.

Our goal was to implement a functioning Blackjack game in which artificial intelligence players programmed in different languages all play at the Blackjack table, utilizing their programmed strategies to get as many wins as possible. By keeping track of each player's wins, losses, busts, and draws, as well as how much time they take on average to think of a move, we can benchmark the performance of these different languages. Furthermore, we can go into a more in-depth comparison of these languages by analyzing the lines of code written for each player, the amount of time taken to write each player, and the pros and cons of writing each one. Specifically, we wanted to compare imperative and functional programming languages.

Overview:

The application starts with the activation of a central GameServer that manages the flow of the entire Blackjack game, the state of the table, and communication with players. The GameServer program acts as a listener that listens for programs trying to connect to it or for moves that are being sent. The GameServer uses a secondary Deck class that implements the functionality of a deck(s) of cards, including a Card subclass and the drawing of a shuffled card. The GameServer also uses a secondary Handler class that manages connections between the server and clients. Each AI player acts as one of these clients. These player clients all connect to the server and join the game, and then proceed to play the game as the GameServer deals out cards and asks players for their moves; essentially a client acts as an advisor to the actual players, meaning a single client could be advising multiple players on whether to hit or stay. To communicate data between programs written in different languages, we implemented sockets where clients can communicate their moves to the server. Based on information received from each player, GameServer will update the board for the clients. In fact, we use a core string called GameState which keeps track of every player on the table (in order of their turn), and all of their respective cards. This is the main string that is sent to each player when it is his turn. Details will be explained below.

Languages Used:

Java

The Java programming language was developed in 1995 by James Gosling at sun microsystems and is designed to let developers "write once, run anywhere". In order to make Java robust and platform independent, the language runs on the Java Virtual Machine (JVM). The JVM is a process virtual machine that takes compiled Java code (bytecode) and executes the programming instructions on the virtual machine. Along with making Java platform independent, the language was developed with several paradigms in mind. Java is a general purpose object-oriented, concurrent, generic and imperative programming language that has most of its syntax derived from other imperative languages like C and C++. Unlike C and C++, Java doesn't support low-level functionalities such as pointer and memory allocation, but the language does have automatic memory management.

By removing most of the low-level functionalities, the Java language creates a level of abstraction where a programmer don't have to be concerned by dealing with allocation and deallocation of memory. The abstraction continues with the paradigm of Java being a "pure" object-oriented language. In Java, everything is a object except primitive types. Objects in java are created using classes because Objects are instance of classes. Classes in Java are used to encapsulate data and methods that allow the programmers to manipulate the data in the instance of that class. This allows a level of abstraction that is not available in low-level languages like C. Another key feature of the Java that allows for more abstraction is inheritance. Inheritance is when a new class is derived from an existing class. The new class created using inheritance can inherit the implementation of the existing class. The use of inheritance allows for more flexible and less redundant code. An example of this is when changes to code are need in a project. Instead of modifying similar functions in multiple classes, only the in code in super class in which all other classes inherit from needs to be changed. This creates less code in the project to manage while allowing multiple objects to have the same methods.

Even though the Java programing language provides key features like inheritance, when you start programing in Java, you quickly notice how verbose the language is. As anyone who has programmed in Java can tell you, theres a lot of boilerplate code to do a task which can be done in less lines of code in other languages like python. This can make it difficult for people are not experienced with Java when starting to program. Another down side of Java is the JVM. Since Java bytecode is run on the virtual machine, Java code is not going to be as fast as other compiled languages. Java also has a very large memory footprint, but as memory gets cheaper and larger this becomes less of a factor.

Besides the boilerplate code, and being slower than other compiled languages, Java provides a vast library and documentation for nearly any programming need. Java may not the easiest language to pick up when compared to python, its library and error system makes it easy and fairly fast to debug. Java has robust error system that provides descriptive error messages that can be used to debug. And since Java is such a widely used language, there are multiple resources like stack overflow and debuggers available that when a programmer encounters a problem or error in their code, they can easily find out why there's an error and what caused it.

Ruby

Ruby is a general purpose dynamic typed and interpreted language designed by Yukihiro Matsumoto in 1995 with multiple paradigms in mind. Some of the paradigms that Ruby is designed with are object-oriented, imperative, functional and reflective.

One of the unique design of the Ruby language is how object oriented programming design is implement. A key features of Ruby that is different then most languages is that everything is an object, including primitives values. Since everything in the language is a object, variable that reference primitive types are actually references to an object that represent primitive types. And to create an object in Ruby, we construct classes and class functions similar to most object-oriented class typed languages. The main difference is the hierarchy and scope of these classes. When you define functions, variables and objects outside of the classes, they become part of the root/self-object and are global scope. Ruby also supports inheritance, in which classes can inherit functions and variables from a parent class.

In Ruby, all member variable of classes are private and can only be assessed with getter and setter methods. But unlike getter and setter methods in other languages, Ruby uses metaprogramming to access these variables declared in the scope of a class. Another syntax quirk about Ruby is that a block statement for loops, if-else and function definitions end with the keyword “end” . Besides that syntax, the Ruby language’s syntax is very similar to python except that indentation is not important.

The language also provides an easy and readable syntax along with automatic memory management. Another feature of the language is that Ruby has built-in data structures like arrays,lists. stacks and maps. And since Ruby is dynamically typed and interpreted, its easy to write code and not have to worry about laying down all the ground for storing data and values. Since Ruby is interpreted, it will be slower than compiled languages. But for most applications, Ruby is an easy language to pick up and program in.

Python

Python was conceived in the late 1980s and its implementation started in December 1989 by Guido van Rossum at CWI in the Netherlands as a successor to the ABC language capable of exception handling and interfacing with the Amoeba operating system. Python is intended to be a highly readable language. It is designed to have an uncluttered visual layout, frequently using English keywords where other languages use punctuation. Furthermore Python has a smaller number of syntactic exceptions and special cases than C or Pascal. Python uses whitespace indentation, rather than curly braces or keywords, to delimit blocks; a feature also termed the off-side rule. An increase in indentation comes after certain statements; a decrease in indentation signifies the end of the current block. It features a dynamic type system, automatic memory management, and has a comprehensive standard library. Python is a language that supports multiple programming paradigms, such as object-oriented, imperative and functional programming or procedural styles. Python is often used as a scripting language however it is sufficient for a wide range of non-scripting contexts.

Python is very easy to learn and use. We were able to learn how to write in Python in less than an hour. It was also easy to debug in the language because most people code in Python and we were able to find where the problem was really quick. Better yet, we can always use the PyDev part of Eclipse to debug our Python programs. The language was very easy to write with because it uses dynamic type classing which allowed us to declare variables on the fly and not get any errors or faults. We only had problems with the socketing when we were coding, but that was because we had no experience with socket programming.

Python is a very easy language to learn and is perfect for beginner programmers. This language is very forgiving and is not as complicated as some other languages like Java or C++, because of its dynamic type system there is no need to declare something in a complicated manner. This language can be used for web development, scientific and numeric computing, software development, and educational purposes.

Haskell

Being the only non imperative language in the selection, Haskell requires a unique way of anticipating how to write code efficiently and elegantly. Because Haskell is almost entirely executed in functions, it is vital to understand that the program works as a series of data transformations, with each function often taking an input, manipulating it, then passing it on to another function. This process however has very few similarities to it's imperative language counterparts. Haskell is referentially transparent, meaning that all functions can be replaced with the value they return without affecting what is outside of the function. It may be that the programmer's entire computational portion of the program happens in one function that uses many other subfunctions for performing the several tasks that it needs to. Another way to look at the functions in this language is that they have no side effects. One never has to worry about global or static variables outside the scope of the function being manipulated when passing a parameter to it. This proves to be very useful for debugging code because if the programmer can find the function causing errors, then most likely the error has already been found.

In Haskell, all data objects are immutable, making it difficult to update a variable or state. A function often has to create a new object if they want to change values to the parameter object; however, with lazy evaluation, all values are only computed when needed. For instance, assume we have a function that takes a list of numbers and returns a list of every number incremented by one. If we repeatedly called this function on a list several times, we would only make one pass through that entire list, instead of making a pass for every call. This provides wonders for efficiency and also allows one to create an infinite list.

Haskell's compiler can be a pain to work with. This is due to the language's strict typeclass system. Although it's type inference is excellent in that it allows the program to know exactly what type an expression yields, it also understands when you may be attempting illegal activity such as inserting a string into a list of integers. This most commonly results in spending more time coding but ultimately results in less time debugging. One will find that if a function compiles, it most likely works.

Haskell tends to be very concise and easy to read, especially because the amount of lines of code are much less than that of imperative pieces of code. Obviously this depends on perspective. As most programmers begin their careers with imperative languages, a purely functional language can seem quite dense. For example, for larger scale functions, the variables used to hold values from the smaller functions are written in a where clause after the main computation. Unless they are written in a neat order and it is clear what the several smaller scoped functions do, it may take time to fully understand what is going on. Finally, one of the most unorthodox aspects of haskell from an imperative perspective is that there are no for or while loops in haskell. This often makes recursion the best strategy to use for iterative methods.

Methodology:

Deck.java

First of all, the Deck class includes a very important subclass within its fields. It is the Card subclass, which includes fields such as a card's value, its symbol, and its suit. For example, a King of Hearts has a value of 10, a symbol "King", and a suit "Hearts". Using this card subclass, the Deck has methods which create a full deck using for loops to insert instances of the Card subclass into a singly linked list. Based on the number of players that GameServer has connecting to it, the server will decide how many decks to create. If there are two decks needed, for example, the singly linked list will end up being 104 elements long. Shuffling the deck isn't even necessary: whenever GameServer needs to draw a card, we simply generate a random number from 0 to the length of the deck, and draw a card from the linked list, which effectively removes it from the linked list and delivers the card information to GameServer simultaneously. By removing random elements from the linked list, we simulate a real deck in which cards are removed from the top of a shuffled deck as they are drawn. We also include a method that erases the entire linked list and creates a fresh one for whenever the GameServer decides a Blackjack game is done and another should begin.

Handler.java

This file is in charge of maintaining connections between the GameServer and player clients. Whenever a client wants to connect to our listening server, we create a new instance of this class. This handler is responsible for keeping track of fields such as the socket currently being used for the connection, the name of the client, and the number of players of each client. It is also responsible for the PrintWriter and the BufferedReader which manage writing strings from the server to the client, and reading strings from the client to the server, respectively. This data transfer is all done through the specific socket created by each handler. At first, this handler simply prints out a greeting string to a client, and reads in the client's name and number of players. After this point, the handler actually sends out a GameState string (discussed below) to each client, and reads in the client's move for that turn. Last but not least, the handler keeps track of players' move times. Specifically, it starts a timer before sending the state of the table to a client, and measures the time elapsed when it receives a reply for a move from that client. These times are stored in a global list, which is also discussed below.

GameServer.java

There are three complex functions within GameServer: connecting clients, updating the table, and running endgame operations. When the server starts, it asks how many games of Blackjack the players should play. It then asks how many different clients or language advisors are going to be connecting in on the game. Once it receives this information, the server starts listening for incoming connections on a specified port. Once an connection is received an new instance of the Handler class is created for that client. Now GameServer no needs to know how many players are being advised by that client (language). After the players' input, the server generates that many players on the table. Each of these players are added to a player list and based on the size of the list, the number of decks to be used is determined. After this the player order is mixed up at random so that no one advisor is advising for a substantial chunk of the game, plus the dealer player is added at the end of this queue since the dealer always goes last. Then the game starts: all the players are simply waiting for their initial two cards so we use the deal function to get random cards out of the Deck class for each player. These cards are written down on a string called GameState which keeps track of which player has what cards. The GameServer then asks the current player whether he wants to hit or stay by sending the GameState through the socket to the current player. The time it takes to receive a response of a "hit" or "stay" is measured by the handler and stored in a global array for the time it takes for every player's move that is advised by that specific language; the answer time for every player's every turn who is being advised by Python is recorded, for example. The GameServer keeps asking this current player for a move until it receives a ‘stay’ answer or the player “busts” and his sum goes over 21. GameState then moves on to the next player and repeats the process with the rest of the players. The GameState then goes through each player’s cards and calculates whether each player won, lost, busted, or had a draw. It adds this calculation to each languages win/loss/bust/draw ratio. A win is if a player is closer to 21 (BlackJack) than the dealer, a loss is if the dealer is closer to 21 than the player or if the player busted, a bust is if a player’s sum goes over 21, and a draw is if a player has the same sum as the dealer. Also if the dealer busts, than any player who does not bust wins.

javaAI.java

The Java AI is similar in structure to the other AIs, and is comprised of three parts: the listener , the parser and the game “AI”. The first part is the listener socket object, which established a connection with GameServer in a try catch block in the main function. Once a connection is made, the run function is called. The run function listens to what the server sends to the javaAI and reads the game state into an arraylist of strings. The game state information is then sent the parser. The parser goes through the arraylist that the game state is stored in and retrieves the players cards, dealers cards and the deck size into private variables that are used by the nextMove method (the “AI”). The next move method determines what total amount of the player cards and if it contains an ace. Once that is determined, the function goes through a series of if else statements to decide on hit or stay and then returns a string values containing “hit” or “stay”

haskell.hs

The Haskell file, like the other AIs, was implemented in three major parts: the client, the parser, and the AI. The IO do block for the client was simple. It created a handle for the incoming connection from the server and used it for all reading and writing operations. It accumulated the game state line by line from the server until the end of the input before sending it as one big string to the parser.

The parser is made up of a handful of functions that each take a piece of the information, and manipulate it before passing it off to another function. The order of types that the information changes into follows this pattern: String->[[String]]->[Card]. First, we read the all of the information as a String and converted it to a list of lists containing strings. This is done by splitting everything at the new line character. Each inner list is either a player name, or a card following the owner in a sequence with a length of at least two. Then, for all the strings that resemble card values, we create a card object and transfer each to a list for later. We end up having a list of the player’s cards, a list of the dealers cards, and a list of everyone’s cards.

The AI deals with each of those lists individually. It needed to know what the player and dealer has to make a decision and it also counts the high and low cards on the table in order to make an educated guess as to what type of card will be drawn next. High cards get a negative one value while low cards get a positive. This sum of these values was mostly used when the player had a difficult hand to hit on such as a hard 14, 15 or 16.

pythongame.py

The Python AI has three main functions, the parser, the socket and the game AI with all of its helper functions. The parser for the string we received from the game server is the same as the Java version, it looked through each line the game server gave it then sent the cards for its turn to a helper function that would calculate the total card number and pass it to the game AI. The difference between Python and the other game AI’s is the socket function. The socket function in Python had a lot of trouble reading in the right amount of strings for each GameState. It would either read in too much or too little of the string causing the parser to throw errors. Eventually we were able to fix the problem, but the process took a while to figure out what was wrong with the socket. We had to read in the entire string instead of only reading in sections at a time. The game AI was nothing special and all we used were some delimiters to check the card number and hit or stay depending on how high or low the number was.

rubyAI.rb

The ruby AI follows a similar structure to the other AI's. The main difference is that socket connect and reading listening to the server happen in the main function. After the listener reads the information, the parser function is called on the game state read in from the server. The game state is stored in an array of strings. The parser function iterates through the array of strings and then retrieves and stores the current players cards. Once the players cards are retrieved, the next move function is called. The next move function looks at the players total cards and then decides through a series of if else statement if the string "hit" or "stay" should be returned.

DealerAI.java

The java Dealer is structured exactly the same as the Java AI. The only difference between the two is that the Dealer's nextmove function is structured a little differently. The dealer determines if there is an ace in the dealers set of cards and we calculated the total sum of the dealers cards. If the dealer is in between a soft 9 and a soft 11 and under both a soft and non soft 17, the dealer will return "hit". In any other case the dealer will return "stay".

Conclusion:

Problems Encountered

As expected, one major problem we had was getting the socket connections between our server and clients to work smoothly. Establishing a connection wasn't so much of a problem as was sending information that every AI's client was compatible with. We had close to no problems when testing whether an AI could connect along the server's port but that was a small success of our data transaction efforts. We too often ran into problems in which a client couldn't properly accept the information passed along the socket. For instance, when we attempted to read in the GameState for our Python file, we would either read in too much or too little. When reading input from a socket in Python, the recv function, which is used for reading socket input is not guaranteed to retrieve the message from the server. Haskell and Java had a similar problem in which the files first attempted to check for the end of file. Because it was the server's job to keep all clients listening until every game had been simulated, we needed a different solution to this problem. We had to include a final line to every GameState that explicitly tells the player that the there is nothing more to read. In Ruby we encountered very few problems with parsing and IO.The problems we did encounter with Ruby are all do to syntaxical reasons.

But with more solutions, came more problems, and therefore more frustration. When debugging we would find extra new line characters placed in odd places, most often at the beginning or end of the GameState. This caused all sorts of problems for our parsers. Because the Haskell parser relies on list manipulation, a new line in the front of the string would mean our code would attempt to access the head of an empty list. We also found that when the server was listening for clients to connect to, we were not able to implement a time window in which the server would listen then stop listening. The program would simply keep waiting for an infinite amount of time till a player would connect. To get past this problem we have GameServer ask how many connections there will be so that it can count the clients connected and no longer listen for more.

The final problem we had is present in all programming experiences: human error. There were many times we couldn't figure out why certain aspects of our program weren't working and it would mainly be due to an over sight of what was being added to GameState, a small indexing error, or even a missing try catch block. Some of these problems caused us hours of re analyzing our code and even using the step function in the eclipse IDE. Overall, once these problems were found and fixed the game worked smoothly and uninterrupted by errors.

Comparison of Languages, Lessons Learned

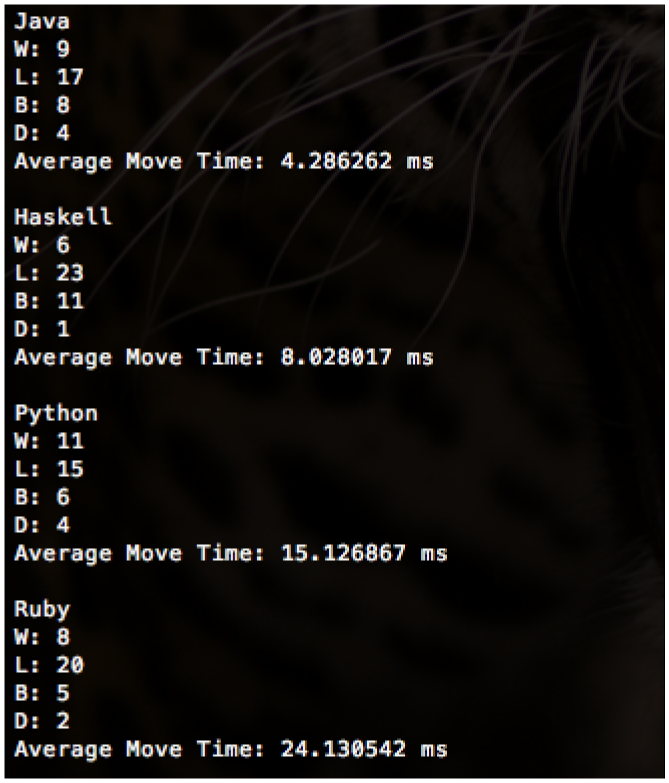
The similarities between Java and Ruby are that both languages are object oriented and use classes to create objects and both have a garbage collector. Other than that, Ruby and Java are completely different languages. Ruby is a dynamically typed language where everything is an object including numbers, and all member variables are private that must be accessed using methods. Java on the other hand is a statically typed, compiled language with its syntax heavily influenced by C and C++. In terms of programming in these two languages, Java took a larger amount of time and debugging. While programming in Ruby, we were able to quickly type my code and have it running within twenty minutes. Even though Ruby is more memory intensive and slower than Java, the productivity level from using a dynamically typed language like Ruby increases as a programmer can write code quickly and have it tested in minutes.

The difference between Python and these other languages was how easy it was to write the Python code without knowing the language to begin with. The team members that wrote in Python were able to grasp the language fairly quickly. Debugging in Python was simple as well, especially when using a tool like Eclipse. The exceptions thrown were easy to find and very simple to fix in comparison to Java's errors. Also, unlike Java, Python isn't overly verbose, and it is dynamically typed. Combine these features with the vast arsenal of libraries that Python offers, and you have a language that can implement almost anything you want, without writing a novel of code. Implementing the socket was straightforward, but we had to use a slightly different approach when parsing the GameState string. While Haskell used pattern matching, and Java grabbed one line at a time and looked at each individual line, Python grabbed the entire string, and used delimiters to parse through the string. This approach wasn't too difficult, but it was a bit unfamiliar. It would be worth mentioning that when we tried to use Java's way of reading in the GameState we encountered the problems that are mentioned in the above section. Lastly, Python has the option to be object oriented if a programmer needs it to be, but fortunately we were able to implement our client without it. Overall, Python is easy to get comfortable coding with.

Haskell was surprisingly easy to implement. Although the common strategies were very different than what we were used to, the homework from class provided enough experience to make a program that worked. Because our team was relatively new to the language, we decided to start with a parser from scratch. We hadn't yet discovered the awesome functionalities of parsec and this was also prior to learning about monads and Haskell IO. Had we started this program after learning everything about the language, things may have been different. As one can see from reading the file, there is an absurd amount of code, a lot of which is completely unused but was left there for its potential. Although the program most certainly could be rewritten to be more efficient and elegant, the struggles we faced allowed for a great learning opportunity on the advantages of functional programming. One great aspect of Haskell was the GHCI interpreter/compiler we ran at the UNIX prompt which allowed us to test each function separately with appropriate parameters. This made testing a simple step by step process involving no extra hassle of having a main class that also needed compilation before running. For the most part, we liked the compiler. It was like an over cautious mother making absolute sure we were ready for school before leaving to catch the bus.

We were able to use the results from games played as a benchmark for comparing the performance and difficulty of implementing artificial intelligence in imperative vs functional languages. After obtaining these results, we compiled a basic analysis of these performances. Interesting things to note: Haskell was always the quickest language to send responses based on what move it wanted to make. Python was the slowest, and Java and Ruby were usually somewhere in the middle. This proves that functional languages execute their code faster than imperative languages, generally. Haskell may be a "lazy" language, but maybe the fact that everything is already defined in variables and functions makes a significant difference in speed. While imperative languages execute every line of code they encounter, Haskell reserves execution only for lines of code that are absolutely needed, possibly making it faster. The reason Python was slower than Java and Ruby is probably the type system. While languages like Java are statically typed, which means the compiler knows the types of variables up front, Python is dynamically typed, which means variables aren't assigned a type until execution. Not only that, but Python is also compiled at runtime, which perhaps slowed down execution a little more. As far as statistics went, Java and Ruby usually had the most wins, Haskell had the least, and Python was usually somewhere in the middle. Does this mean imperative languages implement AI better than functional languages? Perhaps, but Java and Ruby had the most basic strategy out of all our clients. This result tells us a couple things: either simplicity is best when it comes to Blackjack strategy, or functional languages unnecessarily convolute their strategy by pattern matching a little too much and drawing unrealistic conclusions. A simple analogy would be a person who overthinks every problem and comes up with irrational solutions. Realistically, it is probably a combination of both aspects that caused Java/Ruby to win the most and Haskell to win the least. One example of our benchmark findings is shown below, as well as a picture of the GameState string we use to keep track of the table.

GameState that is sent to each A.I. is below and for this trial, we played ten games with four languages plus the dealer. With each language file playing three A.I. Along with the game state, the amount of wins and losses for each language is below along with the average response time from each A.I. In this test run java was the fastest, but for the majority of the time Haskell has the fastest response time and the data is sometimes inconsistent.



```
Java
W: 9
L: 17
B: 8
D: 4
Average Move Time: 4.286262 ms

Haskell
W: 6
L: 23
B: 11
D: 1
Average Move Time: 8.028017 ms

Python
W: 11
L: 15
B: 6
D: 4
Average Move Time: 15.126867 ms

Ruby
W: 8
L: 20
B: 5
D: 2
Average Move Time: 24.130542 ms
```

Parting Words, Future Goals

We consider this project a success. Not only did we get immense coding practice of software principles such as socket communication, object oriented design, artificial intelligence implementation, data analysis, and benchmarking, but we also were able to experience all this with different types of programming languages. We were able to truly compare the performance of imperative and functional programming languages.

In the future, we hope to refine the strategy that the player clients use to make their moves. This will of course involve many more conditional statements, more pattern matching, and more computations. One idea we have is to have each player look at the cards on the board, and decide what the probability is that the next drawn card will be equal to or lower than the amount needed to reach 21. This new strategy will allow us to benchmark the performance of these languages more comprehensively. We also hope to implement a “split” option for players making their moves, as that is considered a sizeable aspect of Blackjack strategy. An idea we had for that is to simply insert an “S” in the middle of a player's hand in GameState, to signify that his hand has been split into two. Lastly, we also want to allow more languages and human players to play the game. Once adding the human player functionality we also hope to add a GUI that human players can use that is more user friendly and more interesting than looking at a black terminal. Adding a human player should be easy; we can simply develop a program that shows GameState to the human player by printing it, and then asks him to input his move to stdin. The rest of the backend work will be identical to the way the clients and server communicate now.

We hope you enjoyed learning about our Blackjack Benchmark, as we certainly enjoyed creating it.