An evaluation of how Dynamic Programming and Game Theory are applied to Liar's Dice

Submitted in partial fulfilment

of the requirements of the degree

Bachelor of Science Honours in Computer Science

at Rhodes University

Trevor Johnson

Supervised by:

Prof. Shaun Bangay

Philip Sterne

November 2007

Abstract

Liar’s Dice is a simple dice game for humans to play, but for a computer, there are a large number of possible options for each decision. This project looks at how Dynamic Programming and Game Theory are used to produce two agents that can play Liar’s Dice. The two agents are produced using different data structures, the first being produced using a tree, while the other is implemented using a matrix. Once these data structures are created, each implement a different strategy. The tree utilises a pure strategy to make its decisions, while the matrix adopts a mixed strategy to make the decisions at the various times in a game of Liar’s Dice. Both these agents are then compared against each other, and finally against a real human player to determine which agent is more successful at playing Liar’s Dice. From these results it is determined that the matrix agent adopting a mixed strategy is more successful at playing Liar’s Dice against a human opponent.

Acknowledgments

I would like to thank my supervisors Prof. Shaun Bangay and Philip Sterne for always being available to discuss ideas and give advice and motivation during this project. I would also like to thank the Department of Computer Science at Rhodes University and all the sponsors, for giving me the opportunity to carry out this project.

Contents

List of Figures

Chapter 1

1. Introduction

The purpose of this chapter is to give a brief introduction to firstly the problem statement for this project as well as an overview of the Liar’s Dice game itself and the various techniques that were considered to produce an optimal solution.

1.1. Problem Statement

Liar’s Dice is a relatively simple game to play for humans, but for a computer, the game is far more complicated. The aim of this project is to produce an agent that can make the optimal decisions in a game of two player Liar’s Dice. A method of optimisation to produce data structures for the decisions is required as there are a large number of moves available to a player at each decision. Once these data structures for the decisions are set up, the best or optimal decision will then need to be decided on. Both of the data structure and optimal decision will need to be produced to solve the problem at hand.

1.2. Liar’s Dice

Liar’s Dice is a popular dice game, played with five poker dice played by two or more players. For this project I will only be considering the two player one dice version of the game

The two player one dice version of Liar’s Dice is played as follows; Player one rolls the dice, and calls a value as to what was rolled. This call could be the truth, i.e. what was actually rolled, or it could be a lie. Player two must then decide whether to accept the call or challenge this call. If the call is accepted, Player two must roll the die to try better this call, regardless of what value was actually rolled. Player two must then make a call about what was rolled. This continues until one of the players decides to challenge a call. When this occurs, the player that rolled the dice and made the call, must reveal what was actually rolled. If the ‘calling’ player called a hand other than the one that was rolled, then that player loses, but if the ‘calling’ player has indeed told the truth, then the player that made the challenge loses.

1.3. Dynamic Programming

“Dynamic programming is an approach to optimization” [Nemhauser, 1966]. It is similar to reinforcement learning, except that instead of modelling an agent interacting with an environment, we now try to find the optimal action by performing all possible actions in all possible states. Performing one of these actions will result in a sub-game, whose solution might be optimal or not. If it is not optimal, then another action is performed, until an optimal solution is achieved. More simply put, Dynamic Programming is an exhaustive search of all the possible actions in a problem, and so the best actions can be determined. This technique has been used successfully to solve other games, for example Yahtzee [Woodward, 2003].

1.4. Game Theory

Game Theory is the study of “decision making under conditions of uncertainty and interdependence. Components of a game include players, strategies, actions, payoffs, outcomes and an equilibrium” [Pearson Education, 2005]. Liar’s Dice is a game of chance and also has an element of deception. In order for the agent to be a successful Liar’s Dice player, it will have to be able to deceive its opponent, and be able to detect when its opponent is trying to deceiving it. As there is a need for the agent to bluff, it is also required that certain aspects of Game Theory be included to make decisions about which action to take.

There are two ways in which games are represented in Game Theory. The first is the normal for of the game, where a matrix holds the payoffs each player can hope to gain from adopting certain strategies, and the second is the extensive form, where the game is represented as a tree of all the possible actions and decisions for both players in the game.

1.5. Document Structure

The remainder of this write up is arranged as follows:

Chapter 2 gives an over view of methods used to solve similar problems to Liar’s Dice.

Chapter 3 discusses the design of the agent. This includes what the agent would need to be capable of and how this would be achieved.

Chapter 4 deals with the different approaches considered in order to produce an agent that meets the design specifications.

Chapter 5 addresses the results that are obtained from the various approaches that were used.

Chapter 6 concludes the thesis, and gives future extensions to the project.

Chapter 2

2. Related Work

There are various methods that can be used to determine an optimal solution for a game or problem. In this chapter I will be looking at how Dynamic Programming, specifically the Markov Decision process and Markov Chains, Game Theory and other techniques can be used to produce the optimal solution or strategy for various games and problems, and so be applied to Liar's Dice.

2.1. Programming techniques

Problems similar to Liar's Dice have been solved using different programming techniques. This section describes each of these techniques and considers whether or not they are applicable to finding an optimal solution to Liar's Dice.

2.1.1. Dynamic Programming

Dynamic Programming is a method of optimisation that has been applied to many problems, for example Yahtzee by Cremers C.J.F [2002] and pig by Neller T.W and Presser C.M.G [2004], which has produced solutions which are very good. Various aspects of Dynamic Programming have been used, including Value Iteration, Markov Chains and the Markov Decision Process. Dynamic Programming finds the solution to a large problem by breaking the problem down into smaller sub-problems which when solved will give the solution to the original problem.

When a problem has the Markov property, it means that decisions to the next state in the game is dependent on the current state of the game and does not depend on previous states of the game. For example, in a game of chess, at any stage of the game, you don't have to know all the previous moves to decide on your next move, you just need to know the current state. A Markov Decision Process is just like a

Markov Chain, except the transition matrix depends on the action taken by the agent at each time step. In Liar’s Dice the current decision (what to call for a given roll, whether to accept or challenge a call, or which dice to roll) depends on the current state only, and any previous information required can be encoded into the current state. Thus Liar’s Dice has the Markov property, and the transition matrix depends on the action taken by the agent at each step and so can modelled as a Markov Chain. Markov Chains can be produce a solution for a problem when solved using Dynamic Programming, as each state leads onto another state, and through solving each state from the end of the game until the beginning, a solution for the problem can be achieved.

2.1.1.1. Markov Chains

Blatt S [2000] uses Markov Chains in order to obtain a state-space model of the board game Risk. She describes how the state-space model of the whole game is a sequence of probability vectors that describe the state of the system at any time, and so the state-space model must be developed for a single battle. Following this, she shows how each of these battles has a transition probability (the probability of moving into the next state) can be found using the Markov property i.e. that this transition probability only depends on the current state.

The transition matrix is set up by calculating the probabilities of moving from one transient state to another transient state, as well as the probabilities of moving from a transient state to an absorbing state, where an absorbing state is a state where one of the players loses, and a transient state is a state where the game can continue.

Once this matrix is set up, she then moves on to show how the actual probabilities are determined. These calculations are specific to the game of Risk, and so would not bear much relevance to the Liar’s Dice game.

There are certain aspects of this paper that are relevant to finding the optimal solution for Liar’s Dice, such as the need for transition matrices, but there are fundamental differences in the type of game being solved in the paper and Liar’s Dice. Both games are games of chance, as the outcome depends on the roll of a die, but Liar’s Dice also has an element of bluffing, and so other sources and ideas would also be required to give an insight to incorporate bluffing. One might assume that a pure strategy could be adopted for Risk based on the probabilities of moving into a certain state. This could also be the case in Liar’s Dice.

2.1.1.2. Markov Decision Process

Dr. Verhoeff T [2000] developed an optimal strategy for the popular dice game Yahtzee. Cremers C.J.F [2002] uses the Markov Decision Process and previous work by Dr. Verhoeff T [2000] to find an alternative optimal solution that has an optimal probability of achieving a given score or more. He states that “The key ingredients in a Markov Decision Process are the states, events leading from one state to the next state, capturing the transition function, probability distribution over the events in every state and score functions for the events in every state” [Cremers C.J.F, 2002]. Liar’s Dice has these properties, and so can be modelled as Markov Decision Process.

After this he begins explaining how events can be lifted to produce games, but the discussion does not actually explain how the Markov Decision Process is applied to the game to solve it. The section dealing with the Markov Decision Process was very brief as it was not the focus of his paper. This paper is relevant in that it shows that the Markov Decision Process can be used to find the optimal solution for Yahtzee, and so could also be the case for Liar’s Dice, but it does not address how this is done.

2.1.2. Value Iteration

Value iteration is described as “an algorithm that iteratively improves estimates of the value of being in each state” by Neller T.W and Presser C.M.G [2004]. In this paper authored by Neller T.W and Presser C.M.G [2004], their aim is to use Value Iteration to solve another relatively simple dice game called pig. The paper begins by stating that the game could be played using a simple strategy (a pure strategy) in which there is no decisions process, and the same strategy is used for every round of the game. This strategy maximises the score per round, but not the probability of winning. While this strategy may work some of the time, there are cases where it will not work. A similar simple strategy could be used in Liar's Dice, but this would make the agent predictable, which is undesirable, and so another method will have to be implemented.

Firstly, Neller T.W and Presser C.M.G [2004] use Value Iteration to solve a simpler version of pig called piglet, and then apply it to the more complicated game. In order to apply Value Iteration to a problem, they state that the world must consist of states, actions and rewards, and that the goal is to compute which action to take in each state so as to maximize future rewards. At any time, the state of the game is known, and there are only a certain number of actions that can be taken. It is also known that for a certain action, there is a probability (possibly zero) that the state will result in another state. For each of these transitions there is an immediate reward available, as well as a future reward, but only to some extent. Neller T.W and Presser C.M.G [2004] then describe how this is applied to piglet, and show that this gives results that converge to a constant value very quickly. These results obtained by Neller T.W and Presser C.M.G [2004] are shown in Figure 1.

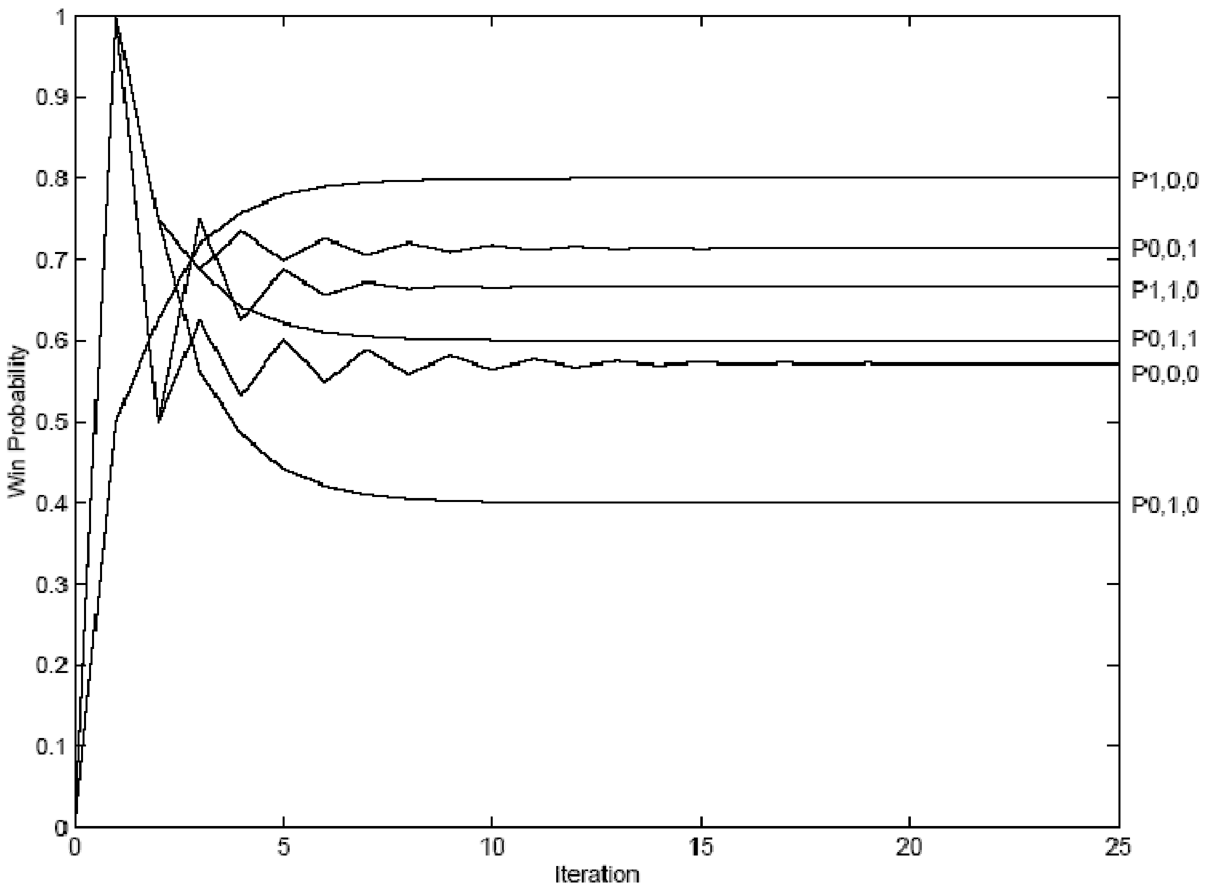


Figure 1: Value Iteration on Piglet with a goal of two points

This same method is then used for Pig, giving a very good result which starts off following the simple "hold at twenty strategy", but then changes to a strategy that optimises the probability of winning when a player reaches a higher score. This is because there is a much high probability of winning at a high score than at a lower one. This same strategy of maximising the probability of winning and not the score could also be used in Liar's Dice.

In the game pig, it is possible for states to be repeated, and so pig is not a candidate to be solved using the Markov Decision Process. This is why Value Iteration was implemented instead.

2.2. Game Theory to determine strategy

Once the data structures of the available strategies have been set up correctly using a suitable programming technique, then the correct strategy needs to be selected. Game

Theory, as mentioned previously, can be used to make optimal decisions under uncertainty. This would be ideal to solve the problem presented by this project. The following section looks into how various implementations of Game Theory are used to find optimal decisions.

2.2.1. Nash Equilibrium

“A Nash equilibrium, named after John Nash, is a set of strategies, one for each player, such that no player has incentive to unilaterally change her action. Players are in equilibrium if a change in strategies by any one of them would lead that player to earn less than if she remained with her current strategy. For games in which players randomize (mixed strategies), the expected or average payoff must be at least as large as that obtainable by any other strategy” [Shor M, 2005]. With this in mind, it would appear that calculating the Nash Equilibrium for the game would give the best strategy to play, and so should be used in the production of the agent.

In Liar’s Dice, the player has the option to lie about what you have rolled. This lie is essentially a bluff. Bertomeu J [2004] uses poker to show that equilibrium strategies must involve bluffs, and he defines a bluff as strategies that “make use of low draws in order to negate information acquisition about potential high hands”. In games like poker, following a simple strategy proves ineffective as they never yield a Nash Equilibrium, and so becomes predictable, for this reason this strategy is ruled out for Liar’s Dice. Bertomeu J [2004] concludes his paper by saying that bluffing is essential to the solution of imperfect information zero-sum games like Liar’s Dice, and so this aspect must be included into the Liar’s Dice agent in order for it to be optimal.

Gilpin, A and Sandholm T [2006] show how to find a Nash equilibrium in Rhode Island Hold’em poker, where bluffing is essential. The paper states that for sequential games with imperfect information, such as Liar’s Dice, taking a normal form of the game, where each strategy is a pure strategy, and converting it to the extensive form, results in an exponential increase in the size of the tree. This should be born in mind when it comes to designing the data structures for the agent.

This confirmed in another paper authored by Miltersen P.B and Sørensen T.B [2005], where they state that “One may analyze an extensive form game by converting it into normal form and then analyzing the resulting matrix game. However, the conversion from extensive to normal form incurs an exponential blow up in the size of the representation.” [Miltersen P.B, Sørensen T.B, 2005]. This would suggest that it is easier to begin with a normal (matrix) form of the game instead of an extensive (tree) form. As well as this, the paper states that the Nash Equilibria of matrix games (i.e., two-player zero-sum games in normal form) coincide with pairs of minimax and maximin mixed strategies.

Reiley D.H et al [2005] wrote a paper which shows how games with imperfect information, such as Liar’s Dice cannot be played with a pure strategy. The paper discusses how to calculate the mixed strategy Nash Equilibrium for a simplified version of poker. A mixed strategy Nash Equilibrium would be very useful in the Liar’s Dice context to decide when to bluff or when to tell the truth for a given roll. This is the most likely method to be used to decide what strategies should be used in the Liar’s Dice agent.

These sources show that there is a need to calculate the Nash Equilibrium for Liar’s Dice. Conitzer V and Sandholm T [2006] describe how to solve the pure and mixed strategies form the normal form of a game.

The commitment to a pure strategy used a recursive method to find the strategy that returned the highest payoff. The method of committing to a mixed strategy uses the normal form of the game which tries to minimise the maximum expected payoff that the opponent can gain, i.e. the use of a minimax strategy. In the case of Liar’s Dice, a mixed strategy would be required.

2.2.2. Other methods

Hoehn B et al. [2005] take another approach to produce an agent that can play Kuhn Poker (a simple two player, three card version of poker). They try to exploit weaknesses in the way humans play games, by observing their play, and learning from this. It should be noted that the agent uses a Nash strategies until the agent has been trained. The paper is concluded by stating that their solution is generally not feasible in a small amount of hands, and so would need a long training time.

Billings D et al., [2002] produced a system called Poki that uses methods for hand evaluation and betting strategy, but also uses learning techniques to construct statistical models of each opponent, and dynamically adapts to exploit observed patterns and tendencies. They make use of an expert system to make decisions when there is relatively little information to make the decision. They claim that the system works well, but is not of the same calibre as a World Class human poker player.

2.3. Liar's Dice Tactics

Literature on Liar's Dice is sparse but a paper by Freeman G.H [1989], the only paper on Liar's Dice found, gives a good insight into various tactics that can be applied when playing Liar's Dice. The paper says "The skill of liars dice consists of knowing when to tell the truth and when to lie –and also when to challenge-and this only comes with experience." [Freeman G.H 1989]. The paper mainly deals with making the decision about which dice to roll after having accepted the truth as well as briefly discussing what tactics should be used when accepting a lie. These tactics could be implemented as an expert system to decide which dice to roll in given situations, but the author states that the tactics discussed are of a fairly simple kind. The author does mention that he has made an attempt to look at strategies available for a simpler version of Liar's Dice, but that it would take a great deal of analysis.

2.4. Summary

There are a number of different ways in which Dynamic Programming and Game Theory can be applied to find solutions to various games. Here are the positive and negative aspects of the methods discussed in this chapter.

The Markov Decision Process and Markov Chains have both been used to produce an optimal solution for various games, and so would be a good choice as a programming technique to produce data structures for the Liar's Dice agent.

Value Iteration, has also produced an optimal agent for the game pig, it is best applied in games where there are states that repeat, but this is not the case in Liar's Dice.

Most of the literature that deals with having to find the optimal strategy for various games calculate the Mixed Nash Equilibrium and use this as the strategy profile. Others have also used a learning agent that tries to exploit weaknesses in the opposition's play, but haven't been as successful as just using the Nash Equilibrium.

The material specifically written on Liar's Dice deals with the tactics one could follow when having to decide which dice to roll, but none of the other decisions. As this project is only dealing with a one dice version of Liar's dice, there is no need to utilise any form of tactics or strategies to decide which dice to roll, as the agent in this project does not need to make this decision.

Chapter 3

3. Design

In this chapter I discuss the various aspects of Liar’s Dice that need to be addressed to produce an agent that can play Liar’s Dice. Following this I look at the different data structures that can be used to model the game of Liar’s Dice, and then how the methods dealt with in the previous chapter can be incorporated with these data structures were used to produce an agent that can play Liar’s Dice.

3.1. The decisions that need to be made

The first decision that needs to be considered is what should be called after having rolled the dice. This is the decision where the agent has the option to bluff i.e. lie about what was rolled, or to tell the truth. This decision is affected by what the previous call of the opponent was as well as what was actually rolled by the agent. If the agent rolled a hand lower than what was actually called, then the agent has no option but to lie. The agent must then decide what the lie should be. All these aspects of the decision need to be considered before it has been made.

The next decision that is required for the agent is the decision to accept or challenge a call. This is the decision where the agent must be able to detect when the opponent is lying or not. A wrong decision here would either lead to the agent losing the game if a challenge is made when the opponent was not lying, or it could lead into a state where the there is a lower chance of the agent winning i.e. if the agent accepts a lie. Both these situations need to be avoided in order to produce an agent that is optimal.

3.2. The data structures

As mentioned briefly in the previous chapter, there are two different ways to represent a game. The first being the normal form of a game. In this form, a game is represented as a matrix of payoff values for both players. These payoffs represent the amount of utility that each player is expected to gain after having played a particular strategy. Liar’s Dice is a zero-sum game, because at the end of the game one player wins, the other player loses, and so the total payoffs to both players must add up to zero. Once the payoffs for all the strategy profiles have been calculated, they can be used to determine which strategy is the Nash Equilibrium.

The second way of representing a game is the extensive form. This form of representing a game uses a tree where each node of the tree represents a decision made by a player in the game, and the edges represent the actions made by players. When the tree terminates i.e. a player chooses an action that leads to the end of the game, the players are both assigned payoffs, which are calculated while traversing the tree. This form shows explicitly the interactions of both players until the end of the game. It should be noted that calculating the normal form of the game from the extensive form results in an exponential blow-up in size, and so for large games would hinder performance.

As both matrices and trees can be implemented as data structures to determine which decisions the agent should make at various points in the game, both were investigated and compared against each other to determine which is more successful.

3.3. How the decisions are made

This section will now look at how the programming techniques investigated in the previous chapter are used to set up the data structures which in turn are used to make the decisions.

3.3.1. Using trees

Instead of using one very large tree to represent the game, the game was broken up into smaller trees representing smaller sub-games, which were solved and whose solutions were then used to solve the sub-sequent sub-games. Through solving each of these sub-games, a solution for the whole game can be achieved. Breaking the game into smaller sub-games reduces the size and complexity of the game, and made it possible to represent the game as a Markov Chain. This is how the game can be solved using Dynamic Programming.

The process of setting up the trees starts with creating a tree for the final sub-game in the game, namely the “accept five” sub-game (the game played after having accepted a call of five); as this is the final sub-game where a decision is required to be made. There is another sub-game after the “accept five” sub-game, but the probability of winning this sub-game is always zero, as the only way to get to this sub-game is by accepting a six which can never be beaten. Figure 2 shows the structure of the “accept five” sub-game tree.

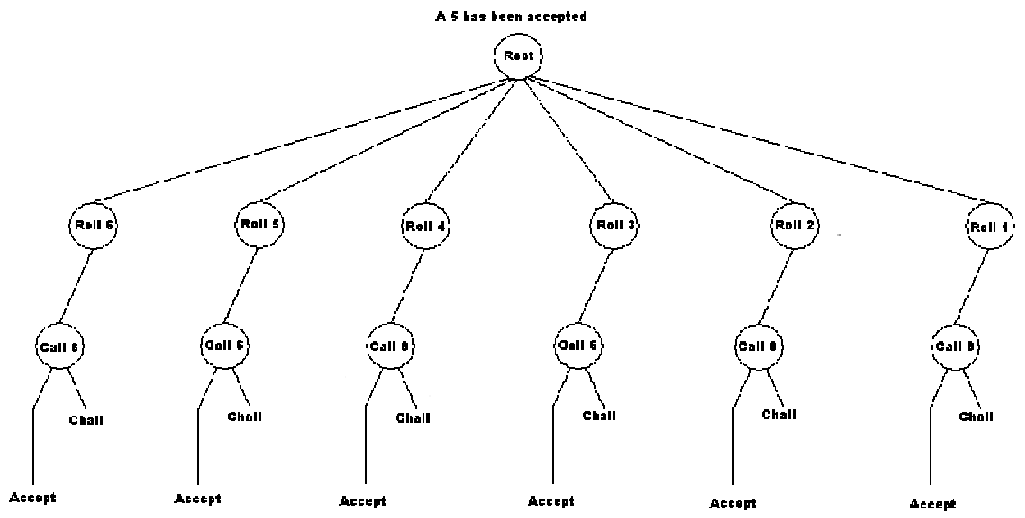


Figure 2 The "accept five" sub-game tree

Before any of the probabilities can be calculated, the trees are set up with the correct number of nodes for all the possible calls. The number of calls is dependent on the value that has been accepted in the previous sub-game. For example, in the diagram, there is no option to call a four, as a five has been accepted, and making a call lower than or equal to the one that has just been accepted would result in a loss. It is also dependent on the value that has been rolled, as making a call that is lower than what was actually rolled reduces the chance of winning and so would not be an optimal decision. Once the tree is set up with the correct number of nodes, the probabilities can then be calculated from the leaves (where calls are either accepted or challenged) and then propagated up the tree until it has been populated with the various probabilities required to make the various decisions. How these probabilities are calculated will be discussed later in this chapter under section 3.3.1.1.

The root node of each sub-game tree holds the probability that the player rolling the die will win before having rolled the die and after having accepted a certain call, for example, the probability in the root node of the “accept five” sub-game holds the probability of winning after having accepted a five, and before having rolled the die. Below the root node there are the nodes for the possible rolls and there is one node for each possible roll. These hold the probability of winning after having rolled the die for each given roll in that sub-game. Below the roll nodes are the possible call nodes. Each of these nodes holds the probability that the player that rolled the die would win if he/she decided on that particular call, after having rolled a certain value in that sub-game. The leaves of the tree are where other player has the choice to accept this call or challenge it. These nodes hold the probability that the caller would win if the call is either accepted or challenged.

The probability of winning for a given call, following a given roll, is calculated for all the possible rolls and calls in each sub-game. The call with the highest probability of winning in each roll is the call that the agent will decide to use in that sub-game, for a given roll. As the call is dependent on the value that is rolled, the value of the call is only known after the die has been rolled.

The decision to accept or challenge a call from the opponent is made based on the probability that the agent would win if the call of the opponent is accepted, taking into account the probability that the agent will roll a higher value than the call of the opponent, and the probability that the agent would win if the call of the opponent is challenged, taking into account the probability that the opponent actually rolled what was called. If it is more probable that the opponent lied, then the agent would decide to challenge the call. If on the other hand it is more probable that the agent would win if the call is accepted, then the call is accepted.

As all the instances of all the possible actions that can be played by the players have been played out, this method will give the agent a decision for every state in the game.

The only problem with this method is that the tree only gives the agent one decision for every sub-game, and so would be come predictable.

3.3.1.1. Calculating the probabilities

It should also be noted that all the probabilities calculated in the trees are calculated for the player that has to roll the dice and then make a call (Player 1). If the probabilities for the player accepting or challenging the call (Player 2) are needed, the probabilities for Player 1 are subtracted from one, giving the probabilities for Player 2.

3.3.1.1.1. For the call decision

As mentioned above, the trees are first set up to the correct specifications before being populated with the probabilities that are required to make the decisions, and that the first one to be set up was the “accept five” sub-game tree. In this sub-game, any decision made by Player 2 to either accept or challenge a call results in the end of the game. Figure 3 shows the “accept five” sub-game tree with the probabilities of winning for the various accept or challenge choices, showing that the game comes to an end in all the cases.

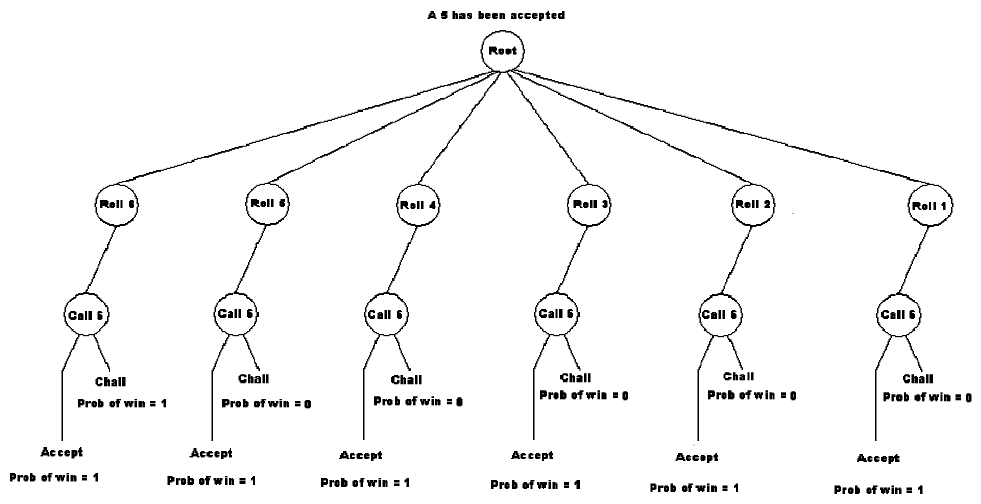


Figure 3 The "accept five" sub-game tree with the probabilities of accepting or challenging a call

Once these probabilities are know, the probability of winning for each call can be calculated using the following equation:

$$P(CallerWin|Call=x, Roll=y)=P(CallerWin|A/C=C, Call=x, Roll=y).P(C)+ (CallerWin|A/C=A).P(A) \quad (1)$$

In this equation P*(CallerWin |Call = x)* is the probability of Player 1 winning with a call of x, *P(CallerWin |A/C = C, Call = x, Roll = y)* is the probability of Player 1 winning if Player 2 challenges a call of x with a roll of y, *P(C)* is the probability that Player 2 challenges the call, *P(CallerWin |A/C = A)* is the probability of Player 1 winning if Player 2 accepts the call and finally *P(A)* is the probability that Player 2 accepts the call.

The probability of winning if Player 2 challenges a call is always either one or zero, depending on whether Player 1 lied or not. It is one if Player 1 told the truth, and it is zero if Player 1 lied. These probabilities are found in the “challenge” leaf nodes of the tree.

In order to calculate the probability that Player 2 will challenge the call, first the number of ways that Player 2 can challenge and win for that call over the total number of ways the call can be challenged is calculated. This is the probability that Player 2 will win if the call is challenged. This is divided by the probability that Player 2 will win if the call is challenged plus the probability that Player 2 will win if the call is accepted. The following equation shows how this can be calculated.

$$P(C) = \frac{(Num(A/C = C, P(A/Cwin) = 1) / Num(A/C = C))}{(Num(A/C = C, P(A/Cwin) = 1) / Num(A/C = C)) + (P(A/Cwin \mid A/C = A))} \quad (2)$$

The probability of winning for Player 1 if Player 2 accepts the call is one minus the value that is in the root node of the sub-game tree of the accepted call. This is because the value in the root node is the probability of winning for Player 2 in the next sub-game, and so subtracting this probability from one gives the probability of winning for Player 1 of the previous sub-game.

Lastly the probability that Player 2 will accept the call is the probability that Player 2 would win if the call is accepted. Figure 4 shows the "accept five" sub-game tree with the probabilities of winning for the different calls.

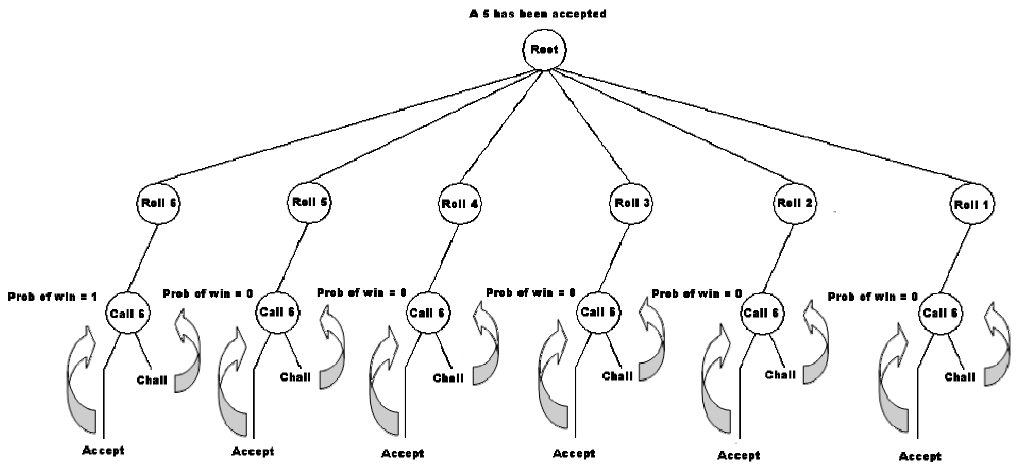


Figure 4 The "accept five" sub-game tree with the probabilities of winning for the various calls

These probabilities are calculated using equation (1) in the following way:

For the rolls where the call is higher than the value that was rolled, the probability of winning if the call is challenged is zero, therefore:

$$P(CallerWin\ |A/C = C,\ Call = x,\ Roll = y).P(C) = 0$$

This leaves

$$P(CallerWin\ |Call = x) = P(CallerWin\ |A/C = A).P(A)$$

But the probability of the opponent accepting a call of six is zero, therefore:

$$P(CallerWin\ |A/C = A).P(A) = 0 \quad \text{therefore}$$

$$P(CallerWin\ |Call = x) = 0$$

In the case where the roll is equal to the call, the probability of the opponent accepting a call of six is also zero, but the probability of winning if the call is challenged is now one, therefore

$$P(CallerWin\ |Call = Roll = x) = P(C)$$

$$P(C) = \frac{(Num(A/C = C, P(OppWin) = 1)/NumCalls)}{(Num(A/C = C, P(OppWin) = 1)/NumCalls) + (P(OppWin\ |\ A/C = A))}$$

$$\text{which is } \frac{5/6}{5/6+0} = 1 \text{ therefore}$$

$$P(CallerWin\ |Call = Roll = x) = 1$$

for the case where the call is equal to the roll.

Once the probability of winning for a particular call is known for all the rolls, the probability of winning after rolling the dice is known. This is simply the highest probability in the call nodes below each roll. The reason for this is that the call node with the highest probability of winning is the call the agent will decide on for each roll. Figure 5 shows the “accept five” sub-game tree with the probability of winning for the various rolls.

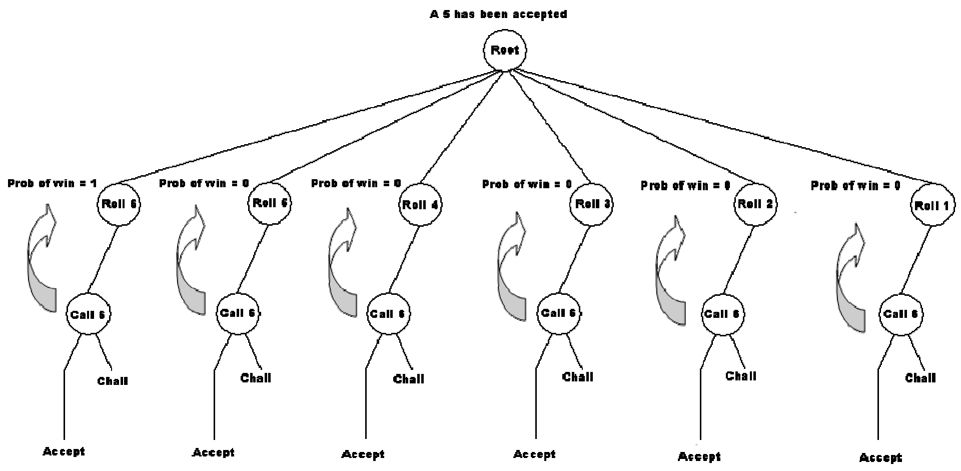


Figure 5 The "accept five" sub-game tree with the probabilities of winning for the various rolls

Lastly the probability of winning before rolling the dice is calculated by taking the average probability of winning for the various rolls, as each roll is as likely as the others. Below is the equation

$$\sum_{x=1}^{6} P(CallerWin, Roll = x) \Big/ NumPossRolls \quad (3)$$

In the example of the “accept five” sub-game, using equation (3)

$$\sum_{x=1}^{6} P(CallerWin, Roll = x) \Big/ NumPossRolls = 1/6$$

This probability is then used in the subsequent trees where the opponent has the option to accept a call of five. Figure 6 is a diagram of the “accept four” sub-game tree, showing where the “accept five” sub-game tree is used.

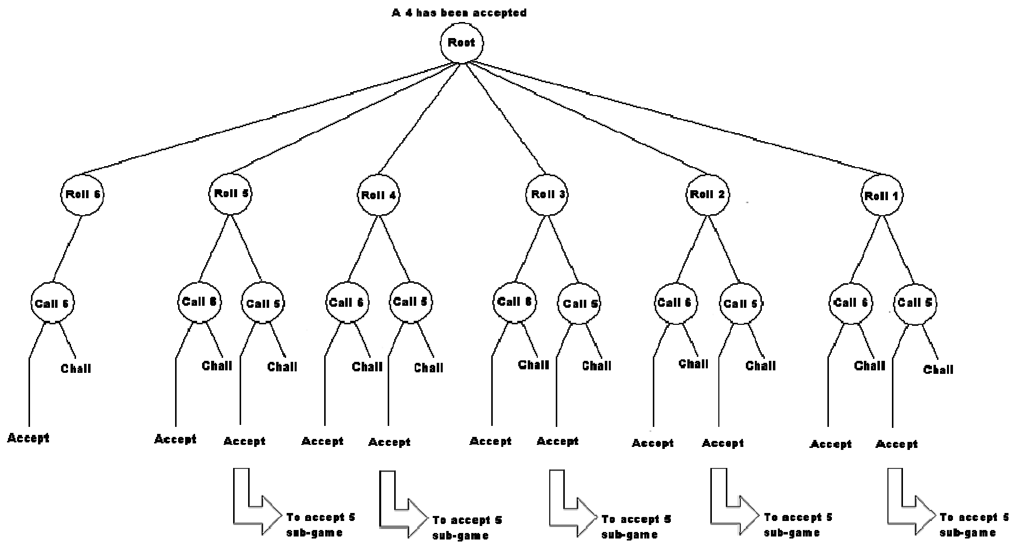


Figure 6 The "accept four" sub-game tree

This process was repeated for all the subsequent sub-games until there are no more sub-games. At this stage there will be a call decision for every possible roll.

3.3.1.1.2. For the accept or challenge decision

The decision to accept or challenge a call is based upon the probability of Player 2 winning if the call of Player 1 is accepted, together with the probability that Player 2 will better that call of Player 1 if the call is accepted and the probability of winning if the call is challenged together with the probability that Player 1 actually rolled what was claimed if Player 2 decides to challenge the call.

The probability of winning if a call of "x" is accepted is the probability in the root of the "accept x" sub-game tree. But this does not take into account the probability that Player 2 might roll a higher value, and so the probability returned from the root node of the sub-game tree must be multiplied by the probability that Player 2 will better the call. This is calculated from the number of possible rolls that are higher than the call that is being considered and then dividing it by the total number of possible rolls, and will give the probability that Player 2 will accept the call and then roll a higher value than the call of Player 1.

The probability of Player 2 winning if the call is challenged is the number of ways a challenged call results in a win for Player 2 over the total number of calls, but this does not take into account the probability that the call was actually rolled. The probability of rolling one particular value is one over the total number of possible rolls. This probability must be multiplied with the probability of winning if the call is challenged in order to obtain the probability that Player 2 challenges the call and that Player 1 lied. In the case of the "accept five" sub-game tree, for a call of six

$$P(A/CWin|Call{=}6,A/C{=}A) = 0$$
$$P(A/C = A, Roll{>}Call) = 0$$

$$P(A/CWin|Call{=}6,A/C{=}A) \cap P(A/C = A, Roll{>}Call) = 0$$

$$P(A/CWin|Call=6,A/C=C) = Num(P(CallerWin|A/C=C)=0)/Num(A/C=C)$$

$$P(A/CWin|Call=6,A/C=C) = 5/6$$

$$P(Roll=Call) = 1/6$$

$$P(A/CWin|Call=6,A/C=C) \cap P(Roll=Call) = 1/6*5/6$$

$$P(A/CWin|Call=6,A/C=C) \cap P(Roll=Call) = 5/36$$

These results reveal that the probability of Player 2 accepting a call of six and then rolling a value higher than six is zero, while the probability of Player 2 challenging the call and it being a lie (which results in a win for Player 2) is five over thirty six, and so it is more probable that Player 2 would win if the call is challenged. In a case like this, the agent would decide to challenge the call. Calculating these probabilities for all the possible calls would give the agent an accept or challenge decision for all the possible calls, but again this gives only one decision for each possible call.

3.3.2. Using matrices

As with the tree data structure, rather than having one large matrix to represent the whole game, the game was broken down into smaller sub-games, each with their own matrix. Again the reason for this is to reduce the complexity of the game, as well as to model the game as a Markov Chain, and solve the game using Dynamic Programming. These sub-games are similar to those used in the tree structure in that they are the sub-games that arise after a certain call has been accepted. For example the “accept four” sub-game is the sub-game that occurs after a four has been accepted.

One call strategy and one accept or challenge strategy references one element in a sub-game matrix. Each one of these elements hold the payoffs that each player can hope to gain from playing a particular call strategy against a particular accept or challenge strategy. Each sub-game matrix then holds all the payoffs for all the possible call and accept or challenge strategies in that particular sub-game. Figure 7 is a diagram of an example sub-game matrix showing how one call strategy and one accept or challenge strategy produce a payoff for both the strategies.

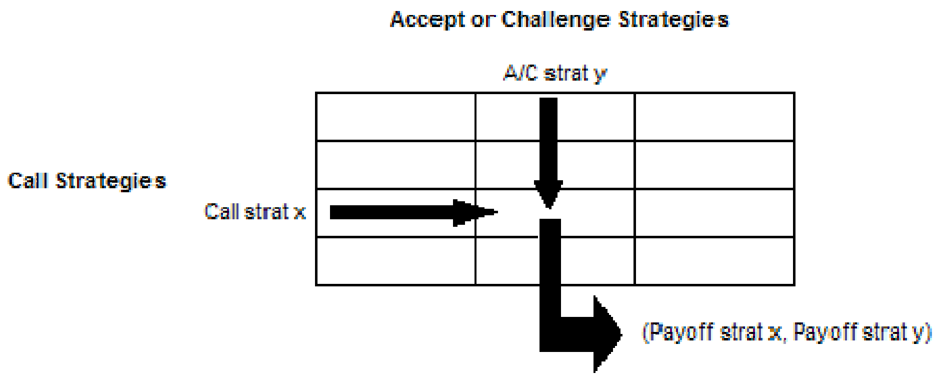

Accept or Challenge Strategies
A/C strat y
Call Strategies
Call strat x
(Payoff strat x, Payoff strat y)

Figure 7 A sub-game matrix showing how one call strategy and one accept or challenge strategy produce a payoff for each of the strategies

The call strategies dictate what the player that has rolled the dice (Player 1) should call for the possible rolls on the dice, for example, call a three if a one is rolled, call a two if a two is rolled, call a three if a three is rolled, and so on for all the possible roll values. The accept or challenge strategies give the player that has to accept or challenge a call (Player 2) the decision to either accept a particular call or challenge it, for example, one strategy might be to accept calls of one, two and four, and challenge calls of three five and six.

For each sub-game, before the payoffs can be calculated, the matrix must be set up with the correct number of call strategies and accept or challenge strategies. In order to do this, the correct number of possible call strategies must first be determined. The call strategy is dependent on what call has been accepted; as a call strategy in a sub-game that contains a call lower than the call that has been accepted in the previous sub-game would result in a loss. As well as this, a call must always be greater than or equal to the value rolled on the die. Once the correct number of call strategies has been calculated, then the correct number of accept or challenge strategies has to be calculated. After the number of call and accept or challenge strategies are know the table can be set up, and the payoffs calculated. Details of how these payoffs are obtained is dealt with later in this chapter under section 3.3.2.1. From these payoffs, the optimal strategy or strategies for both players in a sub-game can be determined through finding the Nash Equilibrium. In a two-player zero-sum game, like Liar's

Dice, this is equivalent to computing a minimax strategy [Conitzer V, Sandholm T 2006].

To decide on a call strategy for a particular sub-game, firstly the accept or challenge strategies with the highest payoff to Player 2 are found. This is the maximum payoff that Player 2 can expect to obtain. This must then be minimised. This is achieved by finding the caller strategy, that when played against the maximum payoff strategies of Player 2, results in minimum payoff for Player 2. As Liar’s Dice is a zero-sum game, this coincides with the strategy with the highest payoffs to Player 1. This results in the Nash Equilibrium strategy for Player 1. If this results in more than one strategy being selected for Player 1, then these strategies each have an equal chance of being selected as the strategy that Player 1 will use in that sub-game, and this is the Mixed Nash Equilibrium strategy. The call decision can then be extracted from the strategy that has been chosen, depending on what was rolled.

In order to decide on an accept or challenge strategy, the Nash Equilibrium for Player 1 must first be calculated. The same method to decide on the call strategy is applied, to find the most likely strategies or strategy that Player 1 will adopt. Again one of these likely strategies is selected and played against the strategies that will give Player 2 the highest payoff. From these, the probability of each of these strategies winning against the particular call strategy is calculated. The more probable a strategy is to win over the selected call strategy, the more likely it is that it will be chosen as the accept or challenge strategy to be played. Once one strategy has been chosen, the call dictates whether or not the call should be accepted or challenged.

3.3.2.1. Calculating the payoffs

As the payoffs are the utility a player can hope to gain from adopting a strategy, the payoffs are modelled as a function of the probability that Player 1 would win if a particular call strategy is selected and played against an accept or challenge strategy. The payoffs could be calculated from probabilities of winning for Player 2, but this would make no difference, as Liar’s Dice is a zero sum game, and so the payoffs would be the same. The function is used to model the payoffs because the higher the probability of winning, the better the chance of winning, and so the higher the expected payoff. This function is:

Payoff to caller = (2x-1) where x = P(CallerWins) (4)

Considering Liar’s Dice is a zero sum game, the payoffs to each player must add up to zero, and so the payoff for Player 2 is the negative of the payoff that Player 1 would receive:

Payoff to acceptOrChall = - (2x-1) where x = P(CallerWins) (5)

Once the matrices are set up to the correct specifications, each possible call strategy is played against each of the possible accept or challenge strategies in order to calculate the probability that Player 1 wins. When one call strategy is played against one accept or challenge strategy, the call strategy gives a call for each individual roll and the accept or challenge strategy determines whether or not that call would be accepted or challenged. There are three cases that can arise from each individual call in the call strategy being played against an accept or challenge strategy; the call could be accepted, the call could be challenged and the call was the truth or the call could be challenged and the call was a lie.

In the case where the accept or challenge strategy opts to accept the call that the call strategy suggests for a particular roll, it does not matter if the roll is equal to the call or not, as the call has been accepted. When a call is accepted, the probability of winning cannot be determined yet as the game has not ended yet, and so the sub-game matrix of the accepted call is created and played. But in this sub-game Player 2 is now the player that has to roll the die, and Player 1 now has to accept or challenge the other players call. The highest payoff for the player accepting or challenging the call in this sub-game matrix is returned as the payoff that the player calling in the first sub-game would hope to gain if a call is accepted. This payoff is then converted back to a probability, as the probability, not the payoff is being calculated. This probability is the probability that the individual call would win if the call is accepted.

The other two cases both result in the termination of the game, as the call has been challenged. If the call was a lie, then the probability that Player 1 decision would win is zero, and if the call was the truth, then the probability that Player 1 would win is one. Figure 8 is a diagram showing how a call strategy is played against an accept or challenge strategy.

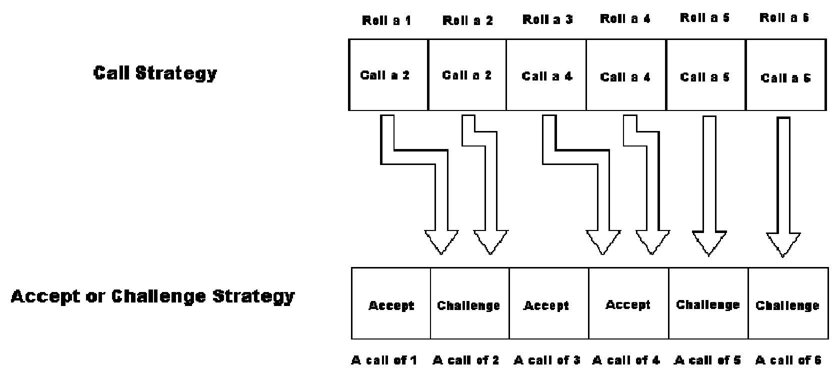


Figure 8 A call strategy being played against an accept or challenge strategy

In this example, if Player 1 rolls a one, a two is called, which will then be challenged by the other player, and so the probability that this individual call in the call strategy will win is zero. If Player 1 rolls a two, the strategy dictates that a two should be called. Again Player 2 will challenge it, but in this case, Player 1 has told the truth, and so the probability that this individual call in the call strategy will win is one. This is the same for the instances where Player 1 rolls a five and calls a five, or rolls a six and calls a six, as Player 1 has not lied and these calls will be challenged by Player 2. If Player 1 rolls a three or a four, the call strategy tells Player 1 to call a four, which Player 2 will then accept. In these cases, the sub-game that follows is the "accept four" sub-game, and the probability of Player 1 winning is extracted as described above.

After having played each individual call in the call strategy against an accept or challenge strategy, the probabilities of winning for each individual call are added together and divided by the total number of calls. This gives the probability that Player 1 would win when adopting that particular call strategy. Using this probability and the function in Equation 4, this will give the payoff the caller would win if it selected that call strategy. Using Equation 5 will give the payoff that Player 2 would hope to gain from choosing its particular strategy. These payoffs are then used as described above to determine the optimal strategy to play for both players.

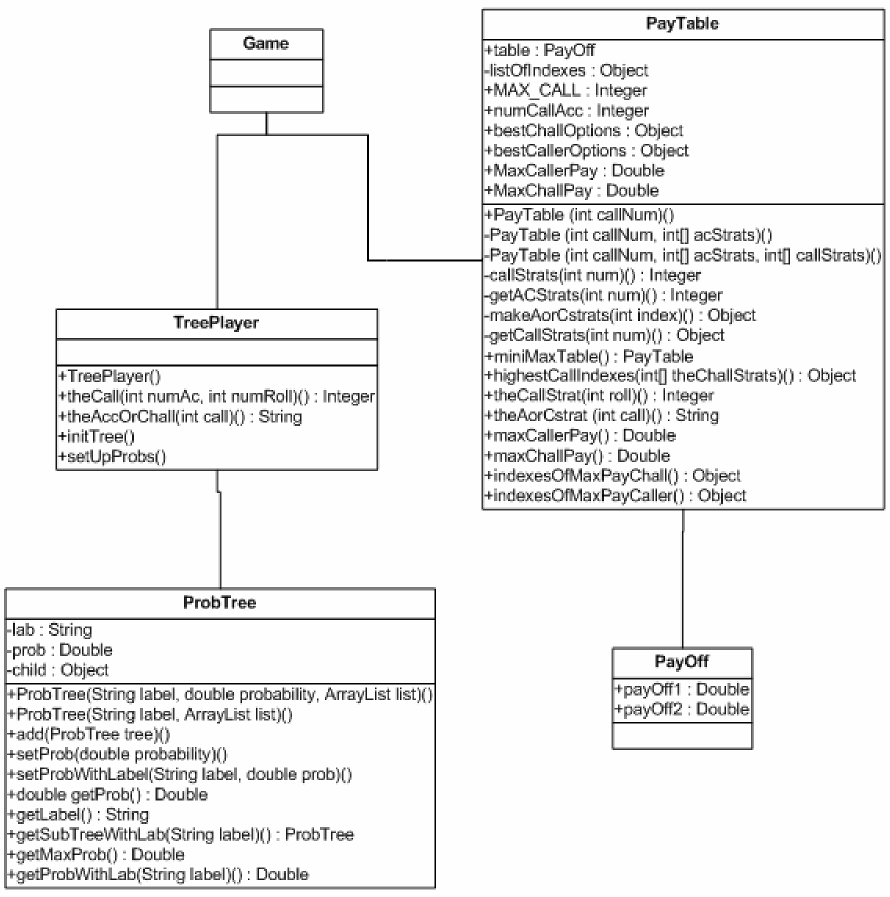
3.4. Summary

With the use of two different data structures two agents can be produced to play Liar’s Dice. The one agent is implemented using a tree used a pure strategy to decide what to call and whether to challenge or accept a call. The call decision is based on the probability that the agent will win if it makes a certain call, while the decision to accept or challenge a call is based on the probability that the agent will win if the call is accepted and the probability that the agent would win if the call is challenged. The second agent is implemented using a payoff matrix, and utilises a mixed strategy to decide on a call and whether to accept or challenge a call.

Chapter 4

4. Implementation

This chapter deals with how the agent was actually implemented, firstly using a tree and then using a matrix. Both were implemented using Java 1.5.0_06-b05 as the programming language, the reason for this is that I have a better understanding of Java compared to other programming languages. Figure 9 shows the UML diagram of the system.



Game
PayTable
+table : PayOff
-listOfIndexes : Object
+MAX_CALL : Integer
+numCallAcc : Integer
+bestChallOptions : Object
+bestCallerOptions : Object
+MaxCallerPay : Double
+MaxChallPay : Double
+PayTable (int callNum)()
-PayTable (int callNum, int[] acStrats)()
-PayTable (int callNum, int[] acStrats, int[] callStrats)()
-callStrats(int num)() : Integer
-getACStrats(int num)() : Integer
-makeAorCstrats(int index)() : Object
-getCallStrats(int num)() : Object
+miniMaxTable() : PayTable
+highestCallIndexes(int[] theChallStrats)() : Object
+theCallStrat(int roll)() : Integer
+theAorCstrat (int call)() : String
+maxCallerPay() : Double
+maxChallPay() : Double
+indexesOfMaxPayChall() : Object
+indexesOfMaxPayCaller() : Object
TreePlayer
+TreePlayer()
+theCall(int numAc, int numRoll)() : Integer
+theAccOrChall(int call)() : String
+initTree()
+setUpProbs()
ProbTree
-lab : String
-prob : Double
-child : Object
+ProbTree(String label, double probability, ArrayList list)()
+ProbTree(String label, ArrayList list)()
+add(ProbTree tree)()
+setProb(double probability)()
+setProbWithLabel(String label, double prob)()
+double getProb() : Double
+getLabel() : String
+getSubTreeWithLab(String label)() : ProbTree
+getMaxProb() : Double
+getProbWithLab(String label)() : Double
PayOff
+payOff1 : Double
+payOff2 : Double

Figure 9 UML diagram of the system

The `Game` class is the class where the games of Liar's Dice are played. When a decision is required from the Tree agent, the tree is first set up using the `initTree()` method in the `TreePlayer` class. The probabilities are the calculated using the `setUpProbs()` method. Once the probabilities are calculated, the call and accept or challenge strategy can be obtained using the `theCall(int callAcc, int numRoll)` and `theAccOrChall(int call)`. Details of how this was done are covered under section 4.1 of this chapter.

If on the other hand a decision is required for the Matrix agent, the matrices are first set up using the constructor of the `PayTable` calss, and then the Nash Equilibrium is calculated using the `miniMax()` method. The details of how this is implemented are discussed in section 4.2 of this chapter. From this, a call and accept or challenge decision can be extracted.

4.1. Tree agent

Before the probabilities required for the decisions can be calculated, the tree must be set up. The tree as a whole is made up of a number of objects called `ProbTree` objects. Each of these objects has a label, probability and an `ArrayList` that holds the sub-trees. A `ProbTree` is created to hold all the sub-game trees first, then the root node of each sub-game, followed by the roll nodes. Next the call nodes are created, and lastly the accept or challenge nodes. Once all are set up with the correct number of nodes, the accept or challenge nodes are added to the call nodes, which are added to the roll nodes, which in turn are added to the sub-game root node. The sub-game root node is then added to the tree This is done using the `initTree()` method of the `TreePlayer` class. Below is the pseudocode to set up the tree.

```
instantiate tree
for all sub-games
{
      instantiate sub-game tree
      for all possible rolls
            {
                  instantiate roll tree
                  for all possible calls
                  {
                        instantiate call tree
                        for either accept or challenge
                        {
                              if accept
                              {
                                    instantiate accept tree
                                    add accept to call
                              }
                              else
                              {
                                    instantiate challenge tree
                                    add challenge to call
                              }
                        }
                  add call to roll
                  }
            add roll to sub-game
            }
   add sub-game to tree
   }
```

Now that the tree has been set up, the probabilities can be calculated using the formulae discussed in the previous chapter. This is accomplished with the `setupProbs()` method in the `TreePlayer` class. Once the probabilities are calculated the tree is written to a file so that the tree does not have to be created every time a decision is needed. Below is the pseudocode to calculate the probabilities.

```
for all sub-games
{
      totalRollProb = 0
      for all possible rolls
      {
            for all possible calls
            {
                  for either accept or challenge
                  {
                        if accept
                              ProbInAcc = prob in root of call
                        if (challenge && call > roll)
                              ProbInChall = 0
                        else
                              ProbInChall = 1
                  }
                  ProbWinAcc = ProbInAcc * (1-ProbInAcc)
                  ProbWinChall = (1-1/call)/(ProbInAcc+(1-1/call)
                  if (call > roll)
                        ProWinCall = ProbWinAcc
                  else if (call == roll)
                        ProWinCall = ProbWinAcc + ProbWinChall
            }
            ProbWinRoll = MaxProb(Call)
            totalRollProb += ProbWinRoll
            }
      ProbWinSubGame = totalRollProb/6
}
```

Now that the probabilities of winning for the various calls in the sub-games have been saved in a file, the `TreePlayer` method `theCall(int callAcc, int numRoll)` returns the call decision based on the call that was accepted before rolling the die and the value that was rolled on the die while the `TreePlayer` method `theAccOrChall(int call)` returns an accept of call decision based on the call that has been made.

4.2. Matrix agent

The matrices are set-up, and the payoffs calculated in the constructor of the `PayTable` calls. This constructor takes the call value that was accepted to lead to the sub-game as a parameter. This is used to calculate the number of call and accept or challenge strategies. Once the number of each of these strategies has been worked out, the matrix can be created to the correct size, and each call strategy can be played against the accept or challenge strategies, and the payoffs calculated. Below is the pseudocode to set up a `PayTable` object.

```
numCallStrats = getNumCallStrats
NumAorCStrats = getNumAorCStrats
table = new paytable[numCallStrats][NumAorCStrats]

for all callStrats
{
    for all AorCStrats

   {
            int[] oneCallStrat = makeCallStrat(CallStrat)
            int[] oneAorCStrat = makeAorCStrat(AorCStrat)
            totalProb = 0

            for (i = 0; i < oneCallStrat.length; i++)
            {
                  call = callStrat[i];
               roll = i + 1;
               aOrC = acStrat[call-1];

                  if(aOrC==accept)
                  {
                        PayTable subGame = new PayTable(call);
                        maxAorCPay = subGame.getMaxPay
                        maxProb = (maxPay+1)/2
                        totalProb += maxProb
                  }

                  if((aOrC == chall && roll == call))
                        totalProb += 1.0;
            }

            aveProb = totalProb/oneCallStrat.length

            payOff1 = (2* aveProb) - 1;
            payOff2 = payOff1 * -1;

         table[CallStrat][AorCStrat].payoff1 = payOff1;
         table[CallStrat][AorCStrat].payoff2 = payOff2;
      }
}
```

Each strategy is represented as an array of integers. For the call strategies, the index value is the possible roll value, and the integer stored at each array index is the call that will be made for that particular roll. The accept or challenge strategy is also represented as an array, but the index value represents each possible call value and the integer stored at each array index is either a one or a zero to represent either an accept or challenge decision. A one represents an accept while a zero represents a challenge decision.

In the case where a call is accepted, a new sub-game is created where the caller is now the player accepting or challenging a call, and so the maximum payoff for the player making the accept or challenge decision is calculated, and the probability of winning calculated from the payoff. The only other case where there is a probability of winning is where the player accepting or challenging a call challenges a call that is the truth. This gives the player calling a probability of winning of one. All these probabilities are added and averaged over the number of possible calls, and the sub-sequent payoff calculated.

The next step is to find the mixed strategy Nash Equilibrium. As stated in the Design chapter, in a two-player zero-sum game, like Liar's Dice, this is equivalent to calculating the minimax strategy. The following pseudocode shows how this was achieved.

```
MaxAorCstrats = PayTable.get aOrCStrats with maxPay to challenger

PayTable bestChallTable = new PayTable (callerStrats, MaxAorCstrats)

MaxCallerStrats = bestChallTable.get callerStrats with maxPay to caller
```

```
PayTable bestCaller = new PayTable (MaxCallerStrats, MaxAorCstrats)
```

First the strategies accept or challenge strategies that return the highest payoff to the player accepting or challenging the call are found. A new `PayTable` object is created with only these accept or challenge strategies which are played against all the possible call strategies. The indexes of the caller strategies that return the highest payoff to the player making the call are found, and a new `PayTable` object created with just these call strategies being played against the best strategy options for the player accepting or challenging the call. These caller strategies are the mixed Nash Equilibrium and each has an equal chance of being selected as the strategy that the caller should follow. When a call is required for a certain roll, the `theCallStrat(int roll)` selects one of the call strategies at random, and looks up the call for the given roll. When an accept or challenge strategy is required, the `theAorCstrat(int call)` method is called. In this method one of the call strategies is selected, followed by an accept or challenge strategy that is selected at random based on the probability that it would win over the selected call strategy. This probability is determined from the payoff. The higher the probability of winning for the accept or challenge strategy, the higher the probability that the strategy will be selected.

The first implementation of the Matrix agent included sub-game matrices for the start of the game, the accept one, accept two, accept three, accept four and accept five. The time taken to set up the start and accept one sub-game matrices was over ten hours, due to the large number of call strategies available in these tables. The number of call strategies is the factorial of the maximum call, which in the first implementation was six, giving seven hundred and twenty possible call strategies in the starting sub-game matrix. In order to reduce the number of call strategies, instead of having a maximum call, a maximum acceptable call was used, which in this case was a call of five. This can be done, as the probability of winning if a six is accepted can be calculated from simple probabilities. This reduced the number of call strategies from seven hundred and twenty to one hundred and twenty. The starting sub-game matrix then took seven hours to compute, but was saved once set up, so that it only had to be calculated once.

Chapter 5

5. Results

This chapter looks at the results that are obtained when the agents are tested against each other and then each against a human opponent. As there is an advantage to the player that has to roll first, the first half of the testing is performed with the one player playing first, and the second half with the other player playing first.

5.1. Tree agent Vs. Matrix agent

The agents are played against each other a total of ten thousand times, with the tree playing first for half of them and the matrix agent playing first the second half. Below are the results obtained from the tests.

	Tree playing first	Matrix playing first	Total wins
Num Matrix wins	1297	2725	4022
Num Tree wins	3703	2275	5978

Table 1 Results for Tree agent Vs. Matrix agent

When the Tree agent is playing first, it is expected that it would win more than the Matrix agent but not by much. This is not the case, as the Tree agent wins over seventy percent of the games. In the cases where the Matrix agent plays first, the Matrix agent wins over half of the games, which is the result that is expected. Figure 9 a graphical representation of the results obtained.

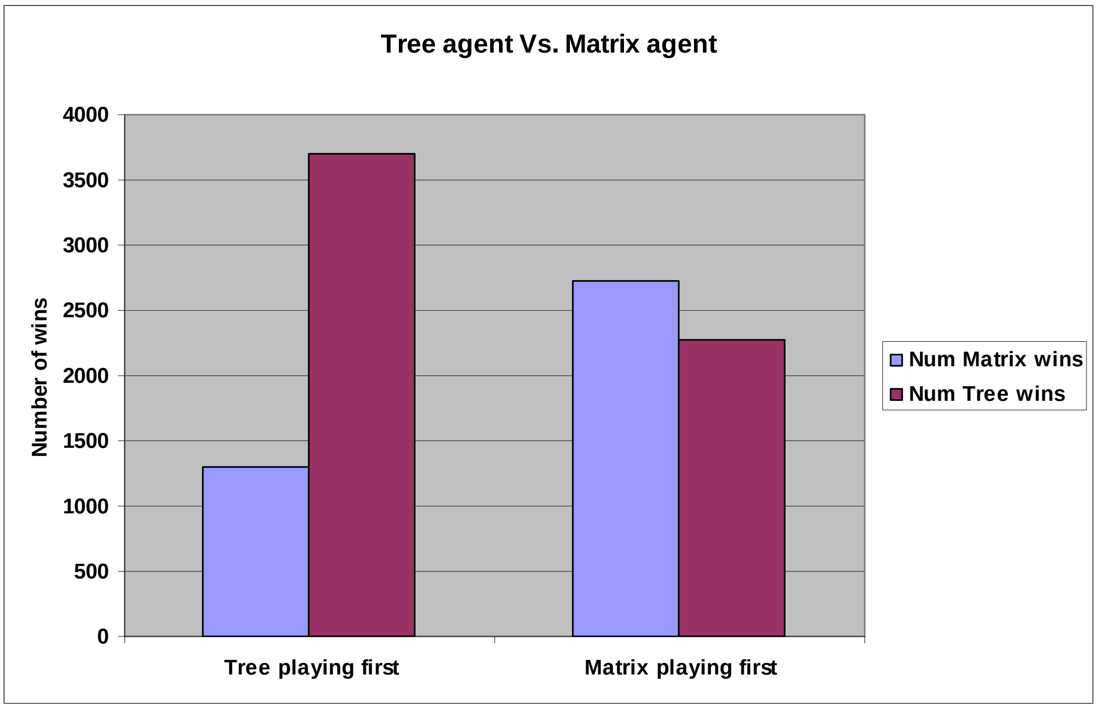


Figure 10 Tree agent Vs. Matrix agent

The reason for the bad performance of the Matrix agent when the Tree agent is playing first is due to the fact that the Matrix agent can not exploit the pure strategy adopted by the Tree agent, as it has been programmed to make the best decision to win the game and so there is a weakness in the opponent, the Matrix agent would not be able to exploit the weakness.

5.2. Human player Vs Human player

These tests serve as control data, to see the number of games won when two human players play Liar's Dice against each other. The testing was done with the one human player, Player one, playing first in half the games, and with the other, Player two, playing first in the second half of the games. The table below shows the results obtained.

	Player one playing first	Player two playing first	Total wins
Num Player one wins	35	29	64
Num Player two wins	25	31	56

Table 2 Results for Player one Vs. Player two

The results show that the player playing first does have a slight advantage, because in the games where Player one plays first, Player one wins more than half of the games, and when Player two plays first, Player two wins more than half.

5.3. Matrix agent Vs. Human player

Next the Matrix agent played against a human player in sixty games against a human player, and again each player started half the games. Below is a table of the results gathered from testing.

	Matrix playing first	Human player playing first	Total wins
Num Matrix wins	36	28	64
Num Human player wins	24	32	56

Table 3 Results for Matrix agent Vs. Human player

These results show that the agent can play this simplified version of Liar's Dice as well as a human player, as the total number of wins is nearly the same for both the human player and the Matrix agent. When the Matrix agent played first, the results show that agent won more often, but not by much, which is what is expected. With the human player playing first, the human player wins more often, which is expected, but the human player does not win as many games playing first as the Matrix agent. Figure 10 shows a graphical representation of the results.

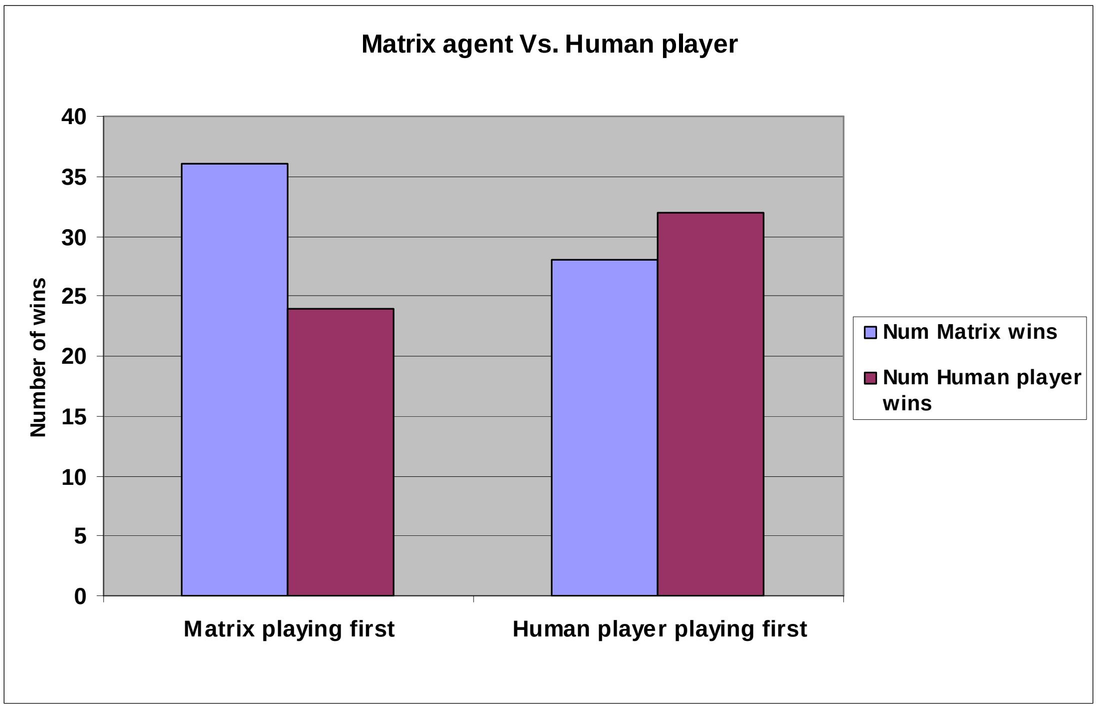


Figure 11 Matrix agent Vs. Human player

5.4. Tree agent Vs. Human player

The final testing performed is to play the Tree agent against a human player. The reason for this is to see how the Tree agent performed against an opponent that might be able to exploit the weaknesses in the pure strategy that it adopts. The Tree agent started thirty of the games, while the human player started the remaining thirty. Below is a table of the results accumulated during the test.

	Tree playing first	Human player playing first	Total wins
Num Tree wins	33	18	51
Num Human player wins	27	42	69

Table 4 Results for Tree agent Vs. Human player

As expected, the Tree agent wins more often than the human player when the Tree agent was rolling first, and the human player wins more often than the Tree agent when the human player was rolling first. But in the case where the human player is rolling first, the human player won about seventy percent of the time. This is due to the human player picking up on the pure strategy that the Tree agent had selected, and exploiting it to his advantage. Figure 11 shows a graphical representation of the results.

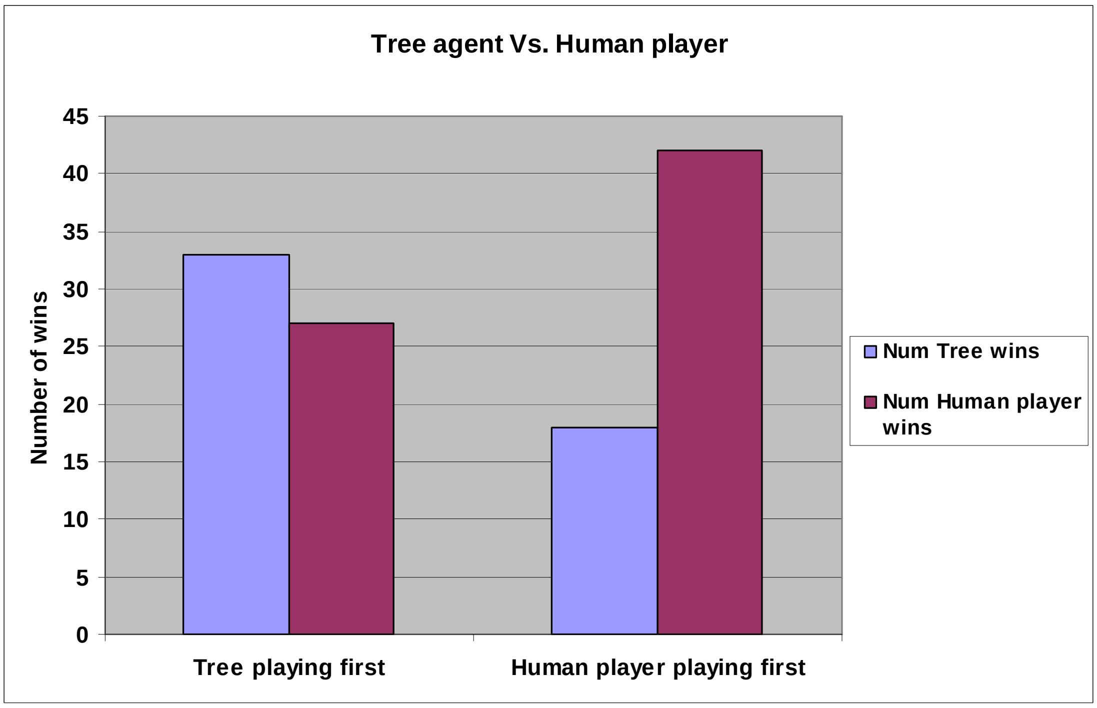


Figure 12 Tree agent Vs. Human player

5.5. Summary

When the two agents play against each other with the Matrix agent playing first, the Matrix agent could be expected to win just more than half of the games. But when the Tree agent is playing first, the Tree agent can be expected to win well over half of the games. When each agent is played against a human opponent on the other hand, the Matrix agent out performs the Tree agent. Both these agents are capable of playing Liar's Dice, but the Tree agent is more successful against the Matrix agent, while the Matrix agent is more successful than the Tree agent against human opponents.

Chapter 6

6. Conclusions

This chapter discusses the contributions that this project has made as well as future work for this project.

Two agents that can play a simple one dice version of Liar’s Dice were produced, the first is implemented as a tree and utilises a pure strategy, while the other is implemented as a matrix and adopts a mixed strategy. These were both created using Dynamic Programming to set up the data structures.

The Tree agent plays well against the Matrix agent, as the Matrix agent could not exploit the pure strategy that the Tree agent had adopted. But when the Tree agent plays an opponent that can exploit this weak pure strategy, the performance of the Tree agent is reduced, especially in the case when the opponent to the Tree agent is the first player to roll the die.

The Matrix agent, as stated above, does not perform as well as the Tree agent when the two are played against each other, but when the Matrix agent is played against a human opponent, the Matrix agent performs as well as a human would expect to when playing first and when the opponent is playing first, and so is the more optimal of the two agents.

6.1. Contributions

This project analysed how Dynamic Programming can be used to produce agents that can play Liar’s Dice. This showed that Dynamic Programming is a good programming technique to produce big data structures like the ones required for Liar’s Dice.

As well as this, the project investigated how the use of different strategies in a game like Liar's Dice can affect the outcome. This project shows that using a pure strategy, like the one adopted by the Tree agent, in games that require bluffing is not effective strategy against human opponents, and so a mixed strategy, like that of the Matrix agent, is required.

6.2. Future work

Future extensions to this project could include extending the one dice version to two or more dice. This extension would require either tactics or strategies to decide which dice to roll after accepting a call. Another possible extension could be to include another player into the game. This would require the use of Linear Programming to determine the mixed Nash Equilibrium to commit to.

References

Nemhauser G. *Introduction to Dynamic Programming*, John Wiley & Sons Inc, 1966.

Blatt S. *RISKy Business: An In-Depth Look at the Game RISK*. Technical Report, Elon University, 2000.

Cremers C.J.F. *How to Beat High Scores in Yahtzee*. MSc Thesis, Technical University of Eindhoven, 2002.

Verhoeff T. *Optimal solitaire Yahtzee strategies*. Presentation, Technical University of Eindhoven, 2000.

Neller T.W, Presser C.M.G. *Optimal Play of the Dice Game Pig.* Technical Report, Gettysburg College, 2004.

Bertomeu J. *On Bluff in Poker*. Technical Report, Carnegie Mellon University, 2004.

Gilpin, A, Sandholm T. *Finding Equilibria in Large Sequential Games of Imperfect Information*. EC '06, June 11-15, 2006.

Hoehn B, Southey F, Holte R.C, Bulitko V. *Effective Short-Term Opponent Exploitation in Simplified Poker.* American Association for Artificial Intelligence, 2005.

Billings D, Davidson A, Schaeffer J, Szafron D. *The Challenge of Poker.* Vol. 134, Issue 1-2 (January 2002), ISSN: 0004-3702, 2002.

Reiley D.H, Urbancic M.B, Walker M. *Stripped-Down Poker, A Classroom Game with Signaling and Bluffing*, Technical Report, 2005.

Conitzer V, Sandholm T. *Computing the Optimal Strategy to Commit to*. ACM 1595932364/06/0006, Carnegie Mellon University, 2006.

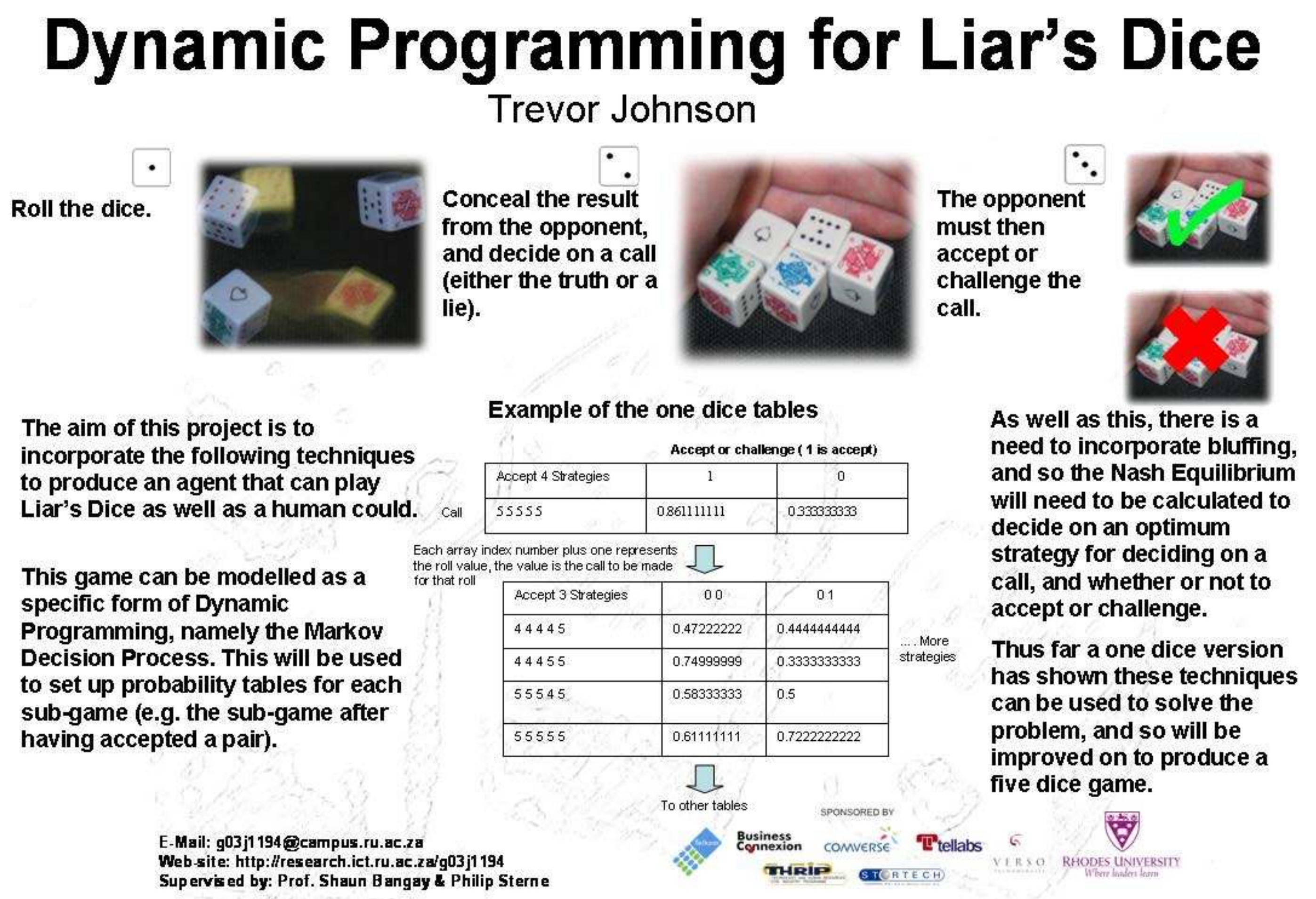
Miltersen P.B, Sørensen T.B. *Computing sequential equilibria for two-player games.* Technical Report, University of Aarhus Denmark, 2005.

Freeman G.H. *The Tactics of Liar's Dice.* Applied Statistics, Vol. 38, No. 3 (1989), 507-516, 1989.

Shor, Mikhael, "Nash Equilibrium,“ Dictionary of Game Theory Terms, Game Theory .net, http://www.gametheory.net/dictionary/NashEquilibrium.html, Accessed 2 November 2006

Pearson Education, "Game Theory," Pearson Education, http://media.pearsoncmg.com/intl/ema/ema_uk_he_lipczynski_indorg_2/0273688073_glossary.html, Accessed 2 November 2006

Appendix A



Dynamic Programming for Liar's Dice

Trevor Johnson

Roll the dice.

Conceal the result from the opponent, and decide on a call (either the truth or a lie).

The opponent must then accept or challenge the call.

The aim of this project is to incorporate the following techniques to produce an agent that can play Liar's Dice as well as a human could.

This game can be modelled as a specific form of Dynamic Programming, namely the Markov Decision Process. This will be used to set up probability tables for each sub-game (e.g. the sub-game after having accepted a pair).

Example of the one dice tables

Accept or challenge (1 is accept)

	Accept 4 Strategies	1	0
Call	5 5 5 5 5	0.861111111	0.333333333

Each array index number plus one represents the roll value, the value is the call to be made for that roll

Accept 3 Strategies	0 0	0 1
4 4 4 4 5	0.47222222	0.4444444444
4 4 4 5 5	0.74999999	0.3333333333
5 5 5 4 5	0.58333333	0.5
5 5 5 5 5	0.61111111	0.7222222222

.... More strategies

To other tables

As well as this, there is a need to incorporate bluffing, and so the Nash Equilibrium will need to be calculated to decide on an optimum strategy for deciding on a call, and whether or not to accept or challenge.

Thus far a one dice version has shown these techniques can be used to solve the problem, and so will be improved on to produce a five dice game.

E-Mail: g03j1194@campus.ru.ac.za
Web-site: http://research.ict.ru.ac.za/g03j1194
Supervised by: Prof. Shaun Bangay & Philip Sterne

SPONSORED BY