Exploring Casino-Style Blackjack using the Monte Carlo Tree Search Method

by Jonathan Collins

Abstract

Monte Carlo Tree Search (MCTS) is an algorithm that enabled improvements in game playing to be made in games with large and often complex game states, such as Go. The algorithm is frequently used to study perfect information games, allowing the algorithm to be optimized to such games using variance pruning techniques. This paper explores the usefulness of MCTS in the hidden information game, blackjack, by comparing it to the performance of a well-known approach to solving the game space, basic strategy, when both strategies are applied with basic casino-style rules such as varying numbers of decks as well as automated shuffling. This paper finds that the MCTS agent was able to perform nearly as well as the basic strategy agents and may be able to match performance in future work.

**Background**

When playing games, human players are able to use intuition to quickly process information while a computer player must observe and consider all available moves before calculating its choice. The most common approach is to provide the machine with a set of heuristics, rules to govern decisions, and filter moves based on the rules chosen. This method does not scale well, requiring a factorial amount of resources in some games and systems. Another drawback is that this naive approach is not flexible, requiring heuristics hand-tailored for the game, and in some cases, even the moves available or the states of the game itself. These critical issues of the naive approach prompted games to adapt the Monte Carlo Tree Search, a search algorithm that builds trees based on move selection in order to more efficiently discover better lines of play, improving the quality of artificial intelligence in machine players.

Monte Carlo Tree Search

The Monte Carlo Tree Search (MCTS) algorithm uses some form of randomization to decide which move to make within a list of available moves [1]. At first, these moves may be decided entirely at random but can be weighted over time to influence how the algorithm decides which moves are more valuable[1]. Exploration occurs when the algorithm branches wide, attempting moves that may be suboptimal to explore lines of play which may turn out the optimal line. Exploitation occurs when the algorithm chooses actions that are currently known to be good lines of play. The MCTS algorithm attempts to balance exploration and exploitation so that the computer player can consider a wide variety of moves and determine which ones are the best actions to take in a given situation [1].

The MCTS algorithm first considers all available actions then chooses at random one of these actions using an algorithm that balances exploration and exploitation. Then, the player makes that move and a copy of the game state is made. The copy is then played out in some number of simulations until the game ends in a win, a loss, or a tie and then determines a value as to how much this terminating state is worth [1]. The copy is then traveled in reverse, propagating backwards by rewarding the actions taken along the way with the determined value [2]. Eventually this rewinds to the original game state, where the next action will be determined, only each subsequent action may be weighted based on the success during the simulations. Tree branches that report more frequent wins are more likely to be taken, while tree branches with high amounts of losses are less likely to be taken. One weakness of this algorithm is that the optimal set of moves may be found in a tree branch that first requires making suboptimal plays. The randomization for MCTS can be truly naive, picking at random every time, or more commonly uses another algorithm called Upper Confidence Tree (UCT), which is an algorithm that balances exploration and exploitation during randomization so that the MCTS can attempt to find optimal moves, even in suboptimal branches[2].

The MCTS algorithm has been used to improve game playing AI for perfect information games (games where all information is known to all players) such as Go [2], real time games such as Ms. Pacman [3], and has allowed research for imperfect games (games with hidden or derived information) such as Texas Hold'em poker [4] and Magic: The Gathering [5] to take a new direction in AI advancement. For Chess and Go, the MCTS allowed AI players to challenge, and in some cases exceed, expert human player skill, especially when the MCTS algorithm combined expert heuristics to filter known lines of play. For Ms. Pacman, the MCTS would allow an AI controlled player to learn and adapt to ghosts that have perfect game information by using the algorithm to engage online learning (learning based on constantly updating game states) allowing the AI controlled player to adapt to strategies for each level of skill difficulty[3]. In some Texas Hold'em variants, the MCTS algorithm was able to learn to match efficiency with other known algorithms. While MCTS did not necessarily play as effective as the other algorithms against an individual opponent, it was the only online learner that played at comparable levels of skill and research may be expanded such that the Monte Carlo learner may adapt to a table of players as opposed to one opponent, with the goal of being able to adapt play style to be able to learn and play competitively at any table [4]. In the case of Magic: The Gathering, the game itself involves deep lines of play based largely on both game states as well as large amounts of hidden and derived information. MCTS has been able to learn the basics of combat strategy within the game, a task that's challenging to accomplish without perfect information [5]. The MCTS algorithm is easily adaptable across many different styles of games and is able to model information in perfect and imperfect games, offline and online learners, as well as real-time and sequential games, making the algorithm a natural research topic for games and systems.

There are multiple ways to implement the Monte Carlo pattern but all of them share a general flow [6]. First, a copy of the current game state is made and then used as the root state for decisions. A root node is made to reflect this state, and then populates this new node with a list or array of available moves that have not yet been considered. A move is picked by some randomization, pointing back to the move-node of the prior move, and then the first copy makes a second copy of the game state, makes the selected move (if it's a leaf node) and plays the game out until the game would terminate[3]. Once the game terminates, it returns a reward value based on the outcome [6][3]. For most games, the basic reward is 1.0 for a win, 0.5 for a draw, and 0.0 for a loss, but can be changed as needed. Once this reward is determined, the algorithm then increments two values in the node, one that keeps track of the total times the games has visited, and one that keeps track of the reward value[3]. For example, a node visited would have its total visits increased by 1, regardless of outcomes, and the reward value can be increased by 0/.5/1.0 based on outcome. Then the algorithm recurses back to the move-node that lead to the chosen leaf node, and updates both visited and rewards like the leaf node did. MCTS will recurse back to the leaf node, updating the original sequence of actions [3], then dispose of the second copy. This process will repeat for some number of iterations, representing a number of simulations [6][3]. Once the simulations have concluded then first copy will evaluate all of its original list of available moves and chose the one with the highest win percentage. It then returns this move to the original game state, which then makes the chosen move, repeating the entire process until the actual game is completed [6].

While the basic algorithm is simple in concept, added complexity can yield improvements in performance, both in terms of speed and accuracy. The Upper Confidence Bounds algorithm, UCB1, helps improve the randomization of choosing moves in MCTS. UCB1 moves MCTS away from choosing purely at random (the naive approach) and towards randomization that weights the best known move more heavily than others [7]. UCB1 chooses the best known option roughly 50% of the time, while choosing other options, evenly, the remaining 50% of the time [7]. This allows for a better balance between exploration and exploitation. Further improving upon UCB1 is the UCB\_Tuned algorithm, which offers two changes of note [3]. The first change is that UCB\_Tuned doesn't use just the win ratio like UCB1, but combines it with an additional component that allows "explored" moves that are yielding winning sequences back to back to rise quicker to be chosen for exploitation, while moves that lose more frequently rise slower[3][7]. Second, this modification allows for a second component to be added to impact node decisions. These modifications range from tree pruning [8] to using heuristics or "Best-of-N" modifications [9].

Prior to MCTS, the game Hex used a pruning algorithm for creating minimax trees, Alpha-Beta, that can outperform MCTS but has a drawback in that it requires strong evaluation functions of the game in question in order to be efficient, and even then, it's not better in all cases[8]. MCTS is an algorithm that scales with "Best-of-N" heuristics, which aim to eliminate illegal moves and game states from being considered, or outright bad moves [9]. This domain knowledge is not required for the MCTS to work, making MCTS a flexible algorithm [8], but providing such heuristics allows MCTS to use hardware more efficiently and a beneficial side effect of "Best-of-N" heuristics is that they scale well with hardware power [9]. As hardware grows more powerful, further research could show MCTS to be the more powerful game AI algorithm over the traditional Alpha-Beta pruning Comparing MCTS to Alpha-Beta has shown that both game-move heuristics and "Best-of-N" heuristics have improved the rate at which MCTS beats Alpha-Beta, from 19% to 28% in 3 second tests; from 29% to 37% in 10 second tests; and from 22% to 32% in 30 second tests [9]. Monte Carlo also improves against Alpha-Beta with higher time limit rounds, up to a critical point where MCTS can but does not necessarily yield better accuracy against itself.

Parallel processing has also been used to improve MCTS, especially when compared to Alpha-Beta pruning. MCTS won and increased 35% of the time against Alpha-Beta on a single core processor, but increasingly won with more cores, eventually reaching a 75% increased win rate on 8 cores of processing power [9]. This is likely due to how each algorithm uses mutex locks. Alpha-Beta must engage locks at nearly every step of the algorithm, which can lead to cores waiting in queue to obtain a lock, while MCTS only needs to engage locks on a node by node basis, and could potentially be growing different areas of the tree at once due to randomization [9]. For example, if the MCTS algorithm is evaluating "Move A" on one thread, then a second thread is evaluating "Move B", then it's possible that neither thread is engaging in conflicting areas of the tree and can likely just grab locks immediately with little to no competition. This variation is also referred to as "Ensemble UTC" when using concurrency to search several independent trees at once [5]. This yields yet another area of future research because parallel computing is still in its infancy, and efficiency in parallel computing times should only serve to benefit the parallel MCTS. Parallelization and "Best-of-N" improvements both scale well with hardware increases and are both being researched for improvement in efficiency and breakthroughs on any of these fronts could propel the MCTS algorithm's efficiency and so further research into any of these areas would likely benefit MCTS as well[5].

Other modifications to the MCTS include treating imperfect information games as perfect information games when modeling as the method can yield results similar to expectations [5]. MCTS agents playing perfect information variants of imperfect information games have been able to learn the games with great efficiency as well as to expand research into multiplayer variations [5]. One side effect to this, however, is that different states may hold different moves necessary to win or gain advantage in situations where the correct move may exist differently between a perfect information and imperfect information model [5]. When MCTS was applied to Ms. Pacman in real time, it was discovered that different decisions lead to different game states, so different MCTS agents may play worse against stronger agents, or better against weaker agents than would be expected of an identical game state, based on how the game arrived at the state[3]. Research of MTCS in the card game Magic: The Gathering (MTG) has shown promising results for imperfect information games. Since MTG uses a customizable deck of cards, players make decisions based off both public information (the primary game state) as well as hidden information (player hands), and because the game rules have great depth and actions can be made during multiple points within a turn in some cases, and some actions may even be taken during opponent's turns, MCTS research on this game may be able to push MCTS harder than the game Go, which as combinatorically large tree branching[5].

Many of these modifications were tested with the MTG MCTS Agents and compared with rules-based agents created to test against human players. Using a strong-rules player, reinforced with heuristics given by competitive player knowledge, a weak-rules player, which used the basic heuristics, and a random player, the strong rules player would compete against humans playing at a competitive level with win rates of 37.5% to 46.6%, playing second and first respectively [5]. The Monte Carlo agent was able to best the strong-rules agent in the majority of games played, using a basic representation of game interaction. Even a weak-rules variant MCTS agent was able to outperform a strong-rules player [5]. Once again, larger simulations do not necessarily yield the best move, but definitely helps up to some critical point and naive randomization proved to be very weak[5]. Poker is another hidden information game that used counterfactual regret minimization (CFR), a method to calculate strategy profiles to use MCTS to produce strong players, despite hidden information [5].

Genetic programming has also been researched for MCTS improvements. Genetic programming is any algorithm that develops solutions to problems via natural selection. Genetic programming works by randomly generating child trees which attempt to solve the task at hand [10]. Then it breeds the best children, using them as parents for the next generation of children, repeating the cycle some number of times until a time limit or number of simulations has been reached [10]. The MCTS algorithm was combined with a genetic programming algorithm in Ms. Pacman, providing a real time environment for these algorithms to explore. Results showed that choosing the training environment (a given stage of the game) largely impacts the outcome of genetic programming, especially when the real game environment is used [10]. Ms. Pacman requires a game tree that accounts for simultaneous moves and must consider her own moves against both the best moves of the enemy ghosts as well as the moves the ghosts actually take. Experiments led to an agent that was created by using genetic programming on various subtrees and then recombining these trees, proving superior and quicker than the standard genetic programming algorithms [10]. The MCTS in the experiments used a split reward system that gave half of the reward based on the game score earned during the current simulation and the other half awarded based on whether or not the pacman agent survived the simulation of Ms. Pacman [10]. Originally the experiments performed playouts with a random agent that was guided loosely by heuristics. The random agent was replaced with a genetic agent that would evolve after each playout and then readjust its play behavior based on game states. As Ms. Pacman plays in real time, the MCTS needs to be able to perform numerous simulations in a smaller than usual time frame, often 40ms to 60 ms, and as a result the experiments used a maximum tree depth to reduce calculation times. Shorter simulations also showed to be more accurate, as did larger numbers of simulations, which works well because the goal of lower calculation times naturally achieves shorter and more numerous simulations in some cases. Ideally the experiment should strive for 25,000 evaluations per round, but due to various inefficiencies, it yielded only 5000 evaluations, leaving significant room for future improvement [10].

The MCTS algorithm builds large trees in order to determine best moves and since tree pruning has been successful with its predecessor researchers turned to decaying simulation strategies. Old information is normally kept between moves, allowing a given strategy to be reinforced, but strong move information may become outdated as the game progresses [11]. Strategies that track statistics data over multiple moves may benefit from decay more than UTC algorithms. One decay strategy, N-gram selection, tracks move sequences in addition to single move. The N-gram selection technique, or NST, keeps track of sequences between multiple players and averages the rewards earned by individual nodes to score the sequences, which are tracked by their respective player agents [11]. One decay factor is applied after an actual move is made, while a second factor is applied after an arbitrary number of rollout simulations are tried, while a third decay strategy can be applied after each simulation and affects only sequences that were played in the current round[11]. Results showed that games where move quality depends on game phases and states are more likely to improve with decay factors. In most cases, a decay value of 0, which resets moves and reward values between simulations, showed similar or better performance improvement than using no decay factor at all[11]. Single player games benefit little, if at all, from decay strategies [11]. Decay strategies worked best when selection of the best strategy relies on recent move history and game states [11].

The UTC methods commonly found in MCTS can also be improved through tree pruning. Three UCT pruning methods, two of which are domain independent, requiring no formal knowledge of what the algorithm is being applied to, as well as one domain dependent method, which requires more detailed information unique to the application[12]. Absolute pruning trims decision nodes that are not the most traveled, resulting in performance roughly equal to the original UTC method [12]. Relative pruning trims all decision nodes on a branch that result in only losses, as long as one node in the branch has some number of wins; this method performs mildly better [12]. Relative pruning, however, cannot be applied in all experiments. Lastly, territorial pruning assigns values to board positions based on which player is most likely to benefit from the move. This approach improved MCTS in Go, but requires knowledge of the game and may be challenging to implement in other applications [12].

While the MCTS offers ways to improve the quality of the algorithm, it's equally important to consider future applications of what the algorithm can achieve in order to guide future work.

For example, MCTS agents can play Go at an expert level on a 9x9 board and can also play at a seasoned level on a 19x19 board, but the best Go research programs are weak in local move situations [13]. Although attempts to address this with parallel computing offer improvement, this improvement doesn't scale well with games like Go, but researchers predict that even one or two major breakthroughs in algorithm research could offer significant improvements in general game playing [13]. These breakthroughs will likely appear from the MCTS being combined with other algorithms. If MCTS can improve the quality of local searches then the MCTS can improve in both speed and accuracy [13]. The publicity of the MCTS breakthrough in the Go problem has promoted funding in the field of power-plant management [13]. MCTS may be extended to allow power-plants to manage energy more efficiently, sometimes with ecological and economic benefits [13] and such research may yet expand the number of fields that are able to apply the MCTS to improve systems.

Computer games could create dynamic stories for characters that change on multiple playthroughs and these games could then contain dynamic content books [14]. Dynamic storytelling could use MCTS to account for multiple actors taking a variety of actions across many locations and formulate believable stories [14]. In some cases, databases have been used to help agents generate story plots dynamically [14]. Some research has allowed dynamic storytelling for individual characters that still fits within the larger story domain. Interactive narrative generation has been used in a framework that acts as a tool for authors to modify both dynamic and automatic stories [14]. Experiments are conducted with a "believability" metric that rewards action paths that make sense, such as detectives being more likely to arrest a murderer after observing murdered actors and obtaining clues about the scenes and uses this metric within a story domain consisting of actors, items, and places [14]. Items give actors additional actions to take, while actors can either pickup items or move to new places as default actions [14]. Actions are marked as more believable or less believable based on secondary criteria, such as an angry actor being more likely to attack another actor, while a calm actor is much less likely to commit the same action [14]. MCTS first adds moves to a tree as usual except then evaluates the believability factor of each move as that move is added to the search tree [14]. Rollouts then create a series of random actions until all story goals are met, which will often be believable unless the only action set that satisfies all goals can only be met with a set of unbelievable actions. A history table weights the actions of the initial story state and then tracks these weights to improve early rollouts and combines heuristics to improve the believability factor[14]. Future work in this area may include exploring other believability metrics or to allow an author the ability to control elements such as pacing and timing of the climax of the story [14].

Operating room simulators used to train operating room staff have been improved by using MCTS to improve the AI player that controls non-player characters so that the AI player plays more accurately like a human player [15]. Experiments allowed the MCTS algorithm to only evaluate authorized actions and evaluation of the final game states [15]. Nodes used during experiments represent the states of the environment, with the actions taken being represented by the travel between nodes [15]. Decaying rewards encourage lines of play that reward the shortest plans of action. Mandatory nodes have fixed reward values and do not get awarded during back propagation, allowing the MCTS to frequently pick required nodes [15]. Non-player characters are also given instructions anytime they are awaiting commands, and these combined restrictions have even allowed an AI to correct human players that attempt to deviate from reasonable and expected lines of play [15].

Paraphrase generation is any algorithm that takes a given source sentence and converts this sentence into a new one with similar meaning [16]. In general, the paraphrase generation process is similar to the language translation problem, where the source and target language is the same [16]. Usually this translation requires decoding, although paraphrase generation may be able to use the MCTS to avoid this [16]. Paraphrase generation can include automatic summary, aiming to create the shortest paraphrase possible [16]. A scoring function is applied to transformation actions, where states are modeled as a sentence with a set of possible actions [16]. These states can then be added to the MCTS algorithm, although currently the algorithm generates ill-formed sentences approximately 37% of the time. Adding a linguistic knowledge-based analyzer could further reduce the rate of awkward paraphrases [16], improving the utility and capabilities of paraphrase generators.

In general, the Monte Carlo Tree Search algorithm is flexible and can assist in modeling a variety of problems, including both perfect and hidden information systems. This paper aims to expand the knowledge of the MCTS algorithm in casino-style Blackjack, a hidden information game with more depth than traditional Blackjack.

Blackjack Rules

Blackjack is effectively a two player, hidden-information game between a player and a dealer. While multiple players may play at a table, players do not compete or interact among one another thus the game's interactions are directly between that of the player and the dealer. The goal of the game is for the player to create a hand that beats the dealer's hand while not busting her own hand by going over 21[17].

A round of Blackjack is relatively straightforward to play. First, a player places a bet and then a dealer deals that player two face up cards, and deals herself one face up card as well as one face down card. Then, the player has the ability to act to modify her hand. Cards are valued by their face value, with kings, queens, and jacks being worth 10, and aces being worth 1 or 11. Aces are treated as 11s unless the hand value would exceed 21, in which case they are automatically worth 1. It's possible to have the first ace in a hand count as 11, with subsequent aces counting as 1 each.

In this experiment the player will have the ability to "hit", "stay", "double down", or "split", based on the rules of the game for that hand. When a player hits, she is dealt another face up card and a player may hit as many times as she desires, or until the hand busts by going over 21[17]. When a player stays, she ends her turn and awaits the dealer's actions to decide the round. A player may choose to double down if they have taken no other actions in their initial hand for the round by placing a bet equal to the original bet. Once the subsequent bet is made, the player is then dealt her card but then her turn ends. This action allows the player an element of skill in order to increase her winnings. Lastly, a player may split a hand that contains two identical number or face cards by placing a subsequent bet equal to the initial bet. For example, if the player has a hand containing two kings, she may split the hand, while a hand of one king and one queen will not be split for this experiment, although occasionally some casinos allow for this modification. Splitting allows the player the potential to play two hands in an attempt to win additional money. The dealer will separate the two cards from the original hand into two separate hands, then the player makes decisions for her first hand, and then her second. For this experiment, the player cannot double down after splitting, nor can she split again, although rarely these options may be considered in some casinos.

Once the player has finished making actions for all of her hands in a turn, play then turns to the dealer, who will act according to the casino rules [17]. "Soft 17" dealers will hit on all hands with a total value lower than 17, and stay on 17 to 21[17]. This is the most common type of dealer, however, a second dealer was almost as common, "Hard 17." A Hard 17 dealer will hit on all values below 17, but if the dealer has a total hand value of 17 to 21 and the hand is counting an ace as an eleven instead of a one, the dealer must hit [17].

Once the dealer finishes her turn, all cards are revealed and the players then compare their hands. A player automatically loses any hand that busts. When a dealer busts, players that haven't busted automatically win. If the dealer's initial hand value is 21, the dealer has a blackjack and will automatically win unless the player also has a starting hand value of 21. A blackjack will beat all non-blackjack hands, even if the hand values are tied. If neither player has a blackjack and no players are busted, then the hand with the highest value wins, and at this point if there are any ties, the player will "push" the hand. When a player wins, they will receive an amount of money matching their bet plus their bet, while a player that pushes will only receive their bet. In the case of blackjack, players will win additional money. In some casinos, players may spend money to purchase insurance to help mitigate the risk of a dealer having 21, however, this is not modeled in the experiment because insurance is not a common enough option to apply generically.

**Previous Research**

MCTS has been applied to a basic version of blackjack where the player is capable of hitting and staying, but not doubling or splitting. Vaidyanathan found that the MCTS agent could almost play as well as an expert agent under these circumstances, and suggested that future research into the subject would include the ability to double and split [17]. The thesis used array matrices in order to track success with the hit/stay model [17]. This approach would allow the simulations to run more quickly, but also makes modeling the effects of doubling and splitting harder [17].

**Method**

Blackjack Simulations

The simulator will playout and record the results for some number of rounds of blackjack. In each round, if the player has enough money to cover the bet amount, the player will be dealt a hand of two cards while the dealer will be dealt once card representing their face up card with the dealer receiving their "face down" card immediately after a player acts.

Once the initial cards of the round are dealt, the player will first check to see what available moves she has, then makes a decision as to which move to make, makes the move, then her turn is over. The dealer receives her face down card, then goes through roughly the same steps as the player except that the dealer does not need to check available moves as dealers only ever have the default moves, hit and stay, available to them. The simulation will use a tree of depth one to model the actions in a hand as, apart from hitting and splitting, the stay and double actions will only use one layer of depth. To resolve the issue of subsequent decisions on hits and splits, the initial decision will be made using the MCTS, while subsequent decisions will hit until the hand busts or reaches 17 to 21, which models after the Soft 17 dealer.

When a player is dealt her initial hand in the round, the hand is assigned a value of 1. Splitting a hand will divvy the hand up into two new hands, subsequently splitting the hand value evenly between the newly split cards. When considering a split, it is possible to win both hand, lose both hands, or win and lose hands. Once the player and the dealer have finished play, the hands are compared and the player is awarded accordingly. Each winning hand will reward the player with a win rate for the round, equal to the amount of the winning hand. In the case of a split, it becomes possible to win .5 points if the player wins one hand but loses the other. This weight is used to reflect the nature of the value in splits, especially when a betting model is used. The blackjack simulation will gather information based on win rates and the money a player ends up with at the end of a simulation, and will compare a "random" player that acts as an inexperienced player, a "basic strategy" player which acts as an expert heuristics player and serves as the control, and a "Monte Carlo" player which will allow for testing of the MCTS algorithm to compare its performance to an expert player. A card counting player will not be used because the simulations will be operating under the assumption that a card shuffler is being used after every round, as this is becoming a large and growing trend in casinos to eliminate card counting, which cannot operate on such a system effectively.

After all rounds have been played in the simulation the results for the player's win rate percentage, ending bankroll of the player, and the time required for the simulation are recorded in text files. The simulation will be strictly between the player and the dealer for simplicity, although adding multiple players should have no effect on the results [17].

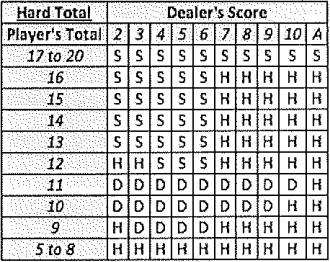
**Player Types**

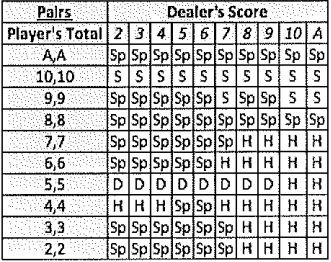
Random Agent

The random agent will select moves randomly from the moves available and then make them. This emulates a novice player who may not be familiar with the rules of the game, or any strategy used to improve her success.

Basic Strategy Agent

Basic strategy in blackjack is the culmination of trial and error blended with mathematical probabilities designed to solve the game on a hand by hand case [17]. This simulation will use the basic strategy listed in [17] that optimizes the agent against a Hard 17 dealer. Ideally this agent will play better against Hard 17 dealers, though the difference in play against a Soft 17 dealer is relatively minor at best. The following represents the basic strategy used in general as well as for splitting hands:



Figure 1

The above tables in Figure 1 list the player's hand on the left and compares it to the dealer's face up card listed on the top, and then chooses to either hit "H", stay "S", double "D", or split the hand "Sp" based on how the player's hand relates to the dealer's face up card. For example, if a player has a hand of 10 and 10, then the player should always stay as the chances of winning this one hand are often greater than if the hand would be split and risking two mediocre hands that may not both win. Alternatively, a player holding a hand with a pair of 9's would split unless the dealer's face up card was a 7, 10, or an A, in which case the dealer may have blackjack if her upcard is either a 10 or an A, and an upcard of 7 could represent a strong hand that's likely to beat both hands after a split.

Monte Carlo Agent

The MCTS agent makes decisions by simulating the current hand of blackjack some number of times and then selects the action that leads to more winning states. It does this by first making a copy of itself, the dealer, and the deck(s) of cards being used, as well as nodes for each of the HIT, STAY, DOUBLE, and SPLIT actions that are available in the hand. When an action is played, the Monte Carlo simulation will increase the count of the number of times the action has been selected, and if the action wins, will increase the win rate of that action equal to the value of all winning hands. In the case of splits, this means that winning two hands is superior to winning one hand, which is superior to winning zero hands.

A Monte Carlo sim first plays each available action exactly once and records the win rate and times played. Then, the simulation chooses between these actions randomly for the remainder of the Monte Carlo sim, using the UCB approach by weighting the winning action to be selected 50% of the time while the non-winning actions share evenly the remaining 50%. This allows a balance between retesting options that were not winning often early on, while allowing a winning action to be selected more provided it remains a winning action. After this is finished, the winning action node is selected and its action is returned back to the agent which will then select this action and play it against the dealer, and repeats this until the hand is completed.

**Research Questions**

1. How does the number of Monte Carlo simulations effect the win rate of the MCTS  
    agent?
2. Is there a number of MCTS sims among the four trial variables that is more efficient than others?
3. How does the number of rounds of blackjack impact the win rates of all agents?
4. Is there a number of blackjack rounds among the four trial variables that is more efficient than the others?
5. How do the agents' win rates compare to each other in the blackjack trials?
6. How do the ending bankrolls of the agents compare to one another assuming an efficient number of MCTS simulations as well as rounds of blackjack can be found when altering the flat bid amount?
7. How does the number of decks effect the ending bankroll of agents?

**Simulations**

RQ1 and RQ2

Experiments to discover the effects of win rates based on the number of Monte Carlo simulations requires isolation of other variables and as such the testing was performed under two conditions. The first condition was against a Soft 17 dealer with 1000 blackjack simulations and 1 deck. The second was against a Hard 17 dealer with 1000 blackjack simulations and 1 deck. This number of blackjack simulations represents a large enough sample size such that the data is accurate while the number of decks was selected as 1 to observe if there was much variance between Monte Carlo simulations. If there is a high variance then it becomes necessary to perform multiple trials with differing deck numbers as well as varying Monte Carlo simulations, while little to no variance would allow the experiment to later test the number of decks as an isolated and independent variable. Simulations would play against both dealer types in order to observe whether or not the MCTS agent played better against one over the other. This experiment would test these conditions with 50, 100, 500, and 1000 Monte Carlo simulations. One hundred data points were collected for each simulation, for a total of 100,000 rounds of blackjack played per data set.

Based on the data collected for this experiment there were no meaningfully significant changes in win rates versus either Hard or Soft dealers across the above values for Monte Carlo simulations. The minimum win rates across all trials showed no significant changes between the numbers of simulations nor between dealer types. The maximum win rate for both Soft and hard dealers was insignificant after accounting for outliers. This experiment shows that a large number of Monte Carlo simulations is not necessary in casino-style blackjack, and that since blackjack has a small number of decision points then using the 50 Monte Carlo simulations provides approximately equal accuracy to using 1000 but without the extra memory overhead.

RQ3, RQ4 and RQ5

Testing the influence that the number of blackjack rounds had on an agent's win rate was performed with the Monte Carlo agent using the 50 simulations model for 100 data points of blackjack simulations. The number of blackjack trials tested were 250, 500, 1000 and 2000 trials. These trials were performed with one deck to isolate the impact of blackjack trials on win rate. As there was no meaningfully significant variance between the Monte Carlo hard/soft agents, only the hard agent was used for the blackjack trials. During the trials, there were no significant differences between the Random hard/soft agents, so only the hard agent was used for comparisons. There were no significant differences between the Basic hard/soft agents with the exception that the soft agent possessed larger standard deviations at smaller numbers of trials but somewhat tighter deviations at larger numbers of trials. The differences in standard deviations is likely due to two outlier win rates observed during data collection, although they appear to have no significant impact on win rates.

Between all variations in the number of trials for a given agent, all agents followed similar patterns, namely that win rates were not impacted significantly based on the difference in number of trials. Simulations with 250 trials held drastically higher variance across all agents while that variance fell sharply at 500 trial simulations. Beyond 500 trials, however, the standard deviation would lower but contribute no further impact on the average win rate.

Random agents were able to attain approximately a 40.6% win rate. The Basic agents performed the best with an approximately 60.5% win rate, while the Monte Carlo agents were close behind with an approximate 58.5% win rate, a difference of around 2%. At 500+ trials however, the Monte Carlo method had a lower standard deviation suggesting that the agent would perform at this win rate more consistently compared to the Basic agent. It's important to note that for these simulations win rate refers to the total number of round wins, with split hand rewarding portions of a round's value. This metric is used to measure and compare the Basic strategy and Monte Carlo agents more readily, and these values will be higher than the average win rates across all hands as a whole.

RQ6

Flat bid simulations were observed using 500 blackjack trials and 50 Monte Carlo simulations as those were found to be the best test numbers for the variables based on the prior tests performed. Both the Monte Carlo Hard and Basic Hard were chosen for this because they played similarly enough to their soft counterparts, and these agents would start flat bid rounds with $10,000 and would always make the maximum bid available. The value of the flat bid was $5, $10, $50, and $100 amounts.

Among the flat bid variables, the Basic Hard agent performed better in almost all areas. In each comparison, the Basic Hard agent would have a higher average winnings than the Monte Carlo Hard agent did. Across the bid range, the Basic Hard agent would hold maximum winnings and spreads that were approximately linear based on the bid amount. This pattern of maximum winnings and spreads makes sense considering that the Basic strategy does not deviate play style based on the bid amount while it maintains a fairly consistent win rate percentage. Consistent play with scaling bid amounts makes sense that the maximum winnings would also scale accordingly. The Monte Carlo agent also held near-linear maximum winnings in proportion to the bid amount, however, differed in spread on the $50 and $100 simulations. The spread in this simulation represents the level of fluctuation in the simulation, with a tighter spread being more probable to arrive at the average. While the Basic Hard agent did have a higher spread than the Monte Carlo agent in these categories, the Basic agent still maintained higher average winnings. The Monte Carlo agent was less likely to lose a large amount of money, however, the Basic agent, while riskier due to larger variance, offers a higher potential payout.

RQ7

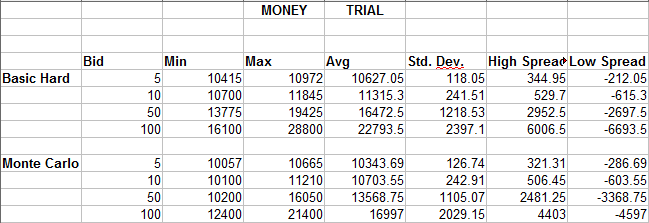
Simulations to discover the impact the number of decks have in casino blackjack on the agents were observed as an extension of the flat bid simulations, but using $100 as the flat bid value. Data was collected using 1, 2, 4, and 8 decks comparing the spreads and averages of the simulations using both the Basic Hard and Monte Carlo hard agents as was used in the flat bid trials.

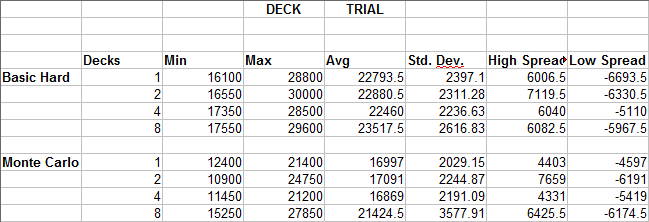
The number of decks held almost no impact on the Basic Hard agent with the exception of going from 4 to 8 decks, in which case the agent gained roughly 2.3% more average bankroll when winning. The Basic Hard agent outperformed the Monte Carlo Hard agent in each case, however, the Monte Carlo agent also had a notable 25.3% increase to winning bankroll average between 4 and 8 decks, but between 1 to 2 and 2 to 4 decks there was little growth.

**Conclusion and Future Work**

The Monte Carlo agent is able to win just under the frequency of basic strategy agents, although basic strategy agents performed better in both flat bid simulations as well as simulations to observe the influence of the number of decks used, these simulations demonstrates that an agent could perform fairly well when not provided an explicit set of instructions. Future research in applications for this algorithm would be that modifying or playing similar games to blackjack could yield improvements if the random sampling methods can be improved. One advantage the algorithm has is that if the simulation rules were to be changed into a new blackjack variant, the Monte Carlo algorithm should be able to adapt easily while it would be inconvenient to redefine a system similar to basic strategy. For current iterations of blackjack the basic strategy is more efficient in implementation than Monte Carlo tree search at this time, however, the numbers are close enough that the right vein of research could challenge this assessment.

Appendix





Works Cited

[1] C. B. Browne et al., "A Survey of Monte Carlo Tree Search Methods" in *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 2012* pp. 1-43

[2] F. Xie and Z. Liu, "Backpropagation Modification in Monte-Carlo Game Tree Search" presented at *Third International Symposium on Intelligent Information Technology Application*, 2009.

[3] S. Samothrakis, D. Robles, and S. Lucas*,*"Fast Approximate Max-n Monte Carlo Tree Search for Ms Pac-Man" in *IEEE TRANS. ON COMP. INTELL. AND AI IN GAMES*, 2011, pp. 142-154

[4] J. Heinrich and D. Silver, "Self-Play Monte-Carlo Tree Search in Computer Poker" in *Computer Poker and Imperfect Information: Papers from the AAAI-14 Workshop*

[5] P. Cowling*,* C. Ward*,* andE. Powley*,* "Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game *Magic: The Gathering" in IEEE TRANS. ON COMP. INTELLIGENCE AND AI IN GAMES, 2012. pp. 241-257.*

[6] T. Toyoda and Y. Kotani, "Monte Carlo Go using Previous Simulation Results" presented at *International Conference on Technologies and Applications of Artificial Intelligence*, 2010. pp. 182-186

[7] Y. Wang and S. Gelly, "Modifications of UCT and sequence-like simulations for Monte-Carlo Go" presented at *IEEE Symposium on Computational Intelligence and Games, 2007.* pp. 175-182

[8] B. Arneson, R. B. Hayward, and P. Henderson, "Monte Carlo Tree Search in *Hex*" in *IEEE Transactions On Computational Intelligence and AI in Games*, Vol. 2, No. 4, December 2010

[9] T. Jakl, "Arimaa challenge – comparission study of MCTS versus alpha-beta methods" M.S. Thesis, Dept. of Theoretical Comp. Sci. and Math. Logic., Charles Univ., Prague, Czech Republic, 2011

[10] Mandy J. W. Tak, Mark H. M. Winands*,*  and Yngvi Björnsson*,* "Decaying Simulation Strategies" in *IEEE Transactions On Computational Intelligence and AI in Games*, Vol. 6, NO. 4, December 2014

[11] A. M. Alhejali and S. M. Lucas, "Using Genetic Programming to Evolve Heuristics for a Monte Carlo Tree Search Ms Pac-Man Agent" at *Conference: Computational Intelligence in Games,* 2013

[12] Jing Huang, Zhiqing Liu, Benjie Lu and Feng Xiao, "Pruning in UCT Algorithm" at *International Conference on Technologies and Applications of Artificial Intelligence*, 2010

[13] Kirk L. Kroeker, "A New Benchmark for Artificial Intelligence" in *Comm. of theACM*, vol. 54, no. 8, August 2011, pp. 13-15.

[14] Bilal Kartal, John Koenig and Stephen J. Guy, "User-Driven Narrative Variation in Large Story Domains using Monte Carlo Tree Search" at *International Conference on Autonomous Agents and Multi-Agent Systems,* 2014

[15] Maxime Sanselone, Stéphane Sanchez, Cédric Sanza, David Panzoli and Yves Duthen, "Control of Non Player Characters in a Medical Learning Game with Monte Carlo Tree Search" at *Conference Companion on Genetic and Evolutionary Computation Companion,* 2014, *pp. 51-52*

[16] Jonathan Chevelu, Thomas Lavergne, Yves Lepage, and Thierry Moudenc, "Introduction of a new paraphrase generation tool based on Monte-Carlo sampling" at *ACL-IJCNLP Conference*, 2009

[17] Vaidyanathan, Atul. 2014. "Monte Carlo Comparison of Strategies for Blackjack."  
Undergrad Thesis. Dartmouth University.