**Comp Science NEA**

Contents

[Analysis 2](#_Toc509133610)

[Introduction to Organisation and Client 2](#_Toc509133611)

[Outline of the Problem 2](#_Toc509133612)

[Investigation 2](#_Toc509133613)

[Analysis of Investigation 4](#_Toc509133614)

[Objectives 7](#_Toc509133615)

[Data Flow Diagram of Game 11](#_Toc509133616)

[Data Flow Diagram of Current System 12](#_Toc509133617)

[Research 13](#_Toc509133618)

[Potential Solutions 15](#_Toc509133619)

[Proposed Solution 19](#_Toc509133620)

[Proposed Solution DFD 20](#_Toc509133621)

[DFD for Game System 22](#_Toc509133622)

[ERD for Proposed Solution 23](#_Toc509133623)

[Proposed System Neural Network Architecture 24](#_Toc509133624)

[Prototyping and Critical Path 26](#_Toc509133625)

[Documented Design 27](#_Toc509133626)

[BlackJack Prototype 27](#_Toc509133627)

[Introduction 27](#_Toc509133628)

[Design – The Blackjack Game and Interface 28](#_Toc509133629)

[Design – Card Counting AI 38](#_Toc509133630)

[Design – Neural Network Based AI 52](#_Toc509133631)

[Design – Simple AI 58](#_Toc509133632)

[Design – Comparison Tool 58](#_Toc509133633)

[Software Development 58](#_Toc509133635)

[Testing 58](#_Toc509133636)

[Evaluation 58](#_Toc509133637)

# Analysis

## Introduction to Organisation and Client

Mr X is a freelance graphics designer who wants to begin developing games. His background in graphic design mean he specialises in developing smooth and aesthetically pleasing interfaces. However, whilst he has been learning how to program and develop games on a fundamental level, his technical ability is still growing, and his mathematical ability is not as proficient as he would like it to be for some parts of this development.

Knowing me from secondary school, from being in the same IT class, he remembers our contrasting interests – his in the more design focused aspects, mine in the more technical side – and has contacted me to see if we can collaborate on this project of his.

## Outline of the Problem

His first game is going to be a casual (online?) Standard deck card games game. The most technical aspect of this project is the development of a competitive AI, which the user can effectively practice against, with varying levels of difficulty. The developer is looking to delegate this objective to someone who has a deep understanding to the mathematical aspects and concepts underlying. Specifically, the developer wants the AI to have different general personalities (in a game context: aggressive, defensive), as well at higher difficulties to have varying personalities based on the situation.

For example, one of the standard deck card games which will be a part of the game will be poker – in this context, the client would like the AI to have a personality where it is aggressive - raises a lot to attempt to intimidate the opponent for example - and another where it plays more passively – folds more often, and raises less often. In addition, when at the higher levels, the AI would play more aggressively when it has a higher bank, or it thinks the opponent is on tilt, and in other scenarios it would play more passively.

The current system, if it were to be developed by the client, would consist of simple logic, and transparent decisions, where the AI would eventually become predictable and the interest from the end users in the game would drop very fast. Under the current system, the client would have to spend a lot of time explicitly programming the AI, in order to accommodate for specific scenarios, to emulate the nuanced decision making of a human. As a result, the client would like the AI to be competitive and a challenge for the end user, as well as this the game development deadline is 6 months, consequently, the client requires the development time to be kept reasonable to this deadline.

## Investigation

**Interview with Primary Client – Mr X**

**Specifically, what is the game you are thinking of developing and which demographic are you aiming it towards?**

It’s good to talk to you again after all this time! I hope you are doing well.

Well, I have an interest in games like poker, I just find them so fun! However, a side me of dislikes the potential gambling aspect in the sense that someone could destroy themselves over a game, and sometimes people just have not had enough practice to start playing for real – my goal is to develop a game for these sort of people, more of a casual base. I just want people to have fun, and I hope to keep them safe whilst they do it, there’s a bit of a vocational aspect for me personally.

**What are the central focuses of the game you are trying to develop?**

You know graphical design is where my strengths lie, so, I want to practice where the skills overlap, so I am thinking more of a focus on the UI. But, I also realise how technical this field can get, so I want to start learning the basics of the logical process, and some parts of web development. The catch is, with a game like this I do not want some of the more casual users to be put off by pressure of having to play against other real players, if they do not want it, so, I want to develop an AI which they can play against, but I am currently struggling with this more than anything else in the project.

**From your statement, it sounds like you have attempted to implement this AI, which suggests that you have design elements in mind. What, at the moment, are the central design elements or features you would like for this AI?**

I think I have a pretty good vision of the AI. I want this AI to have dynamic difficulty levels – I want it to be competitive, but not exceedingly difficult, to the casual user especially, but I also want it to provide a challenge for the more experienced users. After all, in my humble opinion, there is no fun if there is no challenge.

Also, an exciting part for me is designing this game with a compelling caricature aesthetic, with characters who have striking personalities. As a result, I would like the AI system of this game to have differing personalities to reflect these characters. For example, I am designing a half-man, half-bull character to be in the game, the AI for this character should have a more aggressive personality than usual, to reflect these characteristics.

**What is the current system you have, or are in the process of implementing?**

Ahh, it’s not very good, but so far, I have a simple pattern matching system, where the AI checks the state of the game, and then it checks against conditions I have programmed in, it then follows the corresponding action. Just a long if-elif-else chain, really.

**What are the benefits of the current system?**

Well, although it is not as good as I would like it to be, it does work on some level, I suppose. It can play a simple game, and sometimes it is kind of fun to win, however, it can get really boring, really fast. The only real benefit from a developing standpoint is that I can easily add more conditions for it to check against.

**What are the drawbacks of the current system?**

The big problem, really, is that it is very easy to pick up on the patterns the AI follows, I’m not sure if I am a bit biased, because I know the conditions it looks for, as I am the one who made them, regardless, it is too predictable at the moment. I’ll give you a quick example, say you raise a lot of money, the AI will always fold, as it will think that you have a good hand because you raised so much. It is so easy to exploit! Just raise a huge amount every round, and you will always win! Also, the AI will only ever be as good at poker as I am, because I cannot program to check scenarios I am not aware of.

Lastly, although a secondary issue, the AI currently does not exhibit the personality types which I desire. Moreover, to program this in would require a lot of time, because, I would have to program a different reaction to each scenario – so, say I want to have 3 personality types, for each pattern the program recognises I would have to program a different decision or reaction for each of these types. Furthermore, the AI would require a lot of maintenance, due to the nature of having to account for a lot of different scenarios – even if one common scenario is not accounted for, the AI is exploitable and would require constant patching, lowering the overall quality of my product. It’s a lot of hassle at the moment.

**Are there any final points you would like to add?**

The development of this AI is stunting the progress of the development of the final product a lot more than I would like. Preferably I would love to collaborate with someone, so that I can focus on developing the skills I want to focus on. In addition, ideally I would like the AI to not have the current maintenance aspect which it currently has, the best case scenario would be for it to be self-sufficient – by learning from its mistakes, for example – so that it is of an acceptable quality, even on the launch date.

## Analysis of Investigation

Overall, the system needs to be a dynamic implementation of a dynamic artificial opponent, appropriate for players of a wide skill range. It needs to be dynamic in the sense that it can adapt to the player’s ability, and playstyle, as well as having a low maintenance requirement – so that it is possible for a small team to maintain. In addition, to fit with the quirkiness of the game’s narrative, the system needs to have different playing “personalities” to fit with the personalities of the in-game NPC personas.

From this interview it appears that the client would primarily like the AI component of the development to be delegated. Firstly, a quality of the AI which the client currently likes is the aspect of its extendibility – “I can easily add more conditions for it to check against.” whatever the new system will entail should keep this aspect of extendibility.

On the flipside, the main issues with the current system appears to be its effectiveness, specifically, the AI is “it is too predictable at the moment” and “It is so easy to exploit!”, from this, we can reason that the new system needs to be more nuanced so that it cannot be predicted quite as easily. The example which is given is that the “the AI will always fold, as it will think that you have a good hand because you raised so much”, so one specific improvement could be to add a calculation element where the AI puts a probability of the opponent bluffing, then proceeds to fold if they think that the opponent really does have a big hand – either from previous experience of the opposing players tendencies or from absolute probabilities. Specifically, this issue of nuance is exhibited in the fact that the current system will always make the same decision in the same explicit scenario.

The current system is programmed using a “a simple pattern matching system” where the developer has preprogramed the AI to recognise certain states of the board – supposedly the data is taken straight from the game, since the product is of his own development – and then execute the preprogramed action in reaction to this pattern.

The current system works like this:

The big aspect here is the checking of the different edge cases/patterns – all the nuance in the program relies on having a robust case checking system full of a lot of different patterns. As the user states, this system is difficult, because it requires explicit maintenance of each different scenario the game could be in, and if one case is not accounted for and it occurs often enough, then the entire game is exploitable, which ruins the quality of the overall product. From this, we can infer that a better system would to develop an AI which does not need explicit programming to recognise different features, so that it can implicitly adapt to each new scenario, and develop its own nuance.

State\_of\_hand <- get\_hand()

…

Get state of all aspects of game

…

IF Opponent Raises High THEN

FOLD

ENDIF

…

Check more edge cases/patterns

…

Bet <- calculate\_bet(current\_hand\_score)

On the flipside it is important to make sure that the AI has differing difficulty levels, so that the difficulty of the game matches the ability of the current user, so that the user and the bot can remain in a competitive game, so that the users of any skill can enjoy the final product. A subtle aspect of this will be that it is more important for the final AI to be able to play less skilled users than users of high skill, because the client states that he wants to orient the product to “more of a casual base” – in addition, users of higher skill would most likely be playing on professional online gambling websites.

Another big aspect of the AI which the user would like implemented, is the feature of “personality”. The client states that his game has a style of “compelling caricature aesthetic”. The example he gives is of a half-man, half-bull character. The consequence of this for the AI is that the AI should reflect these personalities in game – for example, the half-man, half-bull character would be more aggressive than some of the other characters.

**Main Points:**

* The current system is too predictable, there is no challenge for the user once they figure out the very linear behaviour patterns
* Whilst it is easy to extend the current system by providing more nuanced scenarios to check against, this would increase the maintenance requirement of the system by too much.
* The client would like an aspect of playing personalities in the final system, which would synergise with the caricature NPCs being developed for the more story side of the game.

## Objectives

1. Firstly, the new system should provide a fully autonomous AI which can play against one or multiple opponents in a game of Poker, with a response time of less than 30 seconds, and a competitive difficulty, for a casual player. To elucidate the somewhat abstract definitions of the behaviour of the AI, autonomous means:
2. The AI should be able to adapt to new situations, without having to be explicitly programmed to handle that situation.

Furthermore, competitive means:

1. The AI should remain nuanced enough to not be dominated by opponents adaptations
   1. The AI should not exhibit an overly predictable playstyle, to the point where it can be countered easily. For example, if a casual player can predict its next action more than 50% of the time, during edge cases in a game, then it is too predictable.
   2. This objective does not apply during scenarios in a game for which there is a strictly dominant strategy – for example, if the game were blackjack, then hitting when on a hand value of less than 11 or standing when on 21 are both strictly dominant strategies. In these cases, the AI should prioritise the strictly dominant strategy, rather than the adaptability.
2. If the AI is losing multiple times to a specific opponent, it should identify the style/personality of the opponent and adapt to this player, in order to provide a competitive game to this specific player. For example, this is more of an extension on point a:
   1. If the player keeps using the same strategy to beat the AI, the AI should not keep making the same failing move, unless the failure is down to chance, but transition to more effective move.
   2. In the scenario that the agent is being dominated by a particular playstyle, then the AI should identify this playstyle, by comparing it to previous players or experiences.
   3. The agent should then, if it has the required experience, change its own parameters which have proven more effective against the identified playstyle.
3. If the AI is dominating its opponent too hard, it should automatically identify this and drop its difficulty. This means that the AI should remain competitive to the opponent, but should not dominate them, to the point where it is not fun for the end user to play the game; equally, if the user begins to dominate the AI, the AI should increase its difficulty to adapt to this player. Concretely, if the AI wins 10 or more games in a row, or less if the games were not competitive (ie the user lost is every round they played), then the AI should change its behaviour to play less effectively. Moreover, the user should also have the option to disable this functionality, as they may purposely want to play against its highest difficulty, because they find it fun.
   1. Difficulty of the AI will be defined by the handicap that the AI will apply to itself – at the highest difficulty, the AI should have no handicap applied and will always play the move it thinks is optimal. At lower difficulties it should have a probability to play a move which it thinks is suboptimal, or play a less effective variation of the move it thinks is the most optimal, if applicable.
   2. When the AI has identified a skill disparity, it should increase or lower the handicap applied to suit the current player.
4. The system should adhere to any fundamental game principles as much as it can.
   1. For example, in Blackjack the agent should never hit when it has reached a hand value of 21, and it should always hit when it has a hand value of 10 or less.
   2. Another example of a game principle it should stick to is the fundamental theorem of poker[[1]](#footnote-1). However, since poker is a game of incomplete information it is impossible to adhere to this 100% of the time, so a successful implementation should adhere to this 60% of the time, at the medium difficulty level, and higher at higher difficulties as well as lower on lower difficulties.
5. The module for the AI should provide a very restrictive interface, in order to prevent the end user from destructively adding games to the training, or changing the behaviour of the AI in any sort of way.
6. The program’s interface should only be able allow the client’s program to send the state of the current game, and the only thing it should return is the AI’s next move.
7. In addition to this, it should provide a separate interface after each game, which allows the client’s program to send the record of the game, in order for the AI to be able to analyse it and add it to the AI’s training (discussed in a later objective). This interface should only be accessible after each game, and would need a security verification, to prevent any other scenario from adding a game to the training of the AI – this ensures “False games” (games which did not happen) do not influence the AI’s behaviour, so it only adapts to empirical experience.
8. An addition an interface should be provided which allows the user to configure the personality of the AI, externally from each game. The extent of this interface should include a preset list of options the user can select from. Moreover, this aspect of the AI should not be adjustable within the game, so this interface should be disabled whilst the AI is in a game.
9. Externally to playing the game, the AI needs to be configured to play with differing playing personalities:
10. The pre-set personalities which the user has requested are: basic aggressive, and basic passive. (For more detail read the research section)
11. Aggressive personalities should exhibit behaviour which shows that the AI is more likely to bet more, raise more often, and fold less often, hit more, etc. compared to the base level version of the AI.
12. Passive personalities, in general, should be more inclined to perform slow plays (aka Sandbagging or trapping[[2]](#footnote-2)), this is where the AI would be more inclined to bet weakly, or stand despite having a strong hand, in order to deceive the opponent. This slow playing style should be more likely to be exhibited compared to the baseline AI.
13. The mechanics of the personalities should vary based on the difficulty level.
14. At the lower difficulties, these personalities should reflect emotions – for example, after losing a big hand, the program should emulate a level of “tilt” –meaning the system would not perform as much of an optimal play, after losing a few rounds - contrasting this to if the AI had won a big hand, after which it would “be more confident” and be more likely execute the optimal play.
15. In addition to this, at higher difficulties, the AI should not have a preset personality, but a dynamic style which changes based on the state of the game. Also, the program would be less inclined to exhibit an “emotional” play style – for example, it would be less inclined to go on “tilt” after losing a big hand.
16. There should be an interface to configure the personality of the AI outside of the game, however, this should be restrictive to only external to any game.
17. There is potential for the AI to be trained, if this is applicable, then the program needs to fulfil the following requirements in this regard:
18. The program should be provided to the user pre-trained, and ready to be implemented into the game that they are creating.
19. The AI should use each new game it plays to add to its training, meaning that each game it plays will add to the proficiency of the bot. However, it should only train itself after a batch of 50 games, for example, as to not be influenced too much by anomalous games.
20. An interface should be provided, to allow for the client to add their own store of training games, if they wish to train the AI further.
21. The training data, as well as its algorithm should not be stored with the AI, in order to prevent potential malicious users from attempting to change its behaviour.
22. If the AI plays enough games against a significant user, which dominates the AI, the AI should attempt to identify the playstyle of this player, and retrain itself to play against this style in particular, then apply this new training either to this player, or players with a similar style, if the training it already has is not adequate.
23. To fulfil the other requirements of adapting to the user, the program needs to have a method/interface for identifying each user. This can either be built into the program, or be provided by the client. This involves, at some point, the user being able to log in and be provided with a unique account, in order to identify them. This includes:
24. A sign up system, where the user can provide a username and password, to create an account.
25. A log in system, so that previously existing users can sign into the program, with a record of their previous games.
26. The username should be unique to each user, and this should be checked and sanitized during the sign up process.
27. The password will have minimum requirements of at least one capital letter and a number, to ensure the user does not allow themselves to have their account compromised.
28. The username should be sanitised, to prevent attacks on the system. However, the password should be hashed and the hash value should be stored, rather than the password itself, as a result this does not need to be sanitized.
29. Relating to the last objective, keeping a record of past games is a good way for the end user to reflect on previous decisions and improve, however, there is potential for the AI to use this history of games to improve its own performance. As a result of this double need, there should be a method of storing data about each game played.
30. A record of each game should be stored in a database – with each record being an account of each game played.
    1. This record should store a unique game id, a series of winner ids, a series of winning hands with corresponding winning values, the number of turns each game was, and a list of players who played in each game.
31. There should be a relationship between three tables – a table of users, a table of moves; which will contain information about each move in the game, and lastly a table of games, connecting multiple users to a single game.
32. This database should only be writable by a central server which is hosting the game, every user and the AI should only be able to read from it. Additionally, the users should automatically have access to each game they are involved in, and then any other games which they download externally. This access should be locked whilst a game is in play.
33. As the client potentially would like the final product to be web-based, multiple instances and connections to this database and AI is possible, consequently, some safety features should be implemented to prevent possible conflicts. An example, of this may be the use of timestamp ordering or serialisation.
34. The end user would like to gain a deeper understanding of the dynamics of different playstyles, and agent types in a game setting. As a result, he would like a tool which can be used to get different data about different games, and then parse this to output easily understandable information about how different players or agent parameters lead to different winrates or behaviours.
    1. Firstly, this comparison tool should be able to parse data provided to it from a database. In the design section, the design of this database, will show the format of data this tool should be able to parse.
    2. The tool needs to be able to output information about players or agents both in isolation, and in relation to other players
       1. In isolation, the tool should provide data about the general winrate of the agent, as well as common statistics: for example, for blackjack information should be calculated about what value an agent or player likes to stand, on average.
       2. In relation to other users, the tool should output data about winrates, in both a specific and general way.
          1. Specifically, if there are standout users that the user being analysed has a high play-rate against, or a high win-rate/loss-rate against.
          2. In general, if there is a particular classification of playstyle which the user is particularly good against, or particularly bad against, then this information should be calculated.
          3. On the other hand, if the player is particularly balanced in terms of their win rates against different playstyles, then this should also be displayed.
          4. The change in data over time should be calculated, so that any trends in the user behaviour can be output.
       3. In relation to other users, a hidden MMR (Matchmaking rank) value should be calculated so that users can be matched up with people of a similar skill level. The different difficulty of agents should also get their own MMR value, so that it can be easily predictable what skill of user they will be matched up against
          1. The user should have a required number of placement matches before they are given a viewable MMR
          2. This MMR value should be updated after every game, and should increase with a win, and decrease with a loss.
          3. The relative change of a user’s MMR should be scaled with the difference in MMR between players – for example, if a low ranked player defeats a high ranked player, they should gain more MMR than they would playing against people of a similar skill level.
          4. Users should be matched with other players of a similar skill level.
    3. This all the information calculated by this comparison tool should be output in a visual way so that the user can easily understand the information being shown.
       1. For isolated information about a user, a line graph, or similar chart, should be output, so that the change in the user’s behaviour over time can be plotted.
       2. In relation to other players, a graph could be output, which would show the relationships of playrate and winrate between players, where thicker lines would show a stronger connection.
    4. This tool should be intuitive for the user to user, as a result a UI should be produced which would allow the client to navigate the tool and choose different parameters. This tool should allow the client to enter a user’s id, and choose which different aspect of the user the client would like to view – such as winrate, or best/worst playstyles to play against.

## Data Flow Diagram of Game

Regardless of the nature of the final system, the nature of the game with stay consistent. It is important to lay out the data flow for the game.



## Data Flow Diagram of Current System

**Level 0 DFD:**



As far as the current system goes, this as detailed as it will be. Within the process of “Calculate System Position Value and Generate Move”, a series of if statements are utilised to generate the calculation and move.

## Research

Poker is a game of incomplete information – the complete state of the game is unknown at any given time – unlike a game such as Chess or Checkers. Consequently, this makes designing an AI with concrete, non-flexible calculations and pure mathematical models a less optimal solution. This holds especially since the opponent for the AI will be a human player who will be able to spot these patterns and adapt their own playstyle, and exploit the AI, if it remains linear – this problem is exaggerated especially in the current system.

A decent method would be to find a database of poker hands, such as the UCI Poker Hand Data Set[[3]](#footnote-3)(dataset of hands for 5 card draw, over 1 million instances) or Michael Maurer's IRC Poker Database (University of Alberta), then use statistical analysis to determine the common patterns or behaviours, in general. The biggest issue with this is that the data I have found so far is not contextualised, and the dataset does not contain the moves made by each player. For example, the data could have come from a high rolling club in Las Vegas, or a million instances of different kitchen table games, as a result, it is hard to add any context to the analysis of the data. In addition to this, the most the data could be used for is to analyse which hands are most likely to win – it does not provide any information for general player behaviours; this issue which stems from this is that using solely data analysis to calculate the behaviour of the system is that it may become too linear and play in one playstyle – for example, only betting when playing with a big hand. In addition, these datasets will generate a weakness in the system in the sense that it will not be able to reason about other player's behaviours, and adapt to them, whereas, some human players will be able to reason about the system's behaviour of using purely statistical information to inform its play, and perhaps outplay the system too easily.

Regardless, the fact that there are datasets already available makes it easier to provide some sort of baseline to the system I will develop, and still may be useful. Some concepts to apply to this information may be Bayes Theorem, and Nash equilibrium; these are game theory ideas which could be applied to my system to make it more effective at playing the game. Beyond that, within the game, decision trees and minimax trees to use within the game, despite incomplete information, these can be used to model possible future scenarios and decide which move may lead to the highest value position.

Another aspect which would be explored is the possibility of using random states to explore the trends of Poker, as an incomplete information set from any one game makes it hard to provide concrete and reliable analysis. One such method may involve using the Monte Carlo Simulation[[4]](#footnote-4), and generating my own dataset via an AI simulating many games against itself. Whilst this does, to a degree, amend some aspects of the system not being able to analyse aspects of opponents behaviour as the AI could be preprogramed to play in a certain way (such as tight passive, etc. Discussed later), and then the actions taken by each playstyle can be analysed, and extrapolated to assume that a human opponent would behave in a similar way, if they were to follow a similar playstyle; the problem is that a human opponent may not play exactly the same way, or any of the ways that the pre-programmed AI may play, in addition to this, this adds an extra dimension of initial opponent behaviour analysis, in order to determine which style category they may fit into.

In addition to this, another possibility could be explored fitting with the idea of exploring random initial conditions. Neural networks have been used in conjunction with an evolutionary algorithm, or reinforcement algorithm and made to play against itself to develop a unique playstyle and metagame[[5]](#footnote-5), this microcosmic metagame which the AI derives from itself may throw human opponents off, making it more effective[[6]](#footnote-6). Utilising neural networks may be a useful way to find an optimal behaviour to this problem, with the case of incomplete information, without the need for unreliable calculations based on speculation on the part of the system (as the system in general would not be effective at analysing behaviours and extrapolating this to value of future hands). This could be implemented in different ways – for example, a baseline weight for the system could be developed, and then used within the game, but then developed further against new opponents, in order to generate more effective weightings for their particular playstyle.

There are a few different options to implement a system like this. One possibility is to use a matrix library, such as NumPy[[7]](#footnote-7), and then design and implement my own neural network. This is a viable solution, as long as the matrix library in question is efficient in its operations. This would allow me more freedom in my implementation of the network and applied algorithms. However, this would vastly increase development time, and is more likely to have bugs and performance issues, compared to the higher level libraries, as my implementation would not be as optimal as other libraries due to my inexperience. Another option would be to use a higher level library, such as TensorFlow[[8]](#footnote-8), to build a neural network from a higher level, and then build the more specific aspects of my solution around this – for example I could use Tensorflow to implement a neural network and the learning algorithms (such as gradient descent) and then use my own algorithms and manipulations of the data for the personality aspects.

According to Pokerology.com[[9]](#footnote-9), in terms of "personalities" or playstyles, there are generally two spectrum which a player may fit on: Tight vs Loose, and Passive vs Aggressive. Tight players are defined as players who, in general, only play when they have a strong starting hand, and loose players, in contrast, will play more with weaker hands. In addition to this, Passive players tend to fold more often, and call/check rather than raise when they have a strong hand; following this, aggressive players tend to raise more often and risk more chips.

These can then be combined into more specific general playstyles: for example, a Tight aggressive player (colloquially labelled as a "Shark") tend to not play for many pots, but when they do they try to maximise their opportunities when they have a stronger hand. Moreover, as humans tend not to be linear entities, their playstyle may change in accordance to their emotion (for example, a player who plays worse after losing many pots in a row may be said to be on "Tilt"), or they may purposely change their playstyle to fit the table they are playing at – consequently, there is an extra dimension of Tricky vs Straightforward, where a straightforward player is more likely to fit better into the theoretical description of their general playstyle, whereas a Tricky player may change it up a lot more. In general, according to this source, aggressive playstyles tend to generate more revenue than passive playstyles, where Tight Passive players is tagged as the worst playstyle, putting their playstyle down to "being scared". Whether this may or may not be the case, all these playstyles are relevant to this problem, because the final product may be suitable for a user who is new to the game and may be inclined to play a more "Tight Passive" style, as they are still learning the game, as a result, this needs to be taken into consideration. Moreover, these concrete examples of different playstyles could be mapped to different personalities, which is one of the objectives of the client.

**Main Points:**

* Poker is a game of incomplete information, which means that the traditional AI method of calculating possible moves using minimax trees is not theoretically necessarily the most optimal method.
* UCI Poker Hand Data Set has a large collection of poker hands, along with wins, which could be used for a statistical analysis of the value of each hand. However, this dataset is only relevant for 5 card draw.
* The Monte Carlo method may be a promising way to explore the trends in behaviour of a system pre-programmed in a particular way (eg. Tight aggressive).
* Neural Networks may be an effective system to explore, because a neural network does not require a huge amount of maintenance after the initial training. Moreover, a different set of weightings could be easily mapped to different playing personalities, based on different reward incentives during training.
* Loosely players can be seen through two different spectrums: Tight vs Loose and Passive vs Aggressive, in terms of play style.

## Potential Solutions

**Extension of the Current System – Pseudo-Manual System**

One potential solution would be to extend the current system. One of the main issues with the current system is that the system depends on the knowledge and understanding of the game of the developer. Consequently, the current system could be improved with deeper patterns to check and a more nuanced patterns to check for.

For example, the current system has the drawback of being too predictable, or its behaviour is too linear, to amend this deeper calculations could be used. Using the example the client gave me, the system, at the moment, will always fold when presented with a large raise, as it determines that the amount that the user bets is proportional to the value of their hand – ie it does not take into account the possibility of bluffing. To amend this the system could be extended to add a calculation to determine how skewed that bet is, compared to the likely value, or maximum value of the hand. Extending the system like this would add a dimension behaviour, and the system would be less predictable – the benefit of this is that it is quite easy to extend the system in such a way (just add more patterns to be checked) as well as this it is a more straight forward design and implementation, as long as the developer understands the underlying theory of the game behind it.

On the other hand, the system still retains of its biggest drawbacks – it will take a lot of time and resources to develop a system like this, because, there are a lot of different possible states for a game like poker, and whilst it is possible to generalise these states, to a degree, if one common scenario is missed out and the system defaults to a linear playstyle, then the system becomes very easily exploitable and has failed its requirements. It is quite hard to depend on the robustness of a system designed like this. In addition to this, it will require a lot of maintenance, because, it is unlikely that all the scenarios will be accounted for on with its launch, as a result the system would require constant updating to account for these scenarios, moreover, until it is all amended, the system will not have fulfilled its goal.

Lastly, to fulfil the requirement of personalities as a part of the system, this would involve even more work as the developer would have to design and implement several behaviours for the same system, based on one scenario – one for a passive personality, one for an aggressive personality etc. This makes this solution less viable, based on the time scope and resources of the project.

Main Points:

* Easily extendable
* Complicated design process.
* Simple implementation, but high implementation time.
* High risk design – hard to reason about robustness
* High Maintenance

**Combination of Current System and Statistical Analysis - Semi-Manual System**

An alternative solution would be similar to the current system, but with some of the fundamental concepts altered. For example, rather than hard-coding patterns for the system to check against, the system could instead use a combination of hard calculations, combined with statistical analysis to determine its behaviour.

For example, when determining the value of a hand, the system could still use a subroutine to determine the theoretical value of its own hand, and the probability of the value of the opponents hand relative to their bets, however, when determining its next move, it could use statistical analysis to determine the probability of the current state of the game leading to a win, and then basing its next action upon that, rather than a simple comparison, and pattern matching system as proposed by the current system.

This could be implemented in two different ways, for example, the system could use decision trees to predict future states of the game, and then based on the average value of those outcomes, and it would make its move. This is similar to the current system, however, rather than hard coding these states in, it would be more self-sufficient, by determining the value of its current position by itself. However, this has the potential to be a very inefficient and slow method, as there are so many states that the game could go into, it would take a long time for the system to analyse all of them and determine their value, especially early on in the game. Moreover, if the system took this approach every time, the decision of the system may stagnate as the value of future outcomes may be quite similar to each other, regardless of the move of the system in certain scenarios.

Another implementation of this analysis would be to have a record of previous games, either played by the system, a separate data store, or a combination, then use this data to determine the likelihood of the current value of the position of the system, based on data from previous games. The benefit of an implementation like this is that the system would be make a move more quickly, as long as the data is stored and can be queried in an efficient way. The drawback of this however, is that this system would require a large data store of previous games in order for this to be effective, or else anomalies in the data would have a large impact in the effectiveness of the systems behaviour. This implementation would only be viable if a large store of poker games and results could be found.

Main Points:

* Combination of current system and statistical analysis, in order to reduce linear behaviour of the system and to make the system more generally effective. Similar to last current system, with less maintenance required.
* Implementation of statistical side can be with either decision trees, with values calculated by the system, or from large data store about previous games.
* This system is more self-sufficient as it could improve with more games, making the system less predictable.
* Implementation for personalities is easier as threshold for behaviour (based on calculated value of current position) can be adjusted to emulate different personality behaviour.

**Neural Network / Machine Learning System (Pseudo-automatic)**

Another potential solution would be to model the problem as a neural network and use this model in conjunction with a learning algorithm to generate a system with learned behaviour; this could also be enhanced with use of hard programmed edge cases, or some extra analysis. I could implement this in two ways: either I could use a linear algebra library or as along as it is efficient in its operations I could design my own network and implement algorithms with more freedom to adjust it, alternatively I could use a library with prebuilt neural networks and then program it to fit the problem, which may be more reliable but the trade-off is that I have less freedom in the design and implementation.

This solution would involve modelling a game of poker as a neural network, with the condition of each card in the hand of the system, as well as the state of the pot and community cards as the features, or inputs, and the output would be a single neuron of how much to bet/raise (or if it is 0, or a low number, then fold). Poker is a game of incomplete information, which makes it hard to calculate accurately the value of the current position of the system, as well as using hard calculations to map patterns with the state of the game to linear behaviours, as a result, a neural network model with well-learned weights could be a better solution for producing an effective AI. The reason a neural network would be used, rather than just a classification or polynomial regression system is that it can more easily be used to implement a polynomial system, however, either of those could work, as there would not be a huge number of features in this system.

In conjunction with this would require a learning algorithm. There are a few options in this case, for example, if I could find a large data store of poker games, I could alter the system to optimise a supervised learning problem, and use an algorithm such as gradient descent, to try an learn weights which are most effective for a certain input state. This has the issue for potentially being a bit predictable, and is susceptible to the data (this entire solution is susceptible to the data, but this one more than others) if it is a record of games which are played extremely non-optimally, then the system will try to emulate this and play equally as bad.

An alternative would be an unsupervised problem, where the system would use something such as reinforcement learning or neuro-evolution algorithm to train the system based on games it has played. The issue with this is that, before the system is viable, it would have to play a large number of games to learn from, this would entail having to play a lot of games against a human user (unviable) or another AI. This to some degree solves the issue, of the requirement of a large data store of poker games, because the system, in this case, could be set up to play games against itself, and then adjust the weights of its network, based on which moves worked and which ones did not – if the number of games played is high enough, this should eliminate anomalous moves, or “cheese” strategies which results in a short term win against an AI, but which would be spotted with a human user – such as maybe all-in’ing every round, to intimidate the opponent into folding. Another issue arises in this case, of being stuck in local optima, to amend this the system can be initialised with many random weights, and then pick the highest effective weighting to apply to the system, or pre-programming some initial behaviour to bring the system up to a given level, rather than just a random level, and letting it train from there.

In relation to the issue of personalities, it would be simpler to implement that in this system, as the system could be retrained with either a random weight, or its most effective weighting from the default training, and change the reward calculation (for example, place a higher emphasis on short term winnings for an aggressive personality), then retrain the system, for each personality.

Lastly, this system would be the most self-sufficient out of all the proposed systems, because it each game it plays it would be able to add this new game to its training, resulting in a non-linear behaviour, as each batch of games would be used to adjust its behaviour. This would result in a less predictable system which requires less maintenance.

Main Points:

* Model game as a machine learning system with a few features – the low number of features means efficient processing and learning.
* Weights can be learned either supervised (if database of poker games can be found), or unsupervised (reinforcement learning / neuro-evolution algorithm).
* Personalities somewhat simpler to implement – as reward for the system can be adjusted in relation to the desired output, based on personality types (eg. Higher emphasis on short term winnings for more aggressive personalities)
* Self-sufficient as each game it plays can be used to change its behaviour – this also makes the system less linear.

**Other Potential Solutions:**

* The developer changes their design of the final product, to just being a multiplayer game – thus eliminating the need for an AI system entirely. Not viable, because this is the clients choice, not mine
* Buying a retail AI made for playing poker, and adapt it to fit the current system. This is unviable, because prices of Poker bots can be upwards of a thousand dollars – this is not a reasonable price range for the scope and requirements of this project.

## Proposed Solution

My proposed solution is going to be the Neural Network based solution. I have picked this one, because I feel that it would be the best for the incomplete nature of poker, for the reasons outlined above. Consequently, this means that the system would be effective against higher skill players, and it is easier to purposely limit a system one it has shown that it can perform better, than picking a less effective method and then trying to push past its optimal ceiling.

In addition to this is potentially the best solution to implement the most difficult, arguably, aspect of the system – the personalities. This can be done by changing the reward system (if I use a reinforcement learning algorithm, for example) to correspond to each of the personalities (detailed above). For the other solutions, I would have to adjust with their calculated output manually, and this increases the risk of the system being more linear than desired.

As well as this, for a development team this size, minimal maintenance is paramount, which is another reason why this solution is the most optimal. Due to the nature of the system being able to adapt with beyond the release. Moreover, this quality of utilising a datastore of games to adapt, means that I can use some of the aspects of the statistical analysis solution in conjunction with this solution; I can go further and even hard code some behaviours, if I feel they are fundamental enough, and that the system is not exhibiting them.

Overall, this solution is the most versatile, and adaptable solution, which makes it an optimal choice.

### Proposed Solution DFD

**Level 0 DFD**

The level 0 DFD remains the same, as it is performing the same high level process, however, it will be doing it in a more optimal way.



****

### DFD for Game System



### ERD for Proposed Solution



### Proposed System Neural Network Architecture



On each of the connections of the neural network, there will be a weighting applied , which will correspond to how much “weight” that neural connection has in the output of the network. Here the output will be a n integer, which will correspond to how much the network wants to bet, alternatively how much value the AI places on its current position, and if it is 0 or less than the minimum bet amount then this will correspond to the AI folding. These weightings will be generated using a form of training – most likely reinforcement, however, neural evolution will also be tested.

The activation function for each neuron will be a linear activation function. I have chosen this because I want the network to process the value of its current position, or potential future value of its next move, as a result, a linear activation function fits this need best – as the higher the value of its current position, the higher the bet should be.

This bet will also be processed through another function, to change the final bet, so that the bet can be adjusted for each of the personality types. For example, if the designated personality was more aggressive, it would have a different weighting on its current cards and the pot, compared to other personalities, however, the output bet would be changed to be higher than that of a passive player. An alternative to this could be to use a different activation function, for each of the different personalities.

## Prototyping and Critical Path

The most difficult part of this project will be developing a competitive base AI, which can play the game to a level which is suitable to the user. As a result this is the first part of the program which I will prototype. To do this I will first develop the neural network, and train it using a reinforcement learning algorithm, by having it play against another iteration of itself with different randomly generated weights which will also be improved with the same algorithm. After a designated number of iterations, I will play 5 games against it and determine if I can find any exploitable or predictable patterns, after this I will ask a friend who is more experienced in poker, to also play 5 games against it for the same goal. If this is not effective, I will attempt to develop a new set of weights, either my more iterations, or by using a different learning algorithm, in this case I would use a neural evolution algorithm to develop new weights, and then use the same process to test how effective the AI is.

After this the next most difficult part of the development will be either, to make the AI difficulty adjustable, or to develop the personality types for the AI. I will develop the system for the personality types first, and then use the same testing method to determine if the AI is still a viable opponent, I will also use the same testers for each game to see if its behaviour is different compared to previous games played against it – to see if its behaviour has changed. In addition to this, I will also have the client play same games against the AI to see if the different personality types I have put in play fit in with what he wants from a design perspective.

One this has been completed, I will develop the system to adjust the difficulty of the AI. I will develop this part, and then just use myself or another new player to Poker to determine if the AI is within a competitive difficulty. As it has already been tested from a baseline perspective against a more experienced poker player before this, it should already be competitive at the highest level this AI has been designed for, therefore I would only need to test it against lower skilled opponents. Essentially this is like maximising the playing level of the AI and then limiting it for more casual users, as this is easier than developing it for casual users and then attempting to increase its ceiling. In addition to this I will implement a system for the AI to detect if the difficulty is skewed for its opponent, and to adjust its difficulty automatically if it is. To test this I will set it to its maximum level, and then play against it and either make purposely bad moves, or just play normally, and see if it adjusts its difficulty level after several losses in a row.

The next most challenging part to develop will be the adaptation against particular users, although this is arguably more challenging to develop compared to than the difficulty adjustment, this is a less important aspect compared to that more fundamental objective. To develop this, I will have to develop the database which will store the games played, and the users played against. Once this has been set up, I will design a log in system for the user, and then play against the AI with a particular style in mind, to see if its behaviour changes, if it does then I can have confidence that it fills out this objective, to some degree.

After this, the bulk of the system should be prototyped and tested. Therefore, after this the full interface should be developed, tested and polished, so that the AI is ready to be deployed, without opportunity of being potentially exploited, due to an interface which provides too much access.

# Documented Design

## BlackJack Prototype

### Introduction

Although the user’s focus game is Poker, he has also designed his game around Standard Deck card games in general. As a result, I have started out with designing and prototyping an AI which can play against a dealer, to a competitive level. Here, competitive level is evaluated on the winrate of the AI over 1000 games, with 45% and higher considered a good win rate for blackjack[[10]](#footnote-10). Blackjack is a good game to prototype the AI design, because it has a lot of overlapping elements with Poker:

* They are both games of incomplete information – it is theoretically impossible to calculate the perfect move, as you never have full information about your opponent’s position, you can only infer it from their behaviour
* Both are games of progressive stages – the game final position of each player is not known until later stages of the game, when each hand has been fully dealt.
* Both have betting aspects, however, blackjack is slightly simpler with less rounds of betting, mostly one round, an optional second round if the player decides to double down, whereas poker has several rounds of betting.

In addition to this, it is a good game as a starting point to being prototyping, because it is also simpler, because:

* In blackjack, there is only one opponent: the dealer.
* The dealer behaves in a predictive way: they will hit until they reach a hand value of 17.
* There are less total cards in play, and to keep track of (If this is a strategy used by the AI)
* The way the value of a hand is determined by each card’s numerical value, whereas in poker a hand’s value is determined by what combination of cards the hand and the community cards match.

As a result, I have begun with designing and prototyping a game of poker, where I will utilise similar elements from the chosen design of the final system, and adapt it to blackjack.

### Design – The Blackjack Game and Interface

I have developed my own interface for the blackjack game. The details of the methods have been provided to me by the client:

As of January 2018, this is the current composition chart for the blackjack game.



As well as the current Inheritence Chart:



#### Blackjack Class Design

|  |
| --- |
| Blackjack |
| - Deck : Deck Structure  - Royal Values : Hashing Structure (Such as Dictionary or HashMap)  - Blackjack : Int (Constant)  - Player Hand : Hand Structure  - Dealer Hand : Hand Structure  - Bust : Boolean  + Continue Game : Boolean |
| + Constructor  + Reset Game  - Evaluate Hand Value (Hand : Hand) : Int  - Determine Ace Value (Hand Total : Int, Number of Aces : Int): Int  - Compare Hands (Hands\* : Hand) : Boolean or Int  - Deal(Hand : Hand) : Hand  - Deal Dealers Cards at End : Hand  + Hit : Hand  + Stand  + Output Current Game State to Console  + End Game |

Below are descriptions of the attributes and the methods described above.

**Methods:**

|  |  |
| --- | --- |
| **Method Identifier** | **Description of Method** |
| Constructor | Declare and Initialise all attributes and constants. |
| Reset Game | Restore Attributes to starting values |
| Evaluate Hand Value | Return the Value of the Hand |
| Determine Ace Value | Return whether the Value of the Ace would be better as 1 or 11 |
| Compare Hands | Compare hands passed in and determine a winner or a draw. |
| Deal | Deal the initial two cards to each player |
| Deal Dealers Cards at End | Deal Cards to Dealer until their hand value is greater than 17 or bust. |
| Hit | Deal a card to the player. |
| Stand | End the Game |
| Output State of Game to Console | Display the state of each player’s hand, and value. |
| End Game | Call the methods which determine a winner at the end of the game: Deal Dealer’s Cards at end, and Compare Hands. |

**Attributes:**

|  |  |
| --- | --- |
| **Identifier** | **Description of Purpose** |
| Deck | Stores the Deck of Cards to Draw from |
| Royal Values | Dictionary, storing the values of each of the royal cards. |
| Blackjack | Constant holding the value for a blackjack (21) |
| Player Hand | Hand structure containing the cards currently in the player’s hand. |
| Dealer Hand | Hand structure containing the cards currently in the dealer’s hand. |
| Bust | Boolean holding the state of whether the player is bust or not. |
| Continue Game | Boolean holding the state of whether the game has ended or not. |

The behaviour of this prototype will be discussed later. First let me present the data structure design used in this prototype, and potentially for the poker prototype also.

Updated January 2018:

Due to the fact that I updated this specification to include more encapsulation for the hand class and the behaviours associated with the hand, some of the functionality has been removed from here and added to the hand class. In addition to this, I have modified the behaviour of the blackjack class so that it keeps the deck between games, until the deck runs out of cards, at which point the deck is reinitialised and shuffled.

Below is the updated class diagram:

|  |
| --- |
| Blackjack |
| - Deck : Deck Structure  - Royal Values : Hashing Structure (Such as Dictionary or HashMap)  - Blackjack : Int (Constant)  - Players Queue : Circular Queue of Hands / Players  - Winners : Hand[]  + Continue Game : Boolean  + Deck Iteration : Int |
| + Constructor  + Reset Game  - Create Player Queue(Structure : Hands/Players)  - Compare Hands (Hands\* : Hand) : Boolean or Int  - Deal(Hand : Hand) : Hand  + Hit  + Stand  + Output Current Game State to Console  + Check if Game is Over  + End Game |

Below are descriptions of the new methods or attributes.

**Methods:**

|  |  |
| --- | --- |
| **Method Identifier** | **Description of Method** |
| Constructor | Declare and Initialise all attributes and constants. |
| Reset Game | Restore Attributes to starting values |
| Compare Hands | Compare hands passed in and determine a winner or a draw. |
| Deal | Deal the initial two cards to each player passed in |
| Hit | Deal a card to the current player, determine if they have gone bust or not, and if they have do not repush them back to the player queue. |
| Stand | Update the current hand to stood, and do not repush them to the queue |
| Output State of Game to Console | Display the state of each player’s hand, and value. |
| End Game | Call the methods which determine a winner at the end of the game: Deal Dealer’s Cards at end, and Compare Hands. |
| Create Player Queue | Create a circular queue and push all the starting players to the queue. A structure of starting players should be passed to the method. |

**Attributes:**

|  |  |
| --- | --- |
| **Identifier** | **Description of Purpose** |
| Deck | Stores the Deck of Cards to Draw from |
| Royal Values | Dictionary, storing the values of each of the royal cards. |
| Blackjack | Constant holding the value for a blackjack (21) |
| Continue Game | Boolean holding the state of whether the game has ended or not. |
| Players Queue | Circular Queue holding each player still in the game |
| Winners | Array of Hand IDs holding the winner of the last game played. |
| Deck Iteration | Hold the value of the current deck iteration – if a new deck has been initialised then increment this. |

#### Structures

##### Card

Firstly, the fundamental object of blackjack (and all the games in this project) is the Standard Deck Card. This card structure has a very simple design:

|  |
| --- |
| Card |
| # Value : Integer or Royal (Enum : Int)  # Suit : Suit (Enum : String) |
| + Constructor  + Get Value : Int or Royal (Enum : Int)  + Get Suit : Suit (Enum : String)  + String Casting Override : String  + Equivalence Override (Other Instance in Comparison : Card) : Boolean |

**Methods:**

|  |  |
| --- | --- |
| **Method Identifier** | **Description of Method** |
| Constructor | Declare and Initialise all attributes and constants. |
| Get Suit | Returns the suit value |
| Get Value | Returns the value associated with the Card |
| String Casting Override | Changes the way an instance of card is converted to string – returns the suit and the value of the card. |
| Equivalence Override | Change the way instance equivalence is evaluated to comparing each instance’s value and suit, returning True if those attributes are equal. |

**Attributes:**

|  |  |
| --- | --- |
| **Identifier** | **Description of Purpose** |
| Value | Holds the Value of the Card Object |
| Suit | Holds the Suit of the Object |

For the purposes of blackjack, only the card value is relevant, however, this class can be reused later for the poker prototype. In addition, it provides an easy method for other structures (such as the deck or the blackjack game) to be able to abstract away more detail into this class, making them easier to maintain. In addition, it provides an easy interface for different classes to determine the value of the royals or the Ace, as for every game this will appear as an Enum of Royals, in which case a hashing structure can be used to map each of these cards to a concrete value. This is better than having to hard code different values of the royals to different settings – for example, the alternative would be to hard code methods into the card or deck structure which changes the value of the royal from within these structure, rather than from within the game structure. This would mean having to store unnecessary data and behaviours within the card and deck structure, making them harder to maintain and use.

##### Stack

The stack structure is what I have used to implement the deck structure. I have used an implementation we made in class for this purpose. Its design is shown below. It is implemented backed by an array.

|  |
| --- |
| Stack |
| - Size : Int  - Top of Stack Pointer : Int / Address  - Stack Array : <T> (Data type of stack) |
| + Constructor (Stack Size : Int, Type of Stack : <T>)  + Size Getter : Int  + Push (Element to be pushed : Element of Stack) : Boolean  + Pop : <T>  - Is Full : Bool  - Is Empty: Bool |

**Methods:**

|  |  |
| --- | --- |
| **Method Identifier** | **Description of Method** |
| Constructor | Declare and Initialise all attributes and constants. |
| Size Getter | Returns the value of the attribute holding the value of the size of the stack. |
| Push | Pushes a new item to the top of the stack, returns true if the push was successful. |
| Pop | Pops the element from the top of the stack |
| Peek | Outputs the element at the top of the stack. |
| isEmpty | Returns a Boolean determining if the stack is empty or not. |

**Attributes:**

|  |  |
| --- | --- |
| **Identifier** | **Description of Purpose** |
| Size | Holds the maximum size of the stack |
| Top of Stack Pointer | Pointer pointing to the top of the stack. |
| Stack Array | The fundamental array holding the data of the stack. |

##### Deck

The deck structure is what I will be using the store the cards, and then deal them from within the blackjack game. I implemented the deck as a stack, and utilised inheritance to specialise the Stack behaviour for the Deck context. A stack was most appropriate because the first in, last out behaviour was suitable for this context, as you only need access to the top card of the deck at any instant of time. As the deck inherits the Stack structure, it has all the methods and attributes, which are either public or protected, which are shown above.

|  |
| --- |
| Stack |
| - Size : Int  - Top of Stack Pointer : Int / Address  - Stack Array : <T> (Data type of stack) |
| + Constructor (Stack Size : Int, Type of Stack : <T>)  + Size Getter : Int  + Push (Element to be pushed : Element of Stack) : Boolean  + Pop : <T>  - Is Full : Bool  - Is Empty: Bool |

|  |
| --- |
| Deck |
| - Values : Int Array  - Suits : Suits Array (Suits -> Enum: String) |
| + Constructor  - Initialise Deck |

**Overridden / Deck Methods:**

|  |  |
| --- | --- |
| **Method Identifier** | **Description of Method** |
| Constructor | Declare and Initialise all attributes and constants. |
| Initialise Deck | Generates a queue of all the cards to be pushed to the deck, shuffles them and pushes them to the deck |

**Deck Attributes:**

|  |  |
| --- | --- |
| **Identifier** | **Description of Purpose** |
| Values | Holds all the different values of each card to be pushed to the deck |
| Suits | Holds all the different Suits of each card to be pushed to the deck |

##### Circular Queue

As the game has a variable number of players, a data structure is required to store the players, so that they can be accessed. I have used a circular queue for this, as it has the double purpose of also providing FIFO behaviour, which will provide me with the next player who needs to have their go. This works by popping from the queue at the start of each turn and then, as long as they are still in the game, pushing them back into the queue. Moreover, if they are out of the game (either by going bust or standing) then they are not pushed back onto the queue, so the end of the game can be determined by whether or not the queue is empty.

Moreover, a circular queue was used, because it is more time efficient than a normal static queue. This is because, with a static queue, after each pop all the data stored in the queue has to be moved down the list; this has a big O complexity of O(n), whereas the cost of popping for a circular queue is O(1), as the front and rear pointers have to just be adjusted. The only downside is that an extra pointer has to be stored, but the memory cost of one extra pointer is not significant compared to the efficiency cost of a normal static queue.

A static queue was utilised, rather than a dynamic one, because the number of players in each game is known at the start of the game, more players cannot join, and if a player leaves it does not have an impact on the queue.

|  |
| --- |
| Circular Queue |
| - Size : Int  - Front Pointer : Int / Address  - Rear Pointer: Int/Address  - Circular Queue Array : <T> (Data type of stack) |
| + Constructor (Queue Size : Int, Type of Stack : <T>)  + Size Getter : Int  + Push (Element to be pushed : Element of Stack) : Boolean  + Pop : <T>  + Peek: <T>  - Is Full : Boolean  - Is Empty: Boolean |

##### Hand

The hand structure is what will be used to store the hand of each player. In this case the hand is only ever added to, using the hit behaviour. In this context, an array or linked list are just as appropriate as each other, the direct access of an array is not as significant in this context, however, the maximum number of cards a player can have before bust or blackjack is known (11 – four 1’s, four 2’s and three 3’s), so the dynamic advantages of a linked list are not quite as significant either. However, for the sake of versatility for other standard deck games, a linked list is the best structure, because it will support hands of any size, efficiently; unless you passed in the maximum hand size to the constructor of the structure with the use of an array, however, this makes the structure harder to maintain. Either way the implementation of the hand structure is not as clear-cut as the use of a stack to implement the deck, however, I have chosen a linked-list based structure.

For now, there are no extra behaviours required for the Hand structure. The only attributes would be the cards that the structure holds.

Updated January 2018 –

Upon updating the blackjack class to accommodate for more than one player, I realised it would be a better design to take some of the behaviours built into the blackjack class regarding the hands, and move them into the hand class. Fundamentally the hand class is still backed by an array of cards, however, I have updated the class to include built in behaviours such as getting the current value of the hand, and choosing the ace.

In addition to this, I have also added a new class for the dealer’s hand, which holds all the functionality for the dealer, so that it does not have to be in the blackjack class. This class inherits from the hand class above and has some additional behaviours.

The new class design is as follows:

|  |
| --- |
| Hand |
| - ID : String or Hash Value  - Hand : Card[]  - Stood : Bool  - Bust : Bool  - Royals : Hash Structure (Card -> Integer) |
| + Constructor (Id: String)  + Get Hand Value : Int  - Choose Ace (Hand\_Total : Int, noAces : Int)  + Hit (Card\_To\_Add : Card) : Void |

|  |
| --- |
| Dealer Hand |
| (No additional attributes) |
| + Dealer Play (Deck : Deck) : Void |

**Methods Hand and Dealer Hand:**

|  |  |
| --- | --- |
| **Method Identifier** | **Description of Method** |
| Constructor | Declare and Initialise all attributes and constants. |
| Get Hand Value | Calculates and returns the current value of the hand which the instance holds. |
| Choose Ace | Private method which assists in calculating the value of the hand by picking the value for the ace. |
| Hit | Appends the passed card to the array of cards attribute. |
| Play Dealer | Whilst the value of the instance of the dealer’s hand is below 17: pop cards from the passed deck and hit them to the hand. |

**Attributes:**

|  |  |
| --- | --- |
| **Identifier** | **Description of Purpose** |
| ID | Identifier variable used to identify the instance of the class |
| Hand | Array of cards which is the literal contents of the current instance of the Hand. |
| Stood | Tracker variable keeping track of if this player has stood. |
| Bust | Tracker Variable keeping track of if this player has gone bust. |
| Royals | Hashing structure which maps Instances of Royal Cards to their values within the current game. |

#### Blackjack Class Usage

From a high-level perspective, the blackjack interface provides a hand to each player, and the game mainloop continues until each player stands, reaches a blackjack or bust via hitting. Once the game mainloop ends, the end game method is called to calculate the value of each hand and determine a winner. The game can then be reset using the reset method.

Updated January 2018:

Upon adding the player queue, allowing for multiple players in one game of blackjack, the new functionality of the class is as follows.

During the class mainloop, the current player is output, as well as the value of their hand. Then the player or the AI can decide to hit or stand. This hit or stand functionality is done through one function for each of the behaviours, this method has the same behaviour, however, it performs this behaviour on the hand at the front of the queue which will be popped. If the player stands or is bust, they are out of the game and are not pushed back to the queue. The game ends when there is no more players in the player queue.

After this the an end game method is called, dealing cards to the dealer until the value of their hand is above 17, and then another method is called comparing the value of each hand, and then outputting the winner.

### Design – Card Counting AI

The blackjack dealer AI behaves in a very linear fashion – it will always hit until it reaches at least above 17. Reaching winrate of at least 45% (as provided by my research) is one metric which could be used to assess the performance of the final product. However, another metric would be useful in determining the aptitude of the automatic machine learning system. As a result, a Card Counting AI will be designed and developed alongside it, to compare how well they perform contrasted against each other. This is useful because the card counting AI’s behaviour is more consistent, predictable and easier to dissect, compared to the automatic system.

This AI will utilise a card counting tactic to predict whether it is optimal to hit or stand, based on what cards have already been revealed. The agent will store a record of all the cards in one standard deck and a counter next to each one, which will be decremented as each card is played. From this it will determine the probabilities of the next card it draws resulting in a bust, blackjack or less than 17 (or the dealer’s current hand value). Parameters will have to be tested and chosen which will correspond to the threshold that the agent will operate under before making a move – for example, I will test if the AI should hit up to it predicting a 50% bust chance or less or higher.

#### Structures

##### Complete Binary Search Tree and Node

Storing the cards and the values associated with them are best stored in a Binary Search Tree, this is because as long as I store all the cards in their value order, then searching for each card has a Big O complexity of O(log n), compared to if I used an array, which would result in a Big O complexity of O(n). Although the data set in this scenario is fixed for a small n, it makes the system more extendable.

There are a few ways I could have implemented the binary tree, however, the way I have chosen to do it, is by creating a Node class which essentially stores a value and can point to a left child and a right child, the node will also have auxiliary behaviours (such as get number of children). The nodes of a tree are linked via pointing of the node objects. The Binary Tree class has one attribute – the root node, through which all the other nodes in the tree can be accessed via tree traversals. Most of the methods utilise tree traversals to operate on the tree, or to define a behaviour.

The advantage of this is that it makes some behaviours – such as searching and inserting very quick and easy to program, using tree traversals; at the same time, it makes some other behaviours harder to program, for example, the maintenance of the binary tree. In general, traversing the tree and operating via traversals is much easier to write code for, however, using this system I am limited to traversals, and there is not direct access to nodes, which is why the maintenance of the tree is harder to write, as the tree can be only be manipulated by their pointers – as a solution to this I have written my own algorithm to handle this.

As well as this, I have created a static class of traversals within which lexically closed functions for the node processing and the base case can be passed to one of the three traversal methods (pre, in, or post order traversal). The use of this class makes writing the functions for this class even easier and faster, as most of the boiler plate code for the recursion is abstracted away, in addition to this, most operations required two methods – handling input data, and then the actual recursion function, the use of this static class allows me to keep the code compact by abstracting away the recursion methods.

The node class and the Binary Tree class have been extended from general purpose to a more specific purpose – card counting. I have achieved this by extending the classes via inheritance to add an extra attribute of “card count” to the node – this means along with the value of the card, a Card Node will also have an attribute which will track how many of the corresponding card is left in the deck. This is the main fundamental change, the extensions to the binary tree to the binary card tree extend the tree around this by providing additional behaviours around the card count attribute. One such example is counting the number of cards in a tree equal to or greater than a given value, which utilises traverses returning the count value rather than the node value.

##### Binary Search Tree and Card Counting Tree Class Diagram

|  |
| --- |
| Binary Search Tree |
| + Root: Node object |
| + Constructor(root : node) : Void  + Get Root() : Node  + Insert(toInsert : Node) : Void  + get\_tree\_size() : Int  + maintainTree() : Void  - swap\_max\_LST(swapRoot: Node) : Void  - swap\_minRST(swapRoot: Node) : Void  + get\_max\_LST() : Node  -get\_min\_RST(): Node  + Delete(node) : Void  - Delete\_oneChild(node, nodeParent, nodeIsLeft): Void  - Delete\_twoChildren(node, nodeParent, nodeIsLeft) : Void  - Delete\_noChildren(node, nodeParent, nodeIsLeft) : Void  - Delete\_root() : Void |

|  |
| --- |
| Card Counting Tree |
| (No additional Attributes) |
| + decrement(nodeValue: Int) : Bool  + cardCountGTET(baseNode: CardNode) : Int  + totalCardCount(baseNode: CardNode) : Int |

|  |
| --- |
| Node |
| + nodeValue : Int  + Left : Node  + Right : Node |
| + hasLeft() : Bool  + hasRight() : Bool  + numOfChildren(): Int  + equality override(other : Node) : Bool  + String Cast Override() : String  + Greater Than / Greater Than Or Equal To Override(other : Node ) : Bool  + Less Than / Less Than Or Equal To Override(other: Node) : Bool |

|  |
| --- |
| Card Node |
| + cardCountValue : Int |
| (No additional Methods) |

#### Algorithms

##### Insertion and Deletion of Nodes within the Binary Tree

In this particular scenario insertion does not occur very often – it will only occur during the initialisation of the tree, when all the cards are to be inserted, this is because cards cannot be pushed to a deck, unless the deck is reinitialised. On the other hand, deletion will occur often, whenever the number of cards the node is associated with reaches 0.

Both of these algorithms are essential to the concept of a binary tree. As the binary tree will be balanced and maintained, the big-O complexity of insertion will be O(log n), because insertion is essentially the same operation as searching for a node, except with an insertion operation at the end. Whilst the explanation and the pseudocode could be considered one and the same for both the insertion and searching, I have implemented these operations in two different ways – the search algorithm (discussed later, function name = getNode) has been implemented as a pre-order traversal, it could also be implemented as a breadth first search; whereas for the insertion operation, I have implemented it utilising a while loop:

###### Insertion Pseudocode

FUNCTION insert(node)  
 IF getNode(node) <> Null THEN # If the node already exists in the tree

RETURN False

ENDIF

IF currentRoot = Null THEN # No root at the moment

Root <- node

RETURN True

ENDIF

nextNode <- Root

lastParentLeft <- True

WHILE nextNode <> Null

IF node.value < nextNode.value THEN

nextNode <- nextNode.left

lastParentLeft <- True

ELSE IF node.value > nextNode.value THEN

nextNode <- nextNode.right

lastParentRight <- False

ELSE

RETURN False # should never reach this point

ENDIF

ENDWHILE

IF lastParentLeft = True THEN

nextNode.parent.left <- node

ELSE

nextNode.parent.right <- node

ENDIF

MaintainTree()

RETURN True  
ENDFUNCTION

The deletion of nodes is a bit more involved in that there are a few edge cases in which deletion can become more or less complex. The different scenarios relate to the number of children the node to be deleted have. Consequently, cost of this operation could vary between constant and one recursion.

If the node to be deleted has no children, then it can simply be deleted from the tree – this is the simplest case. If the node to be deleted has one child, then the node’s parent should be connected to the node’s child, and then all pointers to the node can be removed. If the node to be deleted has two children, then the largest node in the node’s left subtree should replace that node – and then the maximum node in the left subtree should be deleted from its old position – this is guaranteed to recur once, because it can only have one child, as it is the maximum node in the node’s left subtree (it cannot have a right subtree).

The last edge case is when the root of the tree is to be deleted, this has to have a slightly different operation, because the principle of everything on the left of a binary tree must be smaller than the root, and everything in the right subtree must be larger has to be maintained. If the root has no children, then this can be handled the same as normal. If the root has one child, then get either the minimum node in the right subtree, or the maximum node in the left subtree, use this as the new root and delete its old position – based on what subtree exists currently within the tree. Lastly, if the root has two children then the same operation can be used as with one child, however, it should be a consistent node used, for example, if there are two children I will always use the minimum node in the right subtree as the node to use as the new root.

###### Deletion Pseudocode

FUNCTION Delete(node)

NumChildrent <- node.numOfChildren()

IF node = root THEN

Delete\_Root()

ELSE

# executes pre order traversal – similar to getNode method

nodeParent <- getParent(node)

nodeIsLeft <- nodeParent.left = node

IF numChildren = 0 THEN

Delete\_noChildren(node, nodeParent, nodeIsLeft)

ELSE IF numChildren = 1 THEN

Delete\_oneChild(node, nodeParent, nodeIsLeft)

ELSE IF numChildren = 2 THEN

Delete\_twoChildnre(node, nodeParent, nodeIsLeft)

ENDIF

ENDIF  
ENDFUNCTION

FUNCTION Delete\_noChildren(node, nodeParent, nodeIsLeft)

IF nodeIsLeft THEN

nodeParent.left <- node

ELSE  
 nodeParent.right <- node

ENDIF

ENDFUNCTION

FUNCTION Delete\_oneChild(node, nodeParent, nodeIsLeft)

childIsLeft <- node.hasLeft()

swapDestination <- Null

child <- Null

IF nodeIsLeft THEN

swapDestination <- nodeParent.left

ELSE

swapDestination <- nodeParent.right

ENDIF

IF childIsLeft THEN

Child <- node.left

ELSE

Child <- node.right

ENDIF

swapDestination <- child

ENDFUNCTION

FUNCTION delete\_twoChildren(node, nodeParent, nodeIsLeft)  
 max\_LST <- get\_max\_LST(node)

Delete(max\_LST)

max\_LST.right <- node.right

max\_LST.left <- node.left

IF nodeIsLeft THEN

nodeParent.left <- max\_LST

ELSE

nodeParent.right <- max\_LST

ENDIF  
ENDFUNCTION

FUNCTION Delete\_Root()

IF Root has no children THEN

Root <- Null

ELSE

IF Root has left child THEN

swapNode <- get\_max\_LST(root)

ELSE

swapNode <- get\_min\_RST(root)

ENDIF

swapNode.left <- root.left

swapNode.right <- root.right

root <- swapNode

ENDIF

ENDFUNCTION

##### Maintenance of Binary Search Tree

The main benefit of using a binary search tree is the fast access time of O(log n). However, this does benefit is not gained, unless the BST is balanced. An example of an unbalanced binary tree is shown below:

This is the worst case scenario, where the data is inserted into the tree in ascending order. At the worst case, the binary search tree effectively becomes a linked list, and the search time complexity becomes O(n) rather than O(log n), and we lose the benefit of using the BST at all. Although this is the extreme case, there are likely to be other cases where parts of the tree are balanced and other parts are unbalanced, which would result in a search complexity between O(n) and O(log n). Consequently, an algorithm made to maintain the structure of the BST would be useful.

There are a few ways this could be done. One of the ways is to order the data as it comes in and store it in an array. Then the data could be split up into an upper and lower part, where the middle of the upper part of the list and the middle value of the lower part of the list is inserted after the middle value of the data is inserted as the root. This guarantees a balanced binary tree, however, this would require either the full data set to be passed to the tree, as it built, or to have to rebuild the entire tree from the beginning every time a new value is inserted. This also means that the data would have to be tracked in an array as well as the binary tree, which increases the space complexity of the tree. Whilst this could be suitable for the scenario, as we know the full data set before hand, a range of values between 1 and 11, however, using this algorithm would make the BST less extendable for the reasons stated above.

Another way of maintaining the structure of the tree would be to automatically update the structure of the tree as new nodes are added. Since I am storing the tree as Node objects pointing to other node objects, I can simply update the pointers of the nodes to adjust the structure of the tree. The complexity of this algorithm appears, firstly from detecting when the tree is unbalanced, and updating the pointers.

I have designed and implemented an algorithm which does this. From a high level, the algorithm works like this:

1. A Post order traversal is used to count from the bottom up how many children each node has – each recursion of this traversal returns the value returned by each of its children + 1. If a node is None then 0 is returned.
2. Before returning a value, the number of nodes in the current nodes left subtree (returned by the left subtree call) and the number of nodes in the right subtree (returned by the right subtree call) are compared, and if the modulus of the difference between them is greater than or equal to 2, then the structure of this subtree (where the root is the current node) needs adjusting. This is because when there is a difference of two children in either subtree, there is an imbalance which can be rebalanced by changing the root.
3. Swap if the LST has more nodes, swap the root with the max node in the LST. If the RST has more nodes, then swap the root with the minimum value in the RST (more detail in the pseudocode)
4. Return False, to flag that a node has been swapped and the checking should start again

This algorithm would have time complexity of O(n2). Whilst this is a high time complexity (although it is still polynomial), it is worth it in this context, because this only impacts on the program during the population of the tree. As a result of this the searching time complexity becomes O(log n). As the tree will be searched very often during the game, the overall time complexity makes this algorithm worth it.

###### Pseudo Code

FUNCTION Maintain\_TREE()

Completed\_Comparing <- False

While Completed\_Comparing is FALSE

Traversal\_Result <- Maintain\_Traverse(tree\_root)

IF Traversal\_Result <> -1 THEN

Completed\_Comparing <- True

ENDIF

ENDWHILE

ENDFUNCTION

FUNCTION Maintain\_Traverse(current\_node)

IF current\_node is Null THEN

Return 0

ENDIF

Left <- Maintain\_Traverse(current\_node.left)

Right <- Maintain\_Traverse(current\_node.right)

IF (Left = -1) or (Right = -1) THEN

RETURN -1

ELIF abs(left – right) >= 2 //Imbalance in the tree – Therefore Balance

IF Left > Right THEN

Swap Current Node with Max Node in LST

ELSE THEN

Swap Current Node with Min in RST

ENDIF

RETURN -1

ENDIF

RETURN (Left + Right + 1)

ENDFUNCTION

##### Generation of Chances via Card Counting

The premise of the card counting AI is to count the cards which have been played in the game, and then base the next move on the probability of drawing a particular card. The behaviour of the CCAI can then be parameterised, based on the thresholds for action on the probabilities generated.

For the blackjack prototype, I have identified three fundamental probabilities which need to be assessed by the AI when generating its next move, the probability that the next card will bring the AI into the following states: bust, blackjack, winning/exceed winning player. I will also include a Boolean parameter called “winning” which will be true when the current AI is winning, the chance that the next hit will result in a winning state will be 100%, as distinct from the next card being certain to bring the AI from a losing state into a winning state.

The way these probabilities are fundamentally generated are the same: find the critical card value which will bring the AI into the state being checked for, add up all the cards which will lead to this state transition, if the AI receives it, then divide this value by the total number of cards left in the deck.

The way this is implemented is that the Binary Tree used to store the card values and the number of cards associated with that value in the deck will be traversed, and any card which is valid for the state transition will be added to the return value.

For going into a blackjack state, there can only be at most one card which will generate this result. Consequently, when counting the cards for the probability of going into blackjack, only the number of cards for the single value needs to be counted, giving a Big-O complexity of O(log n), as the structure used to count the cards is a binary tree. On the other hand, for exceeding the winning player (given that the AI currently losing) or going bust, there are a number of cards which could result in this state transition – any card which is equal to or greater than a critical value. The operation of counting the nodes for a value in the right subtree of the binary tree would have a big-O complexity much less than O(n), because only the number of cards in the right subtree and that turning node would have to be counted. However, if the turning node is in the left subtree of the binary tree, then the process becomes a bit more complicated as the number of cards in its right subtree have to be counted, as all the cards in the nodes above it in the tree hierarchy, as well as all the nodes in the right subtree of the binary tree.

One issue I discovered later on is that some cards may have been deleted from the card counter tree, as all the cards of that value have been played. As a result, sometimes the card I am looking for may no longer be in play. For blackjack, if this is the case, then transitioning to a blackjack state becomes impossible, however, for the other state transitions, this is not as much of an issue, because, there is more than one card which can lead to this state transition. To amend this, if the turning node is not in the tree, the method will iteration up until it finds the next node up which exists, and then use this as the turning node. If the method iterates up and exceeds the maximum value of the tree, then no cards will lead to this state transition and the function will return 0.

The efficiency of this counting algorithm could be maximised by implementing two different methods for these two scenarios, however, this increases the programming complexity massively, and due to the time constraints, I have decided to use a single umbrella method which checks all the nodes in the subtree. Whilst this does have a big-O complexity of O(n), it is much faster to write, and this time can be guaranteed for all nodes. This methods simply executes a post order traversal and counts all the cards values of the nodes which have a card value bigger than or equal to the value of a passed turning node. This method was much simpler to write, as it did not require a node position detection algorithm (detecting if a node is on the left or right side of the subtree) and it only requires a single counting algorithm for all nodes.

The last part of the card counting algorithm is to decrement the nodes in the tree, after each card has been played, and then deleting them when the number of cards left in the deck of that value of card reaches 0. This is achieved by passing in the cards dealt to the CCAI, at the end of a game of blackjack, then decrementing this node, and deleting it if its count reaches 0. This part of the algorithm is quite straight forward, however, there are some edge cases which complicate it a bit. One such case is that every card has been played at the end of a game, in which case the binary tree is reinitialised. The way I have chosen to store the Ace and the Royals in the tree, is to store Jack, King and Queen as 10, and the ace as both 11 and 1; consequently, whenever I decrement the ace I have to decrement both the nodes associated with 1 and 11. Another case is that a deck resets half way through a game, in which case the CCAI has to decrement the binary tree, until it is depleted and then keep the new cards to decrement the new tree after it has been reinitialised. Lastly, when a node has been deleted, the method to maintain the tree to keep a balanced structure is called to keep node searching time at a big-O complexity of O(log n).

From a high level, these algorithms operate in these steps:

Card Counting:

1. Find the current value of the hand of the AI, and then calculate the values of the turning nodes which would result in the transition state of: blackjack, bust, winning. If the turning nodes do not exist, iterate up until you find the next highest turning node, or until you exceed the maximum value of the tree. If the maximum value of the tree is exceeded, return 0.
2. Count the value of the nodes which would result in a state transition – for blackjack, this is just the turning node, for the other states, this would involve counting the cards equal to or greater than the turning node
3. Calculate the probability of the next card resulting in the state transition by counted cards / total number of cards in tree

Decrement and Deletion:

1. Count the number of times a particular value card has been played in the most recent game.
2. Decrement each card value by how many times it has been played in the most recent game. If the value is an ace, decrement both the nodes corresponding to the value 1 and 11.
3. If the count value for a card value has reached 0, after being decremented, delete the corresponding node from the tree.
4. If all the nodes have been deleted from the tree, then reinitialise the tree – the deck will have been reset.

###### Pseudocode

Eg. Calculating bust chance

FUNCTION calcBustChance(hand\_value)

TurningNodeValue <- 22 – hand\_value

IF (TurningNodeValue > MaxCardValue) THEN  
 RETURN 0 # no chance to go bust – card needed is too large

ELSE IF (TurningNodeValue <= 0) THEN

RETURN 1 # Already bust

ENDIF

minExceedValue <- TurningNodeValue

TurningNode <- BinaryCardTree.getNodeByValue(TurningNodeValue, BinaryCardTree.root)

WHILE (TurningNode = Null) and (minExceedVaue <= maximum card value available) THEN minExceedValue <- minExceedValue + 1

TurningNode <- BinaryCardTree.getNodeByValue(minExceedValue, BinaryCardTree.root)

ENDWHILE

IF TurningNode = Null THEN

RETURN 0

ENDIF

numOfBustCards <- BinaryCardTree.cardCountGTET(TurningNode)

totalCards <- BinaryCardTree.totalCardCount()

RETURN numOfBustCards / totalCards

END FUNCTION

**Binary Card Tree**

FUNCTION getNodeByValue(node\_value\_to\_search, current\_node)

IF current\_node is NULL THEN

Pass

ELSE IF current\_node.value = node\_value\_to\_search  
 RETURN current\_node

ENDIF

Left <- getNodeByValue(node\_value\_to\_search, current\_node.left)

Right <- getNodeByValue(node\_value\_to\_search, current\_node.right)

IF Left <> Null THEN  
 Return Left

ELSE IF Right <> Null THEN

Return RIGHT

ENDIF

END FUNCTION

FUNCTION cardCountGTET(turningNode, current\_node)

IF current\_node = NULL THEN  
 RETURN 0

ENDIF

Left <- cardCountGTET(turningNode, current\_node.left)

Right <- carCountGTET(turningNode, current\_node.right)

toAdd <- 0

IF current\_node.value > turningNode.value THEN

toAdd <- current\_node.value

ENDIF

RETURN Left + Right + toAdd

ENDFUNCTION

FUNCTION totalCardCount()

totalValue <- cardCountGTET(self.root)

ace\_node <- getNodeByValue(11)

IF ace\_node <> NULL THEN

totalValue <- totalValue – ace\_node.value

ENDIF

RETURN totalValue

ENDFUNCTION

FUNCTION decrement(card\_value)

Node\_to\_decrement <- getNode(card\_value)

If Node\_to\_decrement = Null THEN

RETURN False

ELSE IF Node\_to\_decrement.value = card\_value THEN

Node\_to\_decrement.value <- Node\_to\_decrement.value – 1

If Node\_to\_decrement.value = 0 THEN

Delete(node\_to\_decrement)

Maintain\_Tree

RETURN True

ENDIF

ENDIF

ENDFUNCTION

**Decrement**

FUNCTION Decrement\_Cards\_From\_Hands(\*args) # \*args will be an arbitrary number of hands

deckUpdated <- False

newCards <- []

FOR hand in args

FOR card in hand

Result <- decrement\_card(card)

IF result = False THEN # this means the card could not be decremented,

because there are no more nodes in the tree

deckUpdated <- True

append(newCards, card)

ENDIF

ENDFOR

ENDFOR

# If deck has updated, reinitialise tree, and decrement the new cards

IF deckUpdated = True THEN

Initialise\_tree()

Decrement\_Cards\_From\_Hands(newCards)

# edge case where the last card dealt is the very last card in the deck

ELSE IF BinaryCardTree.root = Null THEN

Initialise\_tree()

ENDIF

ENDFUNCTION

FUNCTION Decrement\_Card(card)

Result <- Null

IF card.isRoyal() THEN

Result <- Royal\_decrement()

ELSE IF card.isAce() THEN

Result <- Ace\_decrement()

ELSE THEN

Result <- BinaryCardTree.decrement(card.value)

ENDIF

RETURN Result

ENDFUNCTION

FUNCTION Royal\_decrement()

RETURN BinaryCardTree.decrement(10)

ENDFUNCTION

FUNCTION Ace\_decrement()  
 result1 <- BinaryCardTree.decrement(1)

result2 <- BinaryCardTree.decrement(11)

RETURN (result1 OR result2)  
ENDFUNCTION

##### Behaviour based on Aggression Parameters

### Design – Neural Network Based AI

#### Prototype and Progression of Architecture

##### Initial Prototype Design

As decided in the analysis, an automated neural network based AI would be an optimal solution to the requirements, due to its low maintenance cost and versatility. However, the complexity of the neural network based solution comes in the design stage. This is because a neural network is hard to debug, as it is essentially a stochastic process of updating many weights, based on feedback from an environment. As a result, having a well-designed network with a strong fundamental understanding of its architecture will make it easier to make in the long run.

The first prototype of the neural network will have a simple architecture as shown in the analysis. It will have an input layer, a single hidden layer with the same number of nodes and the input layer, and an output layer. It is better to begin simply and prototype the performance and add different features as it needs it.

For the initial prototype for the blackjack prototype, I will simplify it even further by reducing the number of features down to four features: the current value of the AI’s hand, the value of the dealer’s hand and the number of cards in the AI’s and the dealer’s hand. In addition, it will have one hidden layer of the same size as the input layer, and an output layer of two binary outputs: hit and stand.

The network architecture for this prototype is shown below:



In Conjunction with this, I will train this network with the following rewards and the following exploration strategy.

**Rewards:**

* Hit -> Hand value of agent \* normalisation constant
* Stand -> If the AI is leading after a stand, then Hand value of agent \* normalisation constant
* Bust -> (-hand value of agent – 1) \* normalisation constant
* Absolute Winner -> (hand value of agent + 1) \* normalisation constant

The normalisation of these values assists in the optimisation of the gradient descent algorithms, because all of the values for the features will be in a similar range, which means that gradient descent will occur at the same rate for all weights. In this example, I have used 1/30 for the normalisation constant, because no hand will typically exceed a hand value of 30.

These decisions for the rewards is not balanced in the sense that there is often going to be a higher cost for going bust or losing, compared to the reward for winning. This is because the rewards are relative to the hand values, as a result, when the agent goes bust the cost will always be lower than -21, whereas the reward for winning will only be around 21 max. This is somewhat countered by the fact that the agent receives a reward for hitting, and increasing the value of the hand, however this could be optimised. Although this is a good starting point for a prototype.

**Exploration Strategy**

As an initial exploration strategy, I have used the e-greedy policy. During the training of the neural network, there are different behaviours the AI can take when exploring the problem space. In the parameterisation of the neural network, there is a designated number of games for which is dedicated for exploring, after which the neural network will behave in a more predictable manner and act on what the weighting shows is the optimal move, for the highest reward.

The e-greedy policy has a parameter epsilon, which is a probability that the agent will take a random action. Whilst the agent is still exploring the environment, a random action is always taken, so that the agent can explore the environment and use an optimisation algorithm to change the weightings based on the features and rewards. After this exploration phase of the training, the value for epsilon will start high, and with each action the agent takes, epsilon will be decremented to a final parameterised value – the behaviour of the agent will change based on what final value of epsilon is passed.

The final value of epsilon is above 0, so that the agent will not always act in a greedy way, as there is no guarantee the problem space has been explored fully during the exploration phase. This strategy is good because it combines the random and greedy exploration strategies, and it is a fast implementation for a prototyping phase.

##### Prototype Results:



This initial prototype has a high winrate of 45.6%, which is higher than expected[[11]](#footnote-11) which means on the surface this initial prototype performs well, in a pure sense. However, keep in mind that this prototype only tests against one other player – the dealer – who behaves in a predictable way, which could be exploited by a more sophisticated agent, this winrate may drop if the agent were 1v1 against a different opponent, or if used in group play.

The big issue here is that the average value for the agent to stand is 14.6, which is lower than desired, considering the dealer will only stand at a value above 17. Since the agent still has a relatively high winrate, perhaps 17 is not the optimal value to stand on, however, the agent’s winrate is lower than 50% which suggests to me that 14 or 15 may not be the optimal stand value either. However, this could also be down to the rewards being skewed towards a higher loss and bust cost, than the typical win reward.

All this and more neural network components will be addressed in the second prototype.

##### Prototype 2 Architecture and Design

One aspect which can be improved in this second prototype is the architecture. Firstly, there were arguably only two important features in the first prototype: the hand values for each of the players. The length of the hand for each player is not relevant, and does not have much relation to the win condition for any of the players. There is some correlation between the length of the hand and the likelihood of going bust from hitting next, however, there is a more direct relationship with this same result and the feature of the value of the hand for each player. In addition to this, the architecture will only work for a 1v1 scenario.

As a result, I have changed the architecture in the second prototype to take in the hand values for each of the players, in addition to the chances generated from the Card Counter, which is also used by the card counting agent.

This changes the network architecture to look like this:



###### Recurrent Cells

One of the drawbacks of a normal neural network is that it cannot make temporal based decisions. For example, say the neural network knew that an opponent is nearly more likely to stand on the 3rd turn then, the agent may want to take this into account when deciding whether it is better to stand or hit. In essence, the use of a recurrent layer allows the agent to make better decisions within a given game, rather than each move being considered in isolation. Although, the impact of a recurrent neural network may not be as large for a game like blackjack, where each move could be considered in isolation, in theory, and the effective reward would be similar – for example, it does not matter if it is the first turn or the third, it would still be a bad move to hit if you have a hand value of 21. However, in a game like poker, where the previous bets of each person, and the progress of each game matters more, the recurrent neural network design is more effective. Therefore, by implementing this now, it makes for an easier conversion later on.

The recurrent node operates by feeding the output of previous calculations back into the hidden layers of the neural network, essentially providing more than one input layer. An alternative option to this would have been to stack the sampling frames and then fed all the frames into the network at once during training, however, whilst that is applicable for the simple nature of blackjack, using a recurrent cell makes it easier to extend the neural network for other games.



###### Experience Sampling

For a similar purpose to the recurrent cell, rather than feeding every game to the network after a enough games has filled a batch size in a linear manner, by storing the experiences and sampling experiences from them randomly, the network can learn more robustly. Each experience will be stored in a simple array structure of [state, action, reward, new state]. The random sampling of experience allows the network to learn from a wider variety of experiences, preventing overfitting and allowing the network to learn and generalise faster, rather than just learning from the immediate past.

I will implement a simple class which will handle the experience sampling:

###### Dropout Layers

One issue which may occur with any sort of learning algorithm is overfitting – this is where a learning algorithm adjusts to weights to be able to perform well on a given dataset, but then lose its ability to perform in the same way generally, because the network’s weights have been adapted to fit the nuances of the given dataset too much. An example of this is when a neural network is being trained in image classification, and it has a 95% accuracy rate for the training set, however, for a test set it may have only 60% accuracy.

Dropout layers are layers within a neural network, in which the layer has a chance to be deactivated, and then the sum is scaled up and carried through the network, so that there is a convergent sum at the end. These improve generalisation, hence reduce the risk of overfitting because the dependency between neurons, in a given layer, is reduced with dropout which, in turn, increases the potency of each individual node. One of the drawbacks of dropout is that it increases the number of iterations required for convergence, but reduces the training time.

For this project, this is not such a large issue for the initial training of the neural network, because the training dataset can be increased to a large amount with a small time cost, because the game is built in and it is quite quick to complete a game. Since so many games will be played, and they will be randomly sampled for training, the risk of overfitting is not incredibly high, as any anomalies can get smoothed out with more data. However, one aspect of the objectives of this project is the ability for the AI to adapt with the behaviours of different user’s. Whilst this could include the feature of classification of users based on their playstyle, and then changing the playstyle of AI to what is effective against these playstyles, one other option could be to use the dataset of a player and then train the AI against this dataset. In this scenario the given dataset will be limited, although there is also a possibility of grouping similar users together in the same dataset) and it will be much smaller than the dataset which can be generated using the other agents and the dealer, consequently, the chance of overfitting is much higher. Therefore, dropout layers will be an important feature.



###### Separate Target Network and Primary Network

###### Convolutional Layers

###### Advantage and Value Streams

###### Exploration of Exploration Strategies

#### Algorithms – Training Algorithm

#### Algorithms – Using the Prediction Network to Detect Playstyles

#### Algorithms – Changing the Training Rewards for Different Playstyles

#### Algorithms – Adjusting Parameters for Different Behaviours and Play Performance

#### Algorithms – Utilising Different Exploration Strategies for Different Playstyles and Performance

### Design – Simple AI

### Design – Comparison Tool

### 

# SoftwareDevelopment

# Testing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test No.** | **Objective Being Tested** | **Test Description** | **Test Type** | **Test Data** | **Expected Output** |
| 1 | 1a | Have user play a move, and time wait duration for AI to generate a move | Normal |  | Less than 30 seconds |
| 2 |  |  | Erroneous |  | Less than 30 seconds |
| 3 |  |  | Extreme |  | Less than 30 seconds |
| 4 | 1b | Have a new user play against the AI, and have the AI maintain a similar winrate as compared to the dealer |  |  |  |
| 5 | 1c | Have a user predict the agent’s actions for 10 random times in a game. |  |  |  |
| 6 |  | Have a user predict what the agent will do at a hand value of 17 or above, in 10 different games. |  |  |  |
| 7 | 1d |  |  |  |  |
| 8 |  |  |  |  |  |
| 9 |  |  |  |  |  |
| 10 |  |  |  |  |  |
| 11 |  |  |  |  |  |
| 12 |  |  |  |  |  |
| 13 |  |  |  |  |  |
| 14 |  |  |  |  |  |
| 15 |  |  |  |  |  |
| 16 |  |  |  |  |  |
| 17 |  |  |  |  |  |
| 18 |  |  |  |  |  |
| 19 |  |  |  |  |  |
| 20 |  |  |  |  |  |
| 21 |  |  |  |  |  |
| 22 |  |  |  |  |  |
| 23 |  |  |  |  |  |

# Evaluation

1. <https://en.wikipedia.org/wiki/Fundamental_theorem_of_poker> - If any player plays differently compared to how they would play if they could see their opponents cards, those players lose. Any time an opponent plays differently compared to how they would play if they could see your cards, you gain. (Summarised) [↑](#footnote-ref-1)
2. **Sandbagging/Trapping** is purposely betting weaker than your hand would suggest you should, in order to deceive the opponent into thinking that you have a weaker hand than you actually do, causing them to break the fundamental theory of poker. [↑](#footnote-ref-2)
3. <https://archive.ics.uci.edu/ml/datasets/Poker+Hand> (Repository of 5 Card Draw Hands) [↑](#footnote-ref-3)
4. <https://en.wikipedia.org/wiki/Monte_Carlo_method> “Monte Carlo methods (or Monte Carlo experiments) are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. Their essential idea is using randomness to solve problems that might be deterministic in principle.” [↑](#footnote-ref-4)
5. Metagame – Strategy which transcends a prescribed ruleset. An optimal strategy for the game. [↑](#footnote-ref-5)
6. <http://www2.cs.uregina.ca/~hilder/refereed_conference_proceedings/cig09.pdf> (No-Limit Texas Hold’em Poker Agents Created with Evolutionary Neural Networks – Garrett Nicolai and Robert J. Hilderman). [↑](#footnote-ref-6)
7. <http://www.numpy.org/> (Scientific Computing Library for Python) [↑](#footnote-ref-7)
8. <https://www.tensorflow.org/> (High level library for building graphs in Python, for machine learning) [↑](#footnote-ref-8)
9. [www.pokerology.com/lessons/poker-playing-styles/](http://www.pokerology.com/lessons/poker-playing-styles/) (Article on poker playing styles) [↑](#footnote-ref-9)
10. <https://wizardofodds.com/games/blackjack/appendix/4/> (Summarised Net Win in Blackjack Table – probability of net win is 42.42%) [↑](#footnote-ref-10)
11. <https://wizardofodds.com/games/blackjack/appendix/4/> (Summarised Net Win in Blackjack Table – probability of net win is 42.42%) [↑](#footnote-ref-11)