5-2013

# Roulette Wheel Selection Game Player

Scott Tong
*Macalester College*, stong101@gmail.com

5-1-2013

# Roulette Wheel Selection Game Player

Scott Tong

# Roulette Wheel Selection Game Player

Scott Tong
Advisor: Susan Fox

April 6th 2013

# Abstract

General Game Playing is a field of artificial intelligence that seeks to create programs capable of playing any game at an expert-level without the need for human aid. There are two major approaches to general game playing: simulation and heuristic. I focused on the move selection component of a common simulation strategy called Monte Carlo Tree Search. Traditionally, the selection step of Monte Carlo Tree Search uses an algorithm called Upper Confidence Bound Applied to Trees or UCT. In place of this algorithm, I investigated the applicability of a random roulette wheel style of selection. I studied the effectiveness of this roulette wheel style selection using tic-tac-toe and nim. The game player built from Roulette Wheel selection performed well against its opponents. It demonstrated the strengths of a flexible planning strategy throughout these games.


# 1 Introduction

General Game Playing seeks to make artificial intelligence capable of playing any game at an expert-level without the need for human intervention. This is different from previously established artificial intelligence, such as *Deep Blue*, which are capable of expert-level or even master-level play. The crucial distinguishing factor between General Game Players and their predecessors is that despite all of *Deep Blue's* skill in Chess it cannot play tic-tac-toe or any other game. Further, these artificial intelligence tend to rely heavily upon game specific knowledge [Genesereth and Love, 2005]. These artificial intelligence generally use game specific knowledge in the derivation of heuristics to evaluate potential moves. While this results in increased performance, it causes them to make less use of reason and logic. Skills that humans tend to rely on to play games effectively, especially games with which the individual is less familiar. Consequently, we can understand general game playing seeks to develop a deeper understanding of how to make machines reason, a trait closely associated with games.

A general game player should be capable of playing any game. When we think of games and how to define them, we most likely draw upon the rules for the game. General game players also rely on the rules of a game in order to play. However, it would be problematic if every single game player represented game rules in their own way. As one would imagine,

it would make it much more logistically difficult for game players to play any particular game as each new game would have to be encoded in the game players' particular encoding. This problem is mitigated through the development of *game description language* (GDL) [Genesereth et al., 2006]. GDL is a language developed to represent the rules and states of a game. It is used by the majority of general game players .

Simulation game players attempt to play out the game states of a game and make its decision based upon the results of its plays. Essentially this is akin to a Chess player looking at a particular piece and choosing a particular move and then playing the game using this particular move until it has reached a win or loss state in the game or when the simulation is ended. Simulation strategies have different methods to select which particular move to simulate. Another solution to this problem is to simulate every available move. Further, simulation players must have a schema for selecting moves.

In contrast, Heuristic game players decide their next move by evaluating the quality of moves from a particular game state. This is akin to the Chess player looking at a particular game state and deciding the quality of moves available. For example, a simple Chess heuristic might judge a move based on whether or not it results in a piece being lost next turn. The process by which the quality of the moves is decided is often derived from some sort of heuristic algorithm. These heuristics may be derived from many different sources. Although, many heuristic based general game players try to derive at least some of their heuristics dynamically. The importance of dynamically deriving heuristics is that using predefined heuristics can leave the game player vulnerable to games where the predefined criteria are not relevant. Further even if one was able to generate heuristics relevant to every game, these heuristics may be too general to supply much useful information on judging a move. Judging moves seems to also be done in simulation gaming, so it is important to discuss the distinctions between simulation and heuristic gaming.

The primary distinction between the two styles of general game playing is that heuristic game players use their heuristics to attempt to objectively judge whether or not a move is good. For example, an experienced chess player can look at a chess board and tell you whether or not a move is good. Simulation game players use their "heuristics" to compare moves to each other. Simulations use move ratings only with the purpose of comparing a move to another. Simulation move rating systems are not usually applied with the purpose of judging the quality of a single move by itself.

The approach presented in this paper is a modification to Monte Carlo Tree Search (MCTS), a type of simulation search. Monte Carlo Tree Search contains four main steps: selection, expansion, simulation, and back propogation. Roulette wheel simulation seeks to modify the selection step of MCTS by providing an alternative schema by which moves are selected. It works by totaling the scores of the moves currently available in a given game states. The scores of the moves are determined by the sucess of a simulation and the number of steps it took to reach the end result. The scores of the individual moves are then treated as a percentage in a total score. Using the percentages, a move is randomly chosen with the higher scored moves having a larger chance of being chosen.

The results of a game player using Roulette Wheel selection were positive. Roulette Wheel Player was able to obtain a majority win percentage against all of its opponents except for one. The largest contributing factors to its win was its ability to play moves that were a strong balance between exploitative and exploratory. Against the opponent it did not obtain a majority win percent against, Roulette Wheel Player was able to tie in the majority of games.

# 2 Background

## 2.1 Background information

General game playing refers to three core terms utilized in games and general game playing: moves, game states, and game trees. Moves are actions available to the player which allows for the alteration of the game state. The game state is defined as the instance of a particular game after any number of moves have been made. In figure 2.1.1 bellow, a single game state is shown. Figure 2.1.1 is also a special game state, called a terminal or goal state. The terminal state is a game state in which the game has ended. Further the outcome of the game, whether or not a win, loss, or tie has occurred, is decided from the terminal state.

Figure 2.1.1: A terminal game state for tic-tac-toe [tic, a]

The total of all the game states possible in a game is referred to as the game tree. Figure 2.2.1, bellow, is an example of an abridged game tree for tic-tac-toe. This version is abridged by accounting for the symmetries within tic-tac-toe. That is, many on a tic-tac-toe board are exactly the same even if they are made on different spaces. The game tree of a game is all the possible moves in the game. Simple games like tic-tac-toe have small game trees and more complex games like chess have larger game trees. Large game trees are often difficult to navigate and maintain as they are demanding both computationally and in hardware. In general game players moves, game states, and game trees are represented using *game description language*(GDL).

Figure 2.1.2: An abridged Game Tree for Tic-Tac-Toe [tic, b]

Game description language allows for the representation of game moves, states, and, trees. In addition, GDL is also used to represent the rules of the game. GDL is an offshoot of prolog, a logic programming language. [Genesereth et al., 2006]The original specification of GDL was only capable

of representing games without any elements of randomness, such as poker or blackjack. However, GDL 2.0 is capable of dealing with random chance in games. At the moment, GDL is only capable of defining games that have a finite number of moves, a constant number of game states, players, and board spaces. In the future, it is likely that we will see new versions of GDL incorporate additional features so that more games can be represented. GDL is the standard used in the development of general game players as it provides for a consistent way to represent games and their moves, states, and trees.

Figure 2.1.3: Sample GDL of Tic-Tac-Toe Game State and the Game State ir represents [Genesereth and Love, 2005]

General game playing is meant to address any type of game. My research focused on games that were finite, determinable, alternating, and contain no hidden information. Finite games are games that contain a finite amount of moves and game states. Determinable games are games that have clear victories, ties and losses. The previous two traits are generally found in the games currently being explored due to their lower level of complexity. The last quality, alternating games, refers to games in which each player takes turns making a move. This is in contrast to games where there is only one player or games where both players play simultaneously. Hidden information games are games where all of the relevant game information is not known. Poker is an example of a game with hidden information. My research focuses on game where all relevant game information is known. However, there is no particular reason why the strategies discussed in this paper cannot be applied to these types of games.

## 2.2 Simulation Game Players

As stated above, simulation game players select a strategy by running one or more simulations. After the simulations are run, a move is selected by a particular schema. This section gives a overview of the overall approach and examines some of the challenges faced when using it.

Figure 2.1.2, above, shows a hypothetical simulation for Tic-Tac-Toe. In the figure we get an example of a simulation of three moves that may be picked at the start of tic-tac-toe. We then see the next possible step for those moves. The simulations are usually run until the end goal of the game is reached. Figure 2.1.1 demonstrates simulations being run for the multiple moves . The top row or the blank state represents the game state current being examined. The second it contains three of the possible moves that the player could make. If the player used multiple situations than each move would be visited in its own simulation. The third row contains many possible paths the simulation could take, although an individual simulation would only take a single path. Oftentimes, many simulations will be run to account for the many possible paths present when multiple moves are available. Each of these simulations is then run until it reaches a terminal state, a state where the game has finished. The terminal state dictates which of the players has won the game or if there is a tie. Generally, simulation game players run more than one simulation before deciding on their action.

Simulation game players tend to simulate many games when running for two reasons. First, simulating many moves or games allows the player to consider many different solutions. This allows for the player to choose a more optimal move by exposing it to many possible solutions. This is important to do in general game playing because games oftentimes have many solutions. It is important to consider the optimality of the solution in game playing. For the purposes of general game playing, optimal solutions may take a shorter number of moves to win or take moves that prevent the opponent from winning. Second, having many simulations can help to refine an already examined move.

Simulation game players simulate the outcome of a particular move multiple times. This allows for the further refinement of possible solutions. When a particular move is simulated, the simplest simulation will just make random moves. Therefore it is important to simulate a single move many times so that an optimal solution may be converged on. In the third row of figure 2.1.2, there are examples of possible states that can occur in the repeated

simulations of the moves presented in the second row.

Simulation game players have appeal in their iterative style of finding solutions to problems but this approach still has its challenges and difficulties.

One of the primary problems facing simulation game players is answering the question, "which moves do I choose to simulate?". This is an important question as simulation complexities are directly tied to the complexity of the game. Less sophisticated game players simulate every available move. However, there is a serious flaw to this simple simulation approach. A simple game like tic-tac-toe has 765 possible legal positions. With modern computing technologies simulating up to 765 game states from the starting move would not be particularly difficult. However, trying to simulate every starting move in Chess until the end game would be an unreasonable strategy. Therefore, more complex players try to choose particular moves to simulate so as not to waste simulation time. Aside from choosing which moves to simulate, simulation players must have a scheme to compare moves to each other.

Simulation game players judge move quality in different ways. A simple and effective method measures the depth of the move, or how many steps a move took to reach the end-goal to judge the quality of a move. Using this schema, moves that result in a win earlier in the game are prioritized. Other methods for move quality are based around functions that comprise multiple factors; for example, depth of the move, how often it has been simulated, whether the move been previously used etc. Judging moves is important but sometimes game players must account for situations where a given simulation has used up too many resources.

Flow control in simulations is the challenge of running multiple simulations. It is important to prevent a single simulation from consuming all the time and resources to a game player because some games require long simulations to reach a terminal state. Therefore, coming up with an effective scheme to halt simulations becomes important for game players attempting to play complex games such as Go. No single method terminates simulations optimally for all games, but a simple and common method cuts simulations after a certain time has passed. Incomplete simulations can be treated in many ways. A common method is to just treat the move as another candidate ignoring that it never completed. Another method may use a special scoring function for the incomplete move. This scoring mechanism may measure how close the goal state is perceived to be or other factors. Like flow control, there is not a single method that works for every game. Regardless

of the particular solution, it is important to maintain a system of flow control to prevent simulation game players from over-committing to examining a particular move.

## 2.3 Heuristic Game Players

Heuristic game players select their moves based on a heuristic function. Heuristic functions are a mathematical function that evaluates an input and outputs a rating of the input. In general game players, heuristics are used to judge the utility of moves. These heuristics allow the game player to gain information on the quality of the game state. There are many ways to evaluate the state of a given game. For example, we could see how close to a terminal state the current game state is. This a rather simple way of looking at a game because it doesn't differentiate the quality between states equally close to terminal states. For example let us examine the following board positions:

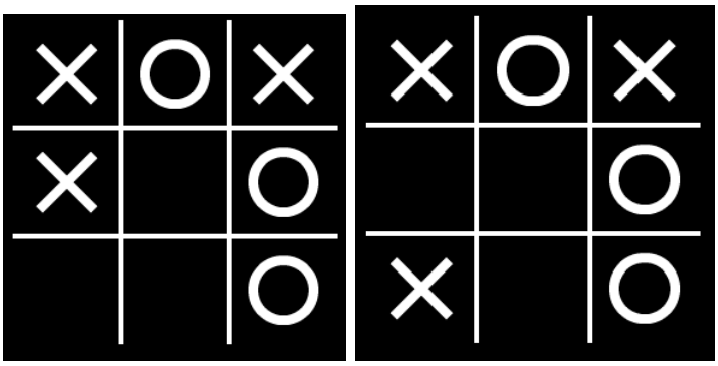

Figure 2.3.1 & Figure 2.3.2

Looking at figure 2.3.1 and figure 2.3.2 we can see that X in figure 2.3.2 is in a much better position because X has two possible ways to win. However, our heuristic above will rate these game states as the same. Figure 2.3.1 and figure 2.3.2 are both 1 move away from victory. We could modify our heuristic to take into account how many game winning states are available in the next move. For tic-tac-toe this is a relatively simple endeavor but for more complex games this may be more difficult. The winners of games such as Go are not determined by which players reaches a terminal goal first. Instead, other factors such as the number of pieces each player has may determine the winner.

We can consider more complex entities in a game to develop our heuristic. For example, we might consider how many pieces a player has on the board.

This proves to be valuable for games like Reversi or Othello where the quality of a move may be directly measured by how many pieces you gain on the board. However, in a game like chess different pieces are worth different values. For example if my opponent in chess keeps all of their pawns and king but I manage to keep my both of my rooks, queen, and king, I am in a better position on the board. We could consider evaluating each piece and assigning value to them. However, some pieces are stronger in certain situations than others. For example, one can argue that the value of a knight or bishop changes depending on the number of pieces on the board. When we consider this example, board position emerges as an important consideration for the development of heuristics

Board position is a crucial component for winning games and, therefore, is important to consider when evaluating game states. In chess, skilled players often play for position, meaning they will sacrifice pieces, potentially very strong ones, to position certain pieces for use later in the game. A simple example of developing position in Chess is having a knight cover a pawn after the opening move. Unfortunately, this strategy is comes from my specific knowledge of Chess. Consequently, I cannot just program in knowledge of move positions. However, information on board positions can be derived from the rules of the game.

Recall that general game players use a universal language, called game description language (GDL), to describe a game. Therefore, heuristics could be parsed through GDL. By default, game players use GDL to interpret the rules of a game in order to determine moves that are legal. However, GDL's use in a game player is not limited to just determining which moves are legal. Instead, individual lines of GDL can be parsed in order to form heuristics. GDL can be parsed in a number of different ways. General strategies for parsing GDL seek to identify common elements found in all games such as critical rules, pieces, and spaces. Critical rules are rules that can directly result in you losing a piece, or the game. Pieces and spaces are more self explanatory. Once, these elements of the game are identified heuristics evaluating the game state are developed within the game player. Game players have to do generate heuristics online so game players using a heuristic strategy have to make trade-offs between speed and quality. Through complex parsing techniques, game players can use GDL to create heuristics by which moves may be evaluated through game specific knowledge.

## 2.4 Monte Carlo Game Players

### 2.4.1 Monte Carlo Simulations

Monte Carlo game players use a technique known as Monte Carlo simulations. Monte Carlo simulations derive solutions by using a random sequence of moves until the solution can be determined. To illustrate how this works let us consider how to approximate $\pi$. Consider a circle inscribed in a unit square. We can then randomly distribute points throughout the square. The number of points needed varies depending on the size of the square. Generally, Monte Carlo simulations want to use as many random moves as they can to approximate their goal. Bellow, figure 2.4.1 illustrates this example.

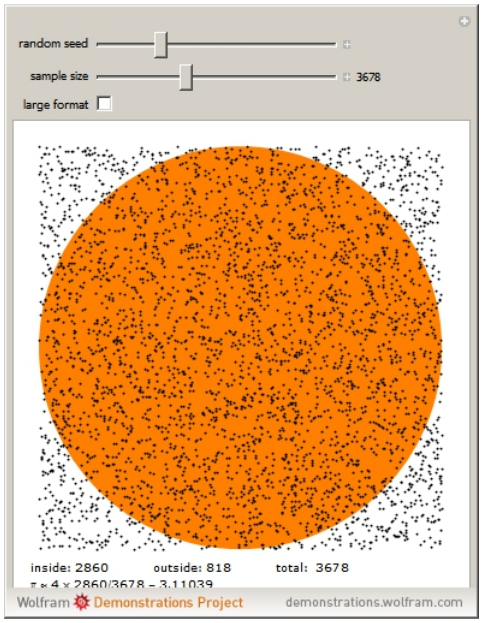


Figure 2.4.1: Monte Carlo simulation to approximate $\pi$ [mon, ]

Monte Carlos' positive qualities are best illustrated through its success in playing Go. Go is a difficult game for artificial intelligence because its game tree is much too large for traditional game search methods.[Browne et al., 2012] Monte Carlo simulations do not require that the entire game tree be stored in memory. Monte Carlo simulations need only enough memory so that they can simulate a game to a terminal state. By their design, Monte Carlo simulations avoid the difficulty and complexities associated with navigating the

whole game tree of a game. Instead, Monte Carlo simulations operate from local game states and run until a terminal state is reached. This approach, mitigates a lot of the difficulties associated with game playing. Further, Monte Carlo strategies do not require much knowledge of the game currently being played. The Monte Carlo algorithm only needs to understand whether or not a move is legal in order to simulate. Consequently, Monte Carlo is a flexible and robust approach that is effective in an environment where vital information is not known until run time, like general game playing.

### 2.4.2 Monte Carlo Tree Search

Monte Carlo Tree Search(MCTS) is the application of Monte Carlo simulations to a tree search. Using Monte Carlo simulations, the algorithm builds a search tree. The search tree in Monte Carlo Tree Search is also referred to as a game tree. This game tree is different from the game tree defined in section 2.1. Their primary difference is that the game tree from section 2.1 refers to the tree containing all the moves possible in a game. The game tree of MCTS refers to the tree specifically built during iterations of the MCTS algorithm. The game tree of MCTS stores previously used game states in order to develop itself. These trees are built during the execution of the algorithm and are not compiled before run time. The tree generated by the algorithm is then used to determine moves. This tree contains the moves that have been played along with an associated score. In the most basic MCTS, the scores are based on whether or not the move resulted in a win, loss, or tie during simulations. The most general Monte Carlo Tree Search follows these 4 general steps: selection, expansion, simulation, and back propagation.

Figure 2.4.1: An overview of the various steps in Monte Carlo Tree Search [Browne et al., 2012]

### 2.4.3 Selection

The selection step of the algorithm chooses a particular node or child to evaluate. The choice in this step is dictated by a selection policy which governs how the algorithm will traverse the tree. The details of the selection policy vary depending on the implementation of the algorithm.

### 2.4.4 Expansion

When the selection step reaches the edge of the constructed tree, the algorithm will reach the expansion step. The goal of the expansion step is to add an additional leaf to the tree so that it may be traversed. The new leaf will represent choices available at that particular moment. For example, in Chess the new leaf would represent a new move made by the simulation, such as moving a pawn forward or castling. The move to be stored in the new leaf can be chosen in a number of ways. In the most pure form of Monte Carlo, the action is chosen at random. This, however, has mixed results and more intelligent methods are often used.

### 2.4.5 Simulation

In the simulation step, the new leaf added to the tree is simulated according to some policy. The policy by which the node is simulated is not specified in

the structure of MCTS. This is left vague on purpose as MCTS is a highly general algorithm. In the case of general game playing, the simulation policy involves playing out the newly expanded node or leaf as a potential path in the game.

### 2.4.6 Back Propagation

In the back propagation step, the tree uses the simulation results to reinforce its nodes. By reinforcing specific nodes, certain moves and paths through the tree are strengthened. In this step, nodes simulated on the tree are modified. The use of the simulation results varies depending on the implementation the back propagation step. In the most simple Monte Carlo, the node scores are just averaged with the simulation results. Another common technique is to adjust the score of the nodes based on their distance to the goal state.[Finnsson, 2009]

## 2.5 Upper Confidence Bound Tree Search (UCT)

Upper Confidence Bound Tree Search (UCT) is an algorithm used to perform the selection step in Monte Carlo Tree Search. UCT was proposed by Koscis and Szepesvari in 2006 as a modification to generic Monte Carlo planning. Monte Carlo planning is an algorithm that uses Monte Carlo simulations to plan future moves from a specific state. Figure 2.5.1 (bellow), outlines a basic form of Monte Carlo planning. Monte Carlo planning, is one of the most basic forms of Monte Carlo used in general game players.

Figure 2.5.1: Generic Monte Carlo planning algorithm
[Kocsis and Szepesvri, 2006]

UCT is favored because it effectively balances the trade-offs between exploiting an already strong move and exploring new moves. UCT primarily concerns itself with addressing the select action step of Monte Carlo planning. In figure 2.5.1, this is the bestAction() function. In the most generic Monte Carlo Planning algorithms, actions are selected uniformly from the available actions. UCT improves upon this by assigning every visited node a payoff specified by the following formula:

- Where $Q_t(s, a, d) + c_{N_a,d}(t), N_s, a, d(t)$.
- $c$ = a bias constant
- $a$ = an action in the case of general game playing a move
- $s$ = a state or for our purposes a game state
- $d$ = depth or how many moves or states preceded the current action
- $t$ = time it took to reach this state
- $Q_t(s, a, d)$ is the estimated value of action $a$ in state $s$ at depth $d$ and time $t$
- $N_s, d(t)$ is the number of times state $s$ has been visited up to time $t$ at depth $d$
- $N_s, a, d(t)$ is the number of times action $a$ was selected when $s$ has been visited, up to time $t$ at depth $d^3$.

Figure 2.5.1 [Kocsis and Szepesvri, 2006]

Figure 2.5.1 outlines the formula by which UCT decides what move to select. The first component, $Q_t(s, a, d)$ judges the move based on its estimated value within the player's game tree. This value is then added to $c_{N_s,d}(t)$ which is the number of times the state has been visited multiplied by a bias constant. This bias constant determines how heavily the player will value moves that it has already visited. Tweaking this constant will alter the player's favoring of already visited moves. In the case where the player needs to be more exploitative we can increase the constant. In the case where exploration is more valuable the constant can be decreased. The last component, $N_s, a, d(t)$ factors in how many times a has been selected when it has been visited. This component of the formula allows for the player to evaluate not only how

many times a node has been visited but how many times it has actually been used. This allows for the player to consider moves it has previously used more effectively.

UCT is a significant improvement over other Monte Carlo planning algorithms.[Kocsis and Szepe In tests, it showed a consistently higher performance than generic Monte Carlo. It also demonstrated superior performance to minimax Monte Carlo, another offshoot of Monte Carlo that has better performances than generic Monte Carlo.[Kocsis and Szepesvri, 2006]

## 2.6 Roulette Wheel Selection

I am using Roulette Wheel Selection as a possible alternative to UCT in the selection step in Monte Carlo Tree Search (MCTS). Roulette wheel selection is a probabilistic selection algorithm. A probabilistic selection algorithm chooses an action based on probabilities associated with the action. This is different from UCT which is a deterministic algorithm. UCT selects its move based on the highest score it can find according its evaluation function. Roulette wheel selection and other probabilistic algorithms are not guaranteed to pick the best move. In roulette wheel selection, better moves have a higher probability of being selected.

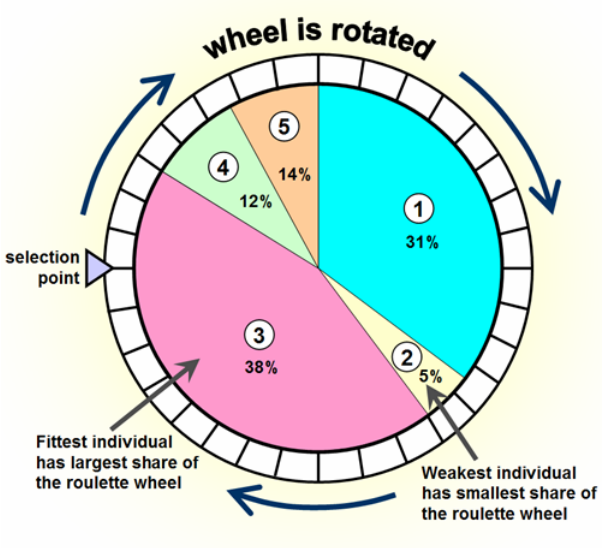


Figure 2.6.1 [rou, ]

In figure 2.6.1 we can see that the best option, number three in the figure, is the largest portion of the "roulette wheel". In contrast option two is given the smallest portion of the roulette wheel. The selection point is randomly determined each time a roulette selection is performed. I chose to investigate this algorithm because it provides a different take on balancing the exploitation of strong moves versus exploration of new moves.

Roulette wheel selection has three components. The first, is setting up the "roulette wheel" by getting the total score of the possible moves. Second, we compute how large each move's section will be. This is accomplished by taking an individual's score and computing its percent of the total score. In the third step, the algorithm generates a random value. This random value is then used to select the move.

**Algorithm 1** Basic Roulette Wheel Selection

```
Compute the total score from all scores
for score in scores do
    scorePercent = the percent score is from total score
end for
MoveSelector = Randomly generated number between 0 and 1.0
Accumulator = 0
for scorePercent in scorePercents do
    Accumulator += scorePercent
    if Accumulator ≥ MoveSelector then
        return the move associated with the current scorePercent
    end if
end for
Note: Each score is linked to a move along with each scorePercent
```

Algorithm 1 presents psuedocode of a basic roulette wheel selection implementation. The most critical component of the psuedocode that is not discussed above is how the move selected. Algorithm 1 demonstrates that the move selection is performed by creating an a accumulator variable and adding each of the score's percentages until it is greater than or equal to the move selection variable. When the accumulator is greater than or equal to the move selection variable, the algorithm returns the move associated with the percentage. Algorithm 1 presents a basic roulette wheel selection but there are certain weaknesses in the basic form that compel the development of a more advanced version.

## 2.7 Roulette Wheel Game Player

The roulette wheel selection I implemented and used in my game player contains additional features that alleviate some weaknesses in a basic roulette wheel selection. Roulette wheel selection's strength and weakness lies in its probabilistic decision making. Probabilistic decision making is theoretically where roulette wheel selection is capable balancing exploration and exploitation. However, sometimes probabilistically selecting a move is problematic in the case of situations where a one move loss and a one move win are present.

Checking for one move wins and one move losses is a simple process. One move wins may be checked for by looping through each of the current moves and examining the game state. Using functions present within game players we are able to check whether or not a game state is a win. If this is the case, we want the game player to pick this move regardless of the other moves. This is a valid choice because the games contain no hidden information. In the case of hidden information, no move is a guaranteed win. Checking for one move losses uses a similar process with an additional step.

One move losses may be accounted for by playing out each possible move for the player and checking each move available to the opponent. Using the same functions as above we can check if the opponent can make a move that results in a win for them. If this is the case, we do not want to use this move and set its score to 0. A move with a score of 0 cannot be picked in the roulette wheel selection portion of the algorithm.

**Algorithm 2** Implemented Roulette Wheel Selection

```
Prune the simulation for one move loss
if If all moves result in a loss then
  return any move
end if
Check for any one move wins
Compute the total score from all scores
for score in scores do
  scorePercent = the percent score is from total score
end for
MoveSelector = Randomly generated number between 0 and 1.0
Accumulator = 0
for scorePercent in scorePercents do
  Accumulator += scorePercent
  if Accumulator ≥ MoveSelector then
    return the move associated with the current scorePercent
  end if
end for
Note: Each score is linked to a move along with each scorePercent
```

Algorithm 2 is pseudocode of the more robust roulette wheel selection that was used in my game player. As stated above, it is the same as basic roulette wheel selection but prevents the use of one move losses and always uses one move wins. The step of checking if all moves are a one move loss and then returning any move is done to increase efficiency. In the case that the game player cannot win, there is no reason to set up all the roulette wheel selection infrastructure because it does not matter what move is chosen. Further, using this scheme can be problematic as no score percent will be capable of bringing the accumulator over the move selection variable.

### 2.7.1 GGP Base

My game player was developed using tools and functions provided by a project called The General Game Playing Base Package (GGP Base). GGP Base is an open source project hosted on Google code. Its goal is to develop and provide useful applications and tools for writing and testing general game players.[ggp, ] For this reason, I chose to develop my general game player within their framework and test my game player against their sample

test players. This includes functions to parse GDL, check for wins in game states, run game players, manage the interaction between game players, and set up matches between game players. GGP Base was a crucial component of the project because it provided all of the key underlying infrastructure necessary for general game playing.

The remaining code written for my game player handles running and scoring simulations for moves. The scores compiled for each move are passed on to algorithm 2 above for move selection. Algorithm 3, bellow, is the main function called in the Roulette Wheel Game Player. The Game Player runs as many simulations as time permits it to. The amount of time simulations may be run is determined by the match rules in which the game player is participating. The more simulations the player can run the more effective it will be at choosing moves. This algorithm uses a what is essentially a standard Monte Carlo approach to its simulations with a slight difference.

The difference between Roulette Wheel Game Player and standard Monte Carlo is in their scoring of situations. Standard Monte Carlo game players score moves based on whether they win, lose, or tie. My game player uses the same strategy but subtracts the depth of the move from the score. The depth is defined as how many moves it took for the simulation to reach a solution. By subtracting the depth of the simulation from the moves' score we prioritize moves that result in an earlier win.

The reason that my game player prioritizes earlier wins is because earlier wins prevent the game player from making mistakes. Long term wins require a higher degree of planning which Roulette Wheel Game Player does not yet have. Further from the perspective of playing games, winning faster is better because it gives your opponent less time to win the game.

**Algorithm 3** Roulette Wheel Game Player

```
Get all available moves
if Only have one available move then
  return only available move
end if
for move in moves do
  if Simulation time limit reached then
    break from for loop
  end if
  randomly simulate each move until a terminal state is reached
  compute the score of a simulation (100 for win, 50 for tie, and 0 for loss)

  subtract the depth of the move from its score
  store the move and its score in possibleMoves
end for
run a roulette selection on possibleMoves
return the move selected from a roulette selection
Note: the depth may at most half the score generated from the simulation
```

## 2.8 Games Used In Tests

For the testing of my game player I used the games Tic-Tac-Toe and Nim. These games were chosen because their winning strategies are well understood. This is beneficial because it is easy to judge what an optimal move is which makes analyzing a game players' performance easier. In addition, these games are simple to play making them appropriate for the initial testing of a new general game player.

### 2.8.1 Tic-Tac-Toe

The Tic-Tac-Toe used in my research is the same version as the commonly played game. In Tic-Tac-Toe two players, X and O, take turns making moves in a 3x3 grid. The winner is determined by whichever player can get three of their marks three in a horizontal, vertical, or diagonal row.

### 2.8.2 Nim

Nim is a two player game played with traditionally three heaps of objects; however, the implementation I used had four heaps of objects. The difference in heaps does not alter game play. The players alternate taking any number of objects from one of the heaps. The goal of the game is to be the last player to remove an object.

## 2.9 Existing Game Players

### 2.9.1 Cadiaplayer

Cadiaplayer is one of the strongest simulation game players that has been developed. It has won the annual general game player competition at Stanford numerous times.[Finnsson, 2009] Cadiaplayer uses a MCTS strategy and uses UCT as its selection strategy. Further, it uses a number of simulation control techniques that allow it to more effectively simulate relevant moves. These techniques make it capable of playing many complex games at a higher level. Cadiaplayer was the primary motivator in studying the MCTS strategy.

### 2.9.2 Fluxplayer

Fluxplayer is a heuristic/knowledge based general game player. [Haufe et al., 2011] Fluxplayer derives knowledge by using GDL parsing techniques. Fluxplayer then uses the knowledge it has parsed about the game as the basis for the development of its heuristics. Fluxplayer's heuristics and many heuristic game players develop their heuristics and run time. Fluxplayer provided insight into how to develop an effective heuristic game player.

# 3 Methodology

I compared Roulette Wheel Game Player to three different game players. Each of the three used different strategies.

Roulette Wheel Game Player played at least 75 matches of tic-tac-toe and nim against each of the game players described bellow. Roulette Wheel Game player played second in the initial 50 simulations for both tic-tac-toe and nim. Going second in both tic-tac-toe and nim is a disadvantage and allowed for the performance of the game player to be better measured. In

the remaining 25 games Roulette Wheel Game Player played first in order to verify the results of the previous set of simulations. If Roulette Wheel Game Player had overall performed worse going first, its quality of performance would require further examination.

The matches between the game players were conducted using GGP Base's game player server tools. Further, the GDL specifications of the games Tic-Tac-Toe and Nim were from GGP Base. Every game was run using the default time settings in GGP Base.

## 3.1 Opponents

### 3.1.1 Random

The first player picks moves at random. This is a rather ineffective strategy but is capable of what I called "one move brilliance" in that due to its move selection methodology periodically makes strong moves. This was used to check if my game player was superior to a consistently weak strategy and to test if it could react against immediately threatening moves.

### 3.1.2 Pure/Simple Monte Carlo

The second game player used a pure or simple monte carlo simulation approach. This player uses monte carlo simulations to generate aggregate scores for each available move and picks the move with the highest score. This player was used to check if the roulette wheel selection modifications to Monte Carlo yielded any significant improvement.

### 3.1.3 Search Light

The last game player I compared Roulette Wheel Game Player to used a basic search strategy. The player searched through available moves to find immediate wins or losses. In the case of immediate wins, search light picks it. In the case of an immediate loss, search light avoids it. Lastly, when evaluating a move than results in neither and immediate win or loss, it checks to see if the move results in a loss during the opponent's next turn. Thus, its strategy may be summarized as playing such that you do not lose in the foreseeable future. This type of game player is reactive and was used to test Roulette Wheel Player's ability to deal with players that attempted to stall its plans.

# 4 Results

Roulette Wheel Game Player had a good degree of success in its matches against other game players. Playaing as the 2nd player in Tic-Tac-Toe, Roulette had an average win rate of 60%. However, this percent is counting ties as a loss. Counting ties as a win we have a 84% win rate. If we then average this win percent, we have a win percent of 72%. In Nim, Roulette Wheel Game Player had a wood percentage of 71.6%. Roulette Wheel Game Player had an undefeated record when it went first in Tic-Tac-Toe and an almost undefeated record when it went first in Nim.

Table 4.1: Tic-Tac-Toe Results with Roulette Wheel Player as Second Player

| | |
|---|---|
| Random | 40 wins 7 ties 3 losses |
| Simple Monte Carlo | 40 wins 3 ties 9 losses |
| Search Light | 11 wins 29 ties 15 losses |

Table 4.2: Nim Results with Roulette Wheel Player as Second Player

| | |
|---|---|
| Random | 37/50 |
| Simple Monte Carlo | 39/50 |
| Search Light | 32/51 |

Table 4.3: Tic-Tac-Toe Results with Roulette Wheel Player as First Player

| | |
|---|---|
| Random | 23 wins 2 ties 0 loses |
| Simple Monte Carlo | 23 wins 1 tie 1 loss |
| Search Light | 7 wins 15 ties 3 losses |

Table 4.4: Nim Results with Roulette Wheel Player as First Player

| | |
|---|---|
| Random | 20/25 |
| Simple Monte Carlo | 22/25 |
| Search Light | 15/25 |

# 5 Discussion

## 5.1 Versus Random

Roulette Wheel Game Player performed well against Random in both tic-tac-toe and nim. The reasons for its strong performance were primary due to Roulette's Wheel Game Player's use of Monte Carlo simulations and its

ability to react to random. Roulette Wheel's use of Monte Carlo Simulations gave it a form of forward planning, which allowed for it to outplay Random. Further, Roulette Wheel's ability to detect moves that are immediately bad prevents Random from capitalizing on any good move it happens to make. In both tic-tac-toe and nim these traits a significant factor in Roulette Wheel Game Players' success.

### 5.1.1 Tic-Tac-Toe

As stated above, Roulette Wheel Game Player consistently beat Random due to its ability to plan ahead, while immediately removing one turn wins. In a game like Tic-Tac-Toe where one move wins are very prominent blocking out immediate wins proves to be an effective strategy. Particularly against Random who cannot generally employ higher levels of strategy, such as laying out a fork. A fork in tic-tac-toe and other games is a situation where a player has more than one winning move available. While it can occasionally generate these scenarios, it cannot do so consistently. Furthermore, Roulette Wheel Game Player's ability to plan ahead left it able to plan out win strategies something Random is lacking.

Roulette Wheel Player maintained a high level of performance against random going first and second. Going first did allow for Roulette Wheel Game Player to be undefeated as can see in table 4.3, with 23 wins and 2 ties. This level of performance wasn't completely unanticipated as going first in tic-tac-toe is an advantage. Particularly because Roulette Wheel Player has the ability to plan ahead and going first allows for its plans to be put into effect earlier in the game.

### 5.1.2 Nim

Roulette Player won the majority of games against Random in Nim. Going second it had a 74% win average as we can see in table 4.2 and an 80% win average as we see in table 4.4. This performance was influenced largely by the planning capabilities of Roulette Wheel Game Player. For reasons similar to tic-tac-toe, planning ahead demonstrates its value. Although, the lower performance in nim is possibly indicative of the problems of one move lookahead planning.

One move lookahead planning may have been insufficient to play nim. Nim is a slightly more complex game than tic-tac-toe and, therefore, may

have required a more complex planning mechanism to achieve higher performance.

Actively pruning the move results for one move wins and losses also met my expectations for contributing to Roulette Wheel Game Player's performance. I theorized that this would be an important factors as Random's move quality varied so heavily that planning for scenarios where it luckily chose the right move was a good decision.

Going first correlated with expected results as Roulette Wheel Player had a higher win percentage going first rather than second, see tables 4.2 and 4.4. Although, I did initially have a higher win percentage expectation when Roulette Wheel Player went first in nim. I expected Roulette Wheel Player to have the highest win percent against random in nim but as seen in table 4.4 it did not. This may have been caused by Random making a long term brilliant play to which Roulette Wheel player did not adapt.

## 5.2 Versus Pure Monte Carlo

Roulette Wheel Game Player performed well against the Monte Carlo game player. This seemed to stem mostly from its ability to play defensively and prune out moves that resulted in immediate losses. Both players have a system of planning but Roulette Wheel's is theoretically more flexible due to its probabilistic approach. My results were inconclusive as to whether or not a differing move selection mechanism played a part in Roulette Wheel Player's higher performance.

In theory, the higher degree of flexibility present in Roulette Wheel game player allows for it to better compensate for poor simulation quality. Both Roulette Wheel and Monte Carlo choose moves depending on the quality of their simulations. Thus, Roulette wheel is not locked into choosing any particular simulation which allows for it to partially mitigate this weakness. In addition, the simulations cannot account for the actual move the opponent will make. However, it was unclear if the larger degree of flexibility contributed directly to Roulette Wheel Player's victories.

### 5.2.1 Tic-Tac-Toe

In tic-tac-toe, Roulette Wheel Player had high win percentages. Going second resulted in a win percentage that was roughly 77%, seen on table 4.1, and going first yielded an 92% win rate, table 4.3. As stated above, the

improved performance was certainly contributed to by the reactive playing qualities in Roulette Wheel Player. In theory playing against Pure Monte Carlo should have demonstrated that Roulette wheel selection made for a better selection mechanism than simply choosing the highest aggregate score from simulations.

The reactive strength of Roulette Wheel Game Player made a large difference in tic-tac-toe. As stated above, tic-tac-toe features many one move win scenarios. Therefore, Roulette Wheel Player's ability to react to Pure Monte Carlo's plans enabled it to win the majority of games.

Similar to its games against Random, Roulette Wheel Game Player performed better when it went first. This correlated with expected outcomes as going first is an advantageous situation in tic-tac-toe.

### 5.2.2 Nim

Roulette Wheel Selection performed well against pure Monte Carlo. As seen in table 4.2 and table 4.4, Roulette Wheel Player had the highest win percent against Pure Monte Carlo both going first and second. Roulette Wheel Player's success occurred from reasons similar to its matches against Random Player. However, tests in nim may have shown the effectiveness of Roulette Wheel selection unlike tic-tac-toe. Due to the slightly longer length of nim, the difference in selection schemes may have been evident. The Monte Carlo that I tested against was the most basic and pure form of Monte Carlo which does not take care in evaluating a move selection beyond its simulation score.

Evidence that Roulette Wheel selection was effective began to emerge when playing against Monte Carlo simulations. By probabilistically picking moves, Roulette Wheel Game Player allowed for more flexibility which proved to be a strength. The Monte Carlo game player was heavily reliant on the quality of its simulations. However, these simulations are completely random, which is why it is vital to run a large amount of them. An additional problem is that the simulations eventually converge on several different moves that the opponent will make. The move the opponent makes is vital because it can dictate the next best move.

Neither game player has the capability to predict its opponent's next action. Therefore, the move we have simulated to be the best may not actually be the best due to our inability to accurately predict our opponent. Thus, the flexibility that emerges in a probabilistic system becomes an asset against Monte Carlo. However, the probabilistic nature of Roulette Wheel

can also lead it to choose moves that are too weak.

The double edged nature of Roulette Wheel selection contributed to a many of Roulette Wheel Game Players' losses. Although if Roulette Wheel game player chooses a weaker move early in the game it can often recover due to its overall tendency to pick stronger moves.

As expected Roulette Wheel Player won more games when going first. As we can see in table 4.4, it had an 88% win rate going first in nim. Similar to tic-tac-toe going first allowed for Roulette Wheel Player to enact its plans earlier.

## 5.3 Versus Search Light

### 5.3.1 Tic-Tac-Toe

Search Light was the only game player where Roulette Wheel Game Player was not able to win a larger portion of games, going second. This is likely due to Search Light's more defensive strategies which allow for it to block moves that result in a loss. This diminishes some of the effectiveness of Roulette Wheel Game Player because its selection strategies are designed to favor shorter term wins.

Roulette Wheel Game Player is not always capable of producing fork style wins. Fork style wins involve creating a situation where there are two winning moves for the player. It is possible for Roulette Wheel Game Player to derive these solutions but it is not guaranteed to. This is an important strategy in a game like Tic-Tac-Toe where the first player is heavily favored to win. Going first, Roulette Wheel Player was able to win more games than Search Light but still tied the majority of the games. This result is once again due to the strength of going first.

As table 4.1 and table 4.3 show the majority of games between Roulette Wheel player resulted in a tie. This was caused by the ability of both players to react to each others' moves. Especially because neither Roulette Wheel Player nor Search Light consistently employ forked wins to try and win they oftentimes block each others' moves until a stalemate is reached.

### 5.3.2 Nim

Search Light proved to be the player that most challenged Roulette Wheel selection when playing Nim. This may have been caused by Search Light's

more defensive strategies. It focuses its capabilities on blocking one move losses to the best of its ability and acting on one move wins, this scheme in Roulette Wheel Game Player was actually inspired by this. By blocking one move wins, Search Light can resist Roulette Wheel Game Players' more rudimentary planning.

As none of the other game players were concerned with defensive actions, Roulette Wheel Selection was able to defeat them because of its ability to play defensively. However, against Search Light its advantage was somewhat diminished. On the other hand, its win rate against Search Light demonstrates again that a planning strategy, even a rudimentary one, has positive effects on a game player's performance.

Search Light does not use any sort of planning, it just merely looks for moves that are immediately good or bad. Roulette Wheel Selection was able to win the majority of games by planning ahead. Further, its increased flexibility prevented it from falling into the trap of relying too heavily on the shortest victories. This was beneficial because Search Light tends to handle these very well.

An interesting occurrence occurred during the nim simulations against Search Light. Mainly, that it actually performed worse going first than going second. We can see on table 4.2 that against Search Light it averaged a 62.7% win rate going second and going first it averaged only a 60% win rate. There are two likely explanations for the occurrence. The first, is that going first proved to be worse for Roulette Whee Player in nim because it gave something for Search Light to react to, which is the strength of the player. Although, this is unlikely because generally speaking nim games are not ended within the first 2 moves. Thus this unexpected result was more likely caused by the smaller simulation set for going first. If we were to replay the simulations in the same amount or larger we would likely see improved results with Roulette Wheel Player performing better going first.

# 6 Future Work

Roulette Wheel Game Player shows promise in its results but its method of selection could be further fine tuned. This improvement would largely focus on altering the selection algorithm so that it did not pick moves with a simulated score lower than a threshold value. This will allow for the game player to prune out additional moves in its selection process. A simple scheme

would be to not choose any move with a score less than 50. In the current implementation, 50 is representative of a tie; therefore, by picking moves with only a score of 50 or higher we are hoping to theoretically allow our game player to play for a tie at worst.

After improving the selection algorithm it would then be appropriate to test Roulette Wheel Game player using more sophisticated games. This would allow us to gain further insight into its performance.

If the improvements to the selection algorithm for Roulette Wheel Game Player showed improvements. We could then begin to integrate it into a full Monte Carlo Tree Search system. This would allow for us to directly test its use as move selection algorithm in a scheme that is used in strong general game players. Further, this would allow for us to see the viability of Roulette Wheel selection if it did not perform well in more complex games. This might occur because Roulette Wheel Game Player mainly uses simplistic Monte Carlo schemes and does not contain infrastructure to handle more complex games.

# 7 Conclusion

In conclusion, Roulette Wheel game player demonstrated that Roulette Wheel selection was an effective strategy in general game playing when playing simple games. By using a probabilistic selection algorithm, Roulette Wheel game player was able to effectively balance the exploration of new moves and the exploitation of already strong moves. Even in the case where it did not win the majority of the time, it was able to bring the game to a tie consistently. These results seem to indicate that future work in refining the algorithm is worthwhile. Further, I would ideally be able to integrate Roulette Wheel selection into the more powerful and complete MCTS general game playing strategy. Based on these early results, a MCTS game player using Roulette Wheel selection may prove to be a competitive opponent against future general game players.

# References


[ggp, ] ggp-base - general game playing base package - google project hosting. http://code.google.com/p/ggp-base/.

[mon, ] Monte carlo simulation circle example. http://demonstrations.wolfram.com/MonteCarloEstimateForPi/HTMLImages/index.en/popup_2.jpg.

[rou, ] Roulette wheel selection.

[tic, a] Tic-tac-toe game state. http://i.istockimg.com/file_thumbview_approve/7472644/2/stock-photo-7472644-hand-drawn-tic-tac-toe-game.jpg.

[tic, b] Tic-tac-toe game tree. http://www.uh.edu/engines/545px-Tic-tac-toe-game-tree.svg.png.

[Browne et al., 2012] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

[Finnsson, 2009] Finnsson, H. (2009). Simulation-based general game playing. *PHD Thesis, Reykjavk University*.

[Genesereth et al., 2006] Genesereth, M., Hinrichs, T., and Love, N. (2006). *General game playing: Game description language specification*.

[Genesereth and Love, 2005] Genesereth, M. and Love, N. (2005). General game playing: Overview of the AAAI competition. *AI Magazine*, 26:6272.

[Haufe et al., 2011] Haufe, S., Michulke, D., Schiffel, S., and Thielscher, M. (2011). Knowledge-based general game playing. *KI - Knstliche Intelligenz*, 25(1):25–33.

[Kocsis and Szepesvri, 2006] Kocsis, L. and Szepesvri, C. (2006). Bandit based monte-carlo planning. *Machine Learning: ECML 2006*, page 282293.