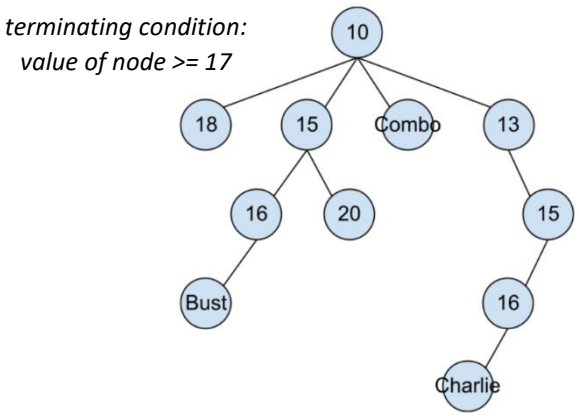
FIT2102 Assignment 2 Report

Tan Chong Ern, 31435661

General Code Design

The code for this Blackjack player revolves around probability trees, while incorporating some heuristic procedures as well as basic strategy to ensure that the `Action` it takes would give it a better chance of beating the dealer. The `Tree` was defined to have a flexible structure, with optimizations for efficient utilization of machine memory to prevent timeout errors. It enables configuration to the terminating condition along with the tree height, which represents the number of cards that can be drawn before the tree stops creating more branches. I decided to include the specification of the terminating condition (or value) because by doing so, we can simulate all course of actions that the dealer may take before hitting a minimum value of 17 and then standing. We give the root a probability of 1, and thus, since all the leaves stem from that probability through multiplication, we find that the sum of all probabilities in the leaves of the tree add up to 1. With this, the probabilities of the dealer landing at each value can be easily calculated by aggregating the values from the leaves of the tree. Probability trees are used in gauging the strength of the player's hand as well (against the dealer's) to decide the next course of `Action`, and this can be found in all `Action`-deciding functions. Here's a very simplified version of a probabilistic tree for the dealer for better visualisation and understanding:



Within the memory, I have decided to create another new data type to store the remaining number of cards for each `Rank` left in the deck, called `CardFreq`. I chose this over just having a list where each index represents a `Rank` because indexing lists can easily lead to runtime errors. I also decided against hardcoding the `Ranks` as attributes of a single datatype, as the code becomes more messy and less elegant with the use of excessive if-else statements, especially when calculating probabilities and deciding on `Actions`. On the other hand, defining a new data type has allowed me to leverage the other existing data types and their functions, namely `Rank` and `List`. This is evident in updating the state of the deck and in creating new hands for branching out probability trees. Furthermore, by keeping track of the deck state as a list of `CardFreqs`, I am also able to use the existing `fold`, `map`, and `filter` operations in my functions, making them less rigid, more readable, and easier to understand.

Another design choice I made concerns the bidding strategy of the player. Because bidding happens at the start of a round when cards are yet to be dealt, it is not so trivial to come up with a solution to determining how much to bid. At first, I chose to always bid the maximum bid; while this did help the player to win a lot at times, the player also lost drastically just as much. Then, I decided to utilize the probability tree such that the player will bid the maximum bid when the chances of getting a good hand are high, and vice versa. While this did lessen the variance of my player's score, it was not enough to secure a good position on the ladder.

Therefore, I chose to leverage the players' points into my bidding strategy so the player can gauge its standing among all other players and decide the amount that it should bid to help it get within the top 3 ranks. When the chances of a good hand are fairly high, the player bids the maximum bid so that in the likely occasion that

it wins, it has a chance of overtaking the player above it and improve its ranking. However, if the chances of a good hand are moderate, the player bids the difference in points with the player below so that in the occasion that the player loses, it will not lose so much that its ranking will drop further. This is possible since we believe we have a good algorithm that decreases the likelihood of busting; such that if the player loses, it is likely that the dealer has won, and all other players have either lost or drew. When the chances of a good hand are low, the player would then bid the minimum bid to avoid extreme losses. This strategy helps the player to maximise profits and minimise losses.

Lastly, I chose to use some parts of basic strategy in deciding when to `Split` and `DoubleDown` because basic strategy is the known optimal way of playing any hand to minimize the losses of a player in Blackjack. This is incorporated with probability trees, so that the player can leverage the memory and information provided aside from this naïve procedure, giving it a more favourable chance of winning the dealer. The general algorithm for the determining the amount to bid, the `Action` to take, as well as other responsibilities of the program can be found as block comments in the code under the section headings (e.g., `Bidding` and `Actions`).

Parsing and Memory Management

For the purposes of the player's strategy, our `Memory` is composed of:

- **The current bid amount** – for knowing the amount to bid for `Insurance`, `Split` or `DoubleDown Actions`
- **The state of the deck** – for calculating the probabilities of all possible hands in the probability tree
- **The player's previous actions** – to know which turn the player is in to filter for the latest information
- **The dealer's latest up card** – to avoid counting it twice when information is given out in a new round
- **The number of unplayed dealer cards** – to offset the probabilities, making them unbiased (from rounds where all players go `Bust` and the dealer does not play)

All these attributes are vital to the player's decision-making process so there are no redundant or unused attributes.

Context-Free Grammar for the Memory shown in Backus-Naur Form

```
<memory> ::= <int> "," <deckState> "," <lastActions> "," <lastUpcard> "," <int>
<int> ::= <digit> | <digit><int>
<digit> ::= "0" | "1" | <twoToNine>
<twoToNine> ::= "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<empty> ::= ""

<deckState> ::= "[" <empty> "]" | "[" <cardFreqs> "]"
<cardFreqs> ::= <cardFreq> | <cardFreq> "," <cardFreqs>
<cardFreq> ::= <rank> ":" <int>

<lastActions> ::= "[" <empty> "]" | "[" <actions> "]"
<actions> ::= <action> | <action> "," <actions>
<action> ::= "Hit" | "Stand" | "DoubleDown" <int> | "Split" <int> | "Bid" <int> | "Insurance" <int>

<lastUpcard> ::= "Just" <card> | "Nothing"
<card> ::= <suit> <rank>
<suit> ::= "D" | "C" | "H" | "S"
<rank> ::= "A" | <twoToNine> | "T" | "J" | "Q" | "K"
```

Parser combinators are used to deserialise the memory into its components, making it very simple to manage the player's memory throughout the game. Certain parsers (in the `Parser Utility` section) were inspired from Parser.hs in Tutorial 11, from which most of the other parser combinators build up from. I chose to create more generic parser combinators such as `parseShow` and `parseList` to improve code readability and reduce the use of string literals, which are a common source of parsing bugs. `parseShow` parses any `Showable`

according to an input list of its possible type values, while `parseList` parses out enclosing square brackets and returns the parsed contents in a list where each item is separated by commas (that are ignored when parsing). Hence, `parseList` is one key way that I integrated my BNF grammar into my parser, especially for the `Actions` list and the list of `CardFreqs`, while `parseShow` is mainly used to parse `Suits` and `Ranks`, since they have predefined `Show` instances.

Even though I wanted the memory to imitate JSON notation at first, I decided against it in favour of less complexity, and because the use of names for objects do not contribute to the player's decision making so I considered it as redundant information.

Functional Programming and Haskell Language Features

Functional programming with a pure language like Haskell has made the programming of complex functions simpler, without the need to keep track of mutable variables that are the source of many runtime bugs. The strict typechecking has also ensured that the program created is robust, allowing typeholes for solving many problems. Furthermore, due to the nature of the program being immutable and all the functions being pure, we can acquire results from our functions deterministically making it easy to reproduce errors and fix them.

Haskell's Language features like composition, partial composition, and eta reductions were applied to higher-order functions where appropriate, especially in the use of array functions such as `filter`, `map`, and `fold`. The Haskell typeclasses of `Functor`, `Applicative`, `Monad`, `Foldable`, and `Traversable` have all been used throughout this program. The parsers, memory management, and probability trees are where these typeclasses tend to be the most effective in, because of the recurring need to apply functions onto data that is wrapped within a context. On the other hand, the more logical procedures of the program are more suited towards using pattern matching due to the nature of making decisions based on a set of conditions.

Final Thoughts and Conclusion

Throughout the development of this Probabilistic AI program, the hardest part that I encountered with the program was not the `Memory` or the `Parsers`, but actually fine-tuning the thresholds for deciding `Actions` based on the probability trees. This is because the game of Blackjack is very much based on random luck, so there is no solid way to guarantee a win against the dealer. Initially, I defined an instance of `Monoid` and `Foldable` for the probabilistic tree as well. However, I decided to remove it because it led to quite an amount of extra code that have very little usage, and since normal recursion gives a very simplistic and intuitive way for defining the same functionality. Overall, I found it satisfying to be able to apply what I've learnt, especially on `Functors` and `Foldables`, to create a program with concise but effective code.

In conclusion, this project has opened up new perspectives for me on coding with the use of pure, deterministic and curried higher-order functions, and it has taught me to appreciate the simplistic nature of functional programming and the merits of immutability.