# The Resistance

---

## CITS3001 Report

Chauntelle Bonser (22706693)

Thomas Cleary (21704985)

# Table of Contents

# 1.0 Literature Review of Suitable Techniques

## 1.1 The Resistance

The resistance is a deterministic game where players are divided into two parties, the spies and the resistance. Spies have perfect information of the game, as they are aware of the identity of other spies and thus know who the resistance members are. Whereas the resistance has imperfect information, and the entire state of the game is not apparent, instead it must be deduced through logical reasoning obtained from vote decisions and mission outcomes. The spy's objective is to sabotage missions without compromising their identity, while the role of the resistance is to identify who they cannot trust to make decisions that will support a win. [1] The following topics explore potential techniques that can be adapted to construct an effective agent for The Resistance.

The Resistance is a game with a high degree of uncertainty, with the only known information for Resistance members being their own role. Spies on the other hand know the role of every player. Despite the extra knowledge gifted to the spies any player who must make a decision throughout a game cannot know for certain the state of the game after making such a decision. In fact, a player may go all game without their decisions having any impact whatsoever.

## 1.2 Deductive vs Inductive Reasoning

Deductive reasoning is a process that evaluates a hypothesis and reaches a logical conclusion provided sufficient information, usually concluding if a statement is true or false. The result allows us to infer that if something is true for a particular class of things in general, then it would also be true for all members in the subset of that class. Inductive reasoning on the other hand utilises observations and heuristics in a probabilistic situation where there is insufficient information to make a definitive determination. [2] In the case of the Resistance, where there is significant amounts of uncertainty and a lack of transparency, the latter method of probabilistic insinuation is better suited.

## 1.3 Monte-Carlo Tree-Search

The Monte-Carlo Tree-Search (MCTS) iteratively searches and constructs an asymmetric decision tree from the results of locally created simulations [3]. The simulations are executed using randomised moves and the average win rate of each different move is recorded and accrued to existing nodes [4]. To optimise the search, the MCTS follows an exploration-exploitation trade-off principle. Exploration expands the tree's breadth faster than the depth, exponentially increasing the computational time and storage, which is particularly ineffective in a game such as The Resistance where the branching factor is quite large. This is where exploitation comes into play. Exploitation reduces this effect by following a greedy approach that selects the locally optimal path that has the greatest estimated value, extending the depth over the breadth.

There are four steps to the MCTS:

1. Selection: The tree is traversed from the root node using a selection policy until either a terminal node (a node representing a win or loss / playout) or a node that is not fully expanded is reached. The selection policy most commonly used is the Upper Confidence Bound for Trees (UCT). The algorithm follows the following formula:

$$UCT(v') = \frac{Q(v')}{N(v')} + c\sqrt{2\frac{\ln N(v)}{N(v')}}$$

Where $v$ is the parent node. $v'$ is the child node, $N(v)$ is the number of times $v$ has been visited, $N(v')$ is the number of times $v'$ has been visited, $Q(v')$ is the accumulated total reward of all playouts (win or loss) that passed through $v'$ and $c > 0$ is the exploration

constant. [3] The first term is known as the exploitation factor, and the second term is referred to as the exploration factor. [5]

2. Expansion: A child node is then appended to the selected node if it is non-terminal which is reached through a legal move from the parent node.
3. Simulation: Legal moves are randomly simulated until an end-state is obtained and the playout is determined [5], or a sufficiently accurate estimation is found without requiring further simulations. [6]
4. Back-propagation: The result from the simulation is then dynamically accumulated on all of the parents of the node until the root node is reached.

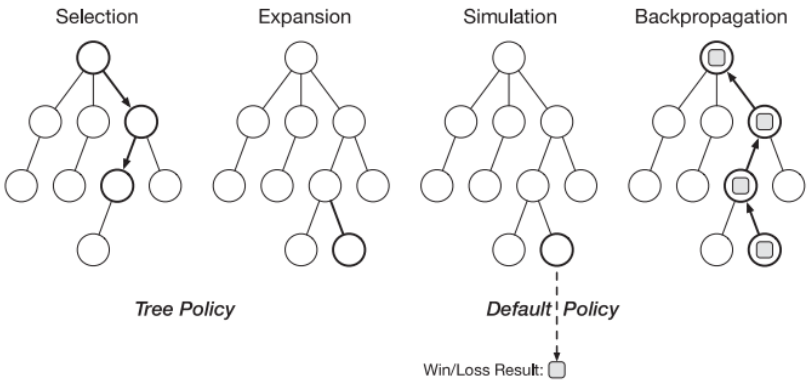


Figure 1. MCTS algorithm steps [5]

A modified form of the MCTS was introduced by Cowing [5] as the *Information Set Monte-Carlo Tree-Search* (ISMCTS) to encapsulate decisions made in situations where there is imperfect information regarding the game state. Each child node that stems from an opponent's node depicts all of the opponent's moves that are within the player's information set. The probability distribution is then altered due to the non-uniform nature of the tree and a modified version of the UCT is used:

$$ISUCT(v') = \frac{Q(v')}{N(n')} + c\sqrt{\frac{\ln N'(v')}{N(v')}}$$

A successful application of the MCTS can be seen in Go, a deterministic and perfect information board game. [7] In the research article by A. Gusmão and T. Raiko [8], a MCTS solution with Revisit-UTS algorithm was found to be an effective solution to constructing an AI for Go. This solution is also far more efficient than techniques such as minimax search with alpha-beta pruning due to the large board size of up to 361 tiles.

The research paper [9] by D. Whitehouse describes an effective strategy to approach hidden information games like The Resistance which was done by grouping information sets independently. In this method, an ISMCTS was used where each node in the tree had its own unique information set and only decision nodes were stored to increase memory efficiency. This technique is possible as the MCTS does not require a complete knowledge of the game's domain when applied to large problems.

## 1.4 Bayes Theorem

Bayes theorem is a method of finding the value of $P(y|x)$ if $P(x|y)$ is known and combines two rules of probability: the sum rule $P(x) = \sum_{y \in Y} P(x, y)$, and the product rule $P(x, y) = P(x)P(y|x)$. [10] The bayes theorem is then given as:

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)}$$

An example of Bayes theorem is used to find the probability of getting heads when flipping a biased coin. [11] In this situation there are two coins, a biased coin, $C_b$, that results in heads 75% of the time or $P(H|C_b) = 0.75$ and a fair coin, $C$, where $P(H|C) = 0.5$. We want to find $P(C_b|H)$, the probability that the coin is biased if it results in heads. Both coins are as equally likely to be chosen, therefore $P(C) = P(C_b) = 0.5$. $P(H)$ is found by summing the probability of observing heads with both coins:

$$P(H) = P(H|C) \times P(C) + P(H|C_b) \times P(C_b)$$

$$P(H) = 0.5 \times 0.5 + 0.75 \times 0.5 = 0.625$$

Then substituting these values into the Bayes theorem:

$$P(C_b|H) = \frac{P(H|C_b)P(C_b)}{P(H)} = \frac{0.75 \times 0.5}{0.625} = 0.6$$

Therefore, there is a 60% chance that a coin resulting in heads is the biased coin. On further iterations, the probabilities are subject to change. For example, if the coin is tossed again the value of $P(C_b)$ changes because the first toss suggests that the biased coin is more likely to be chosen. [11] Overtime, the predictions are refined.

A Bayesian application in five-card poker, a non-deterministic zero-sum game with imperfect information, can be made by constructing a Bayesian network as seen in figure 2 for decision making. [12] The method estimates the chances of winning by calculating conditional probability tables based on the available information, such as what cards that are face up on the table and what cards in the players hands. The algorithm then selects an action with some probability based on the expected winnings of that action.

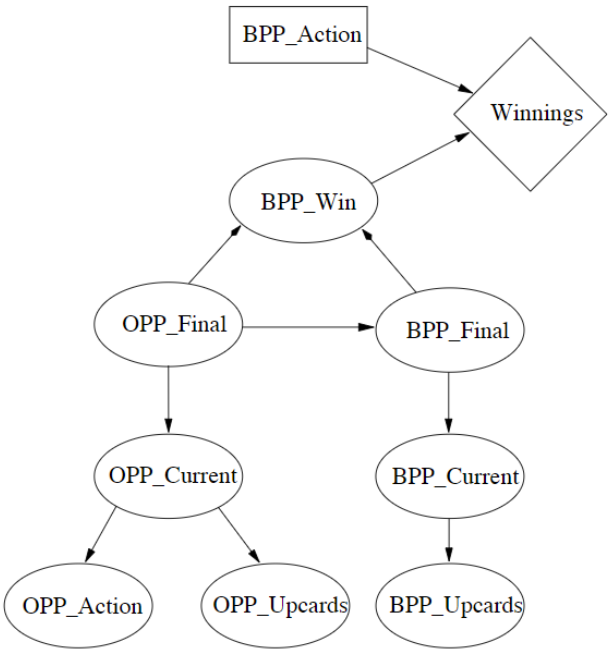


Figure 2. Bayesian network structure for 5-card poker [12]

# 2.0 Selected Technique

In The Resistance, a game with a high degree of uncertainty, the use of deductive reasoning and logic does not seem to provide an approach that can be of much use when there are only five rounds in which to attain definitive information. Deductive reasoning requires true and false values which Resistance members do not have initially and likely will not have by the end of five rounds however, spies do have such information. Even though spies would be able to provide Boolean values of player roles, they are still not able to predict the future state of the game as many states require the decision of multiple players. Hence it does not seem likely that deductive reasoning would be an effective approach to take with building our agents for The Resistance.

Clearly using a technique involving Bayes Theorem may provide a useful strategy for playing a non-deterministic game. Resistance members would be able to maintain probability tables about who they assume the spies to be. These probabilities could then be refined as more information is ascertained from the playout of the game. Spies on the other hand would not benefit so much from this as they already know the role of every player. However, they may benefit from also keeping such probabilities, so they are aware of who the Resistance members assume the spies to be, thus providing them with information on whether it is worth not betraying a mission. However, we are more interested in investigating the MCTS Technique.

A technique such as MCTS is clearly very successful at finding good moves in deterministic games such as Chess or Go where the future state of the game after a move can be determined by the player who's turn it is. What is unclear to us is how well this technique applies to a non-deterministic game such as The Resistance.

For example, if a leader proposes a team to go on a mission they cannot determine the following state of the game. Either the vote passes and the mission goes ahead, or the vote does not pass and the next leader proposes one of many possible teams. This in clear contrast to a game such as Chess, where a player can know the next state of the game after they have decided what move to make. If they decide to move a pawn one space up the board, the following state of the game will be that of the previous with the pawn moved one space up the board.

The MCTS is an "anytime" algorithm, allowing us to cease execution of the search at any time point and still receive a decision is a clear advantage of MCTS regardless of whether the game is deterministic or not. Depending on the run time we will be able to adjust the parameter that determines when the algorithm should stop running, be it a time limit or limit on the number of simulations executed.

Implementing MCTS on the game tree of deterministic games such as Chess or Go has been shown to be very effective at providing good play. Our question then is how well does it perform when applied to a game with large uncertainty such as The Resistance?

# 3.0 Implementation of Technique

Firstly, two simple reflex agents were created to gather a baseline for performance. *RandomAgent* makes decisions purely at random by proposing random teams, voting randomly and betraying randomly. *Basic*Agent on the other hand implements some basic decision-making logic that a casual player may try and use. For example, it keeps track of the number of times a players has been involved in a failed mission and uses this information to determine teams to propose, missions to vote for and whether as a spy if it should betray the mission or not. The overall aim is to implement the logic of the *RandomAgent* and *Basic*Agent into the **simulation** stage of two separate MCTS agents and see how their performance compares to the simple reflex agents.

To perform a MCTS we need a game tree to search. We have decided to abstract the state of a game into three possible states namely, *ROOT_PROPOSE, PROPOSE* and *MISSION*. Each state holds information about the current game such as then number of missions failed and succeeded, the current leader, the mission proposed / on the mission.

*PROPOSE* is a node representing a team proposal from the game's current leader. Its children consist of a single child where the vote for the proposal is passed, and the next state of the game is *MISSION* as well as a child for each possible team the next leader could propose if the vote for the mission is not passed. In essence the branches connecting the *PROPOSE* state and its children on the game tree can be considered the voting decision of the player group.

The *MISSION* state is a node representing a mission the player group has voted for. It will either pass or fail depending on if there is a spy that decides to betray the mission or not. Therefore, its children consist of a *PROPOSE* node for each possible team the next leader could propose but for both the case that the mission fails and succeeds. The branches between the node and its children can be considered the outcome of the mission, pass or fail. Figure 3 below is a representation of this tree (which itself is found in *game_tree.py*).

A special game state *ROOT_PROPOSE* is used when our agent is asked to propose a team. Its children consist only of the possible teams it could propose and are represented by individual *PROPOSE* states.

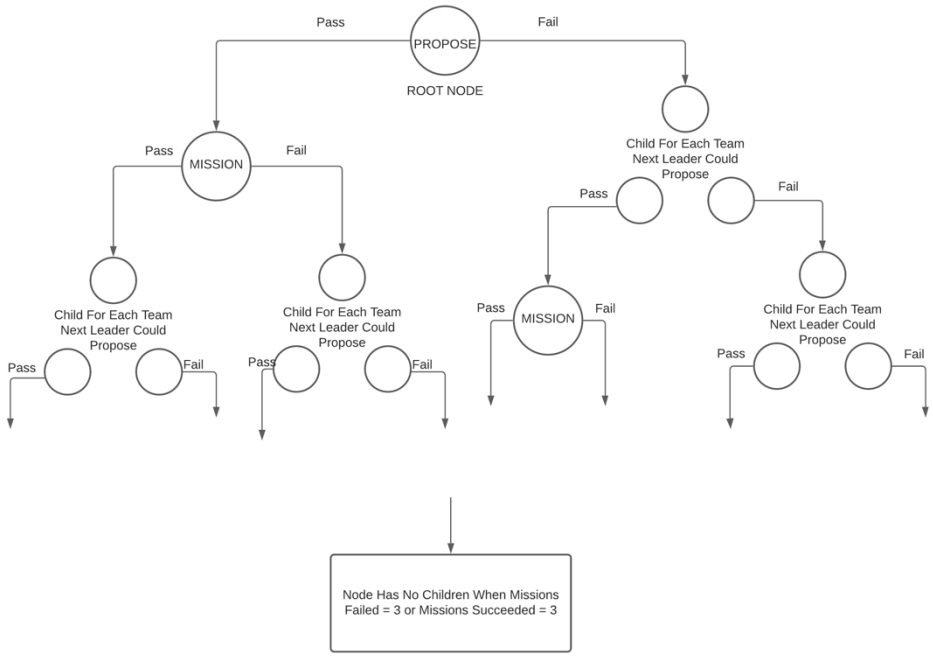


Figure 3. Representation of the game tree used in our MCTS starting at a mission proposal

Now when our MCTS agents are asked to propose a mission, vote on a mission, or betray a mission we can correctly build a tree with the current state of the game as the root node.

If our agent is asked to propose a mission team then it will start the root of the game tree as a *ROOT_PROPOSE* state. If it is asked to vote on a mission the root shall be a *PROPOSE* state with the mission team that of the one proposed by the current leader. If the agent is asked to be betray a mission the root node will be a *MISSION* state.

Now starting at the root node, we can perform a MCTS to determine the best move to make based off the playouts performed. By performing many playouts MCTS aims to find the child node of the root node that produced the most simulated wins.

Each iteration involves four steps:

1. **Selection**
   Starting from the root node we must traverse the game tree until we find a leaf node, a node with no children. To determine what path to take down the tree a selection policy is used. In this case Upper Confidence Bound for Trees (UCT) is used to select a path that balances between picking paths that have been successful in previous iterations and those that have not been explored much.

2. **Expansion**
   Once we have found a leaf node, we must expand its children and select one to simulate from. As we have not seen these nodes before one is selected at random.

3. **Simulation**
   From this selected child node, we simulate the rest of the game until it is over. We employ two different approaches to simulation in our MCTS agents as to replicate how *randomAgent* and *basicAgent* would play the game.

4. **Backpropagation**
   The result, either a win or loss for the agent, then needs to be updated within each node visited on the path to the child node. Each node visited also needs its simulation count incremented (these values are attributes of the *GameState* class).

When the MCTS algorithm has finished we can then determine the best move for the agent to make based off the transition of state type from the root node to the best child node, that is the child node with the highest UCT value.

| Root Node State | Best Child State | Best Move |
|---|---|---|
| ROOT_PROPOSE | Always PROPOSE | Use the mission team from this child as the team to propose |
| PROPOSE | ROOT_PROPOSE, PROPOSE | Next best state is to have another team proposed.<br>Therefore, vote FAIL |
| PROPOSE | MISSION | Next best state is for mission to go ahead.<br>Therefore, vote PASS |
| MISSION | PROPOSE (missions passed + 1) | Next best state has 1 more mission passed<br>Therefore, DO NOT BETRAY mission |
| MISSION | PROPOSE (missions failed + 1) | Next best state has 1 more mission failed.<br>Therefore, BETRAY mission |

To compare against both of our reflex agents, *randomAgent and basicAgent*, two separate simulation functions were created to use in separate MCTS agents. The first being a simulation function named *random_simulation()* randomly decides whether a game was won or lost and is used by our first MCTS agent, *randomMCTS*. The second simulation function named *basic_simulate*() plays out the remainder of the game as if basic agents were playing. The number of fails each player has been involved in are used to just like in *basic*Agent to determine, or guess each players move.

By using simulations that replicate the above-mentioned reflex agents decision processes we hope to be able to determine whether MCTS is able to help an agent perform better when pitted against these simple agents.

# 4.0 Validation of Agent Performance

To test each of our agent's performance we have tracked their overall win ratio as well as win ratio as a spy and resistance member separately. We have created two simulation scenarios with different sets of opponents. The first pits the agent against only *randomAgent* opponents, whilst the second uses only *basicAgent* opponents.

Firstly, we gathered a baseline for which to compare our MCTS agents against by testing our simple reflex agents. As these agents require little processing time, we were able to run one million games for both simulation scenarios for each agent. This allowed us to find converged values for the above-mentioned win ratios as shown below in figure 4.

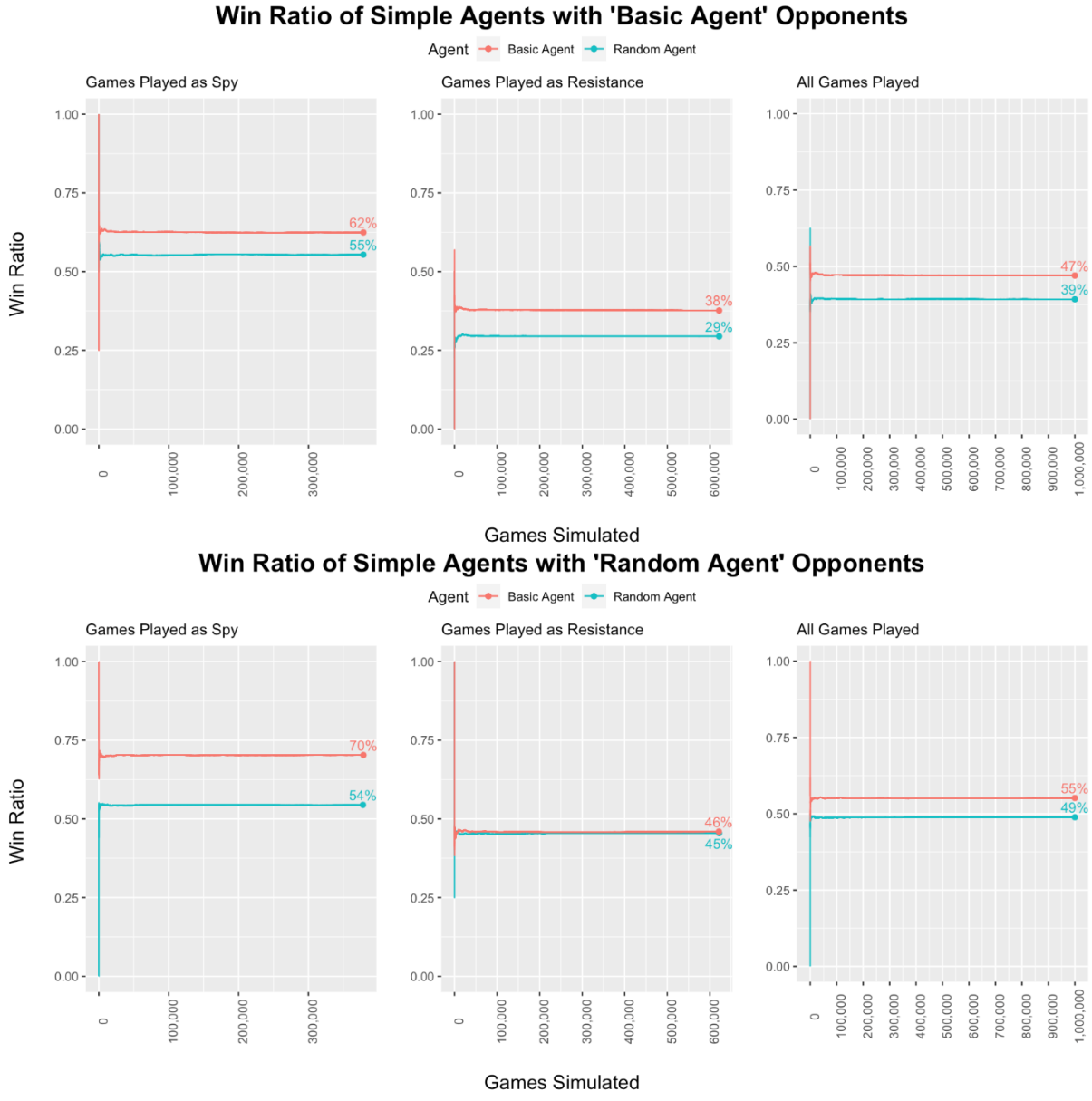


Figure 4. Simple Agent Performance

Using these "baselines" we can now compare the performance of both our MCTS agents, *randomMCTSAgent* and *basic*MCTSAgent when placed in the same game environments. As the MCTS agents require much more processing time we have limited the number of games simulated to 5000. Figure 5 illustrates the results of the simulations.

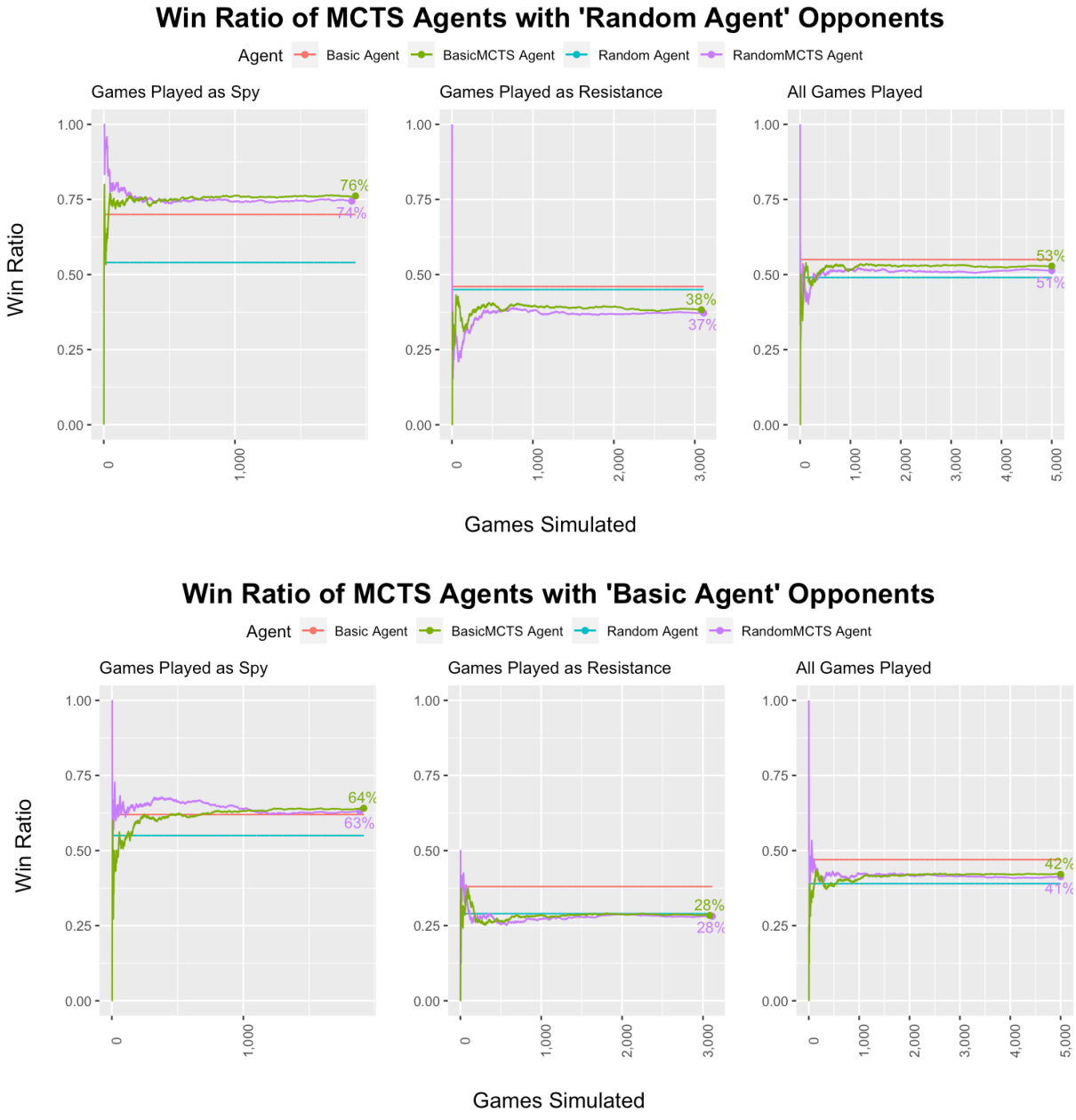


Figure 5. MCTS Agent Performance

| % Wins for Tested Agent with "Random Agent" Opponents | | | |
|---|---|---|---|
| **Agent Name** | **%Wins as Spy** | **% Wins as Resistance** | **Overall Win %** |
| randomAgent | 54% | 45% | 49% |
| basicAgent | 70% | 46% | 55% |
| randomMCTSAgent | 74% | 37% | 51% |
| basicMCTSAgent | 76% | 38% | 53% |

From the results displayed above it is clear there is an advantage towards playing as a spy. The win ratios for playing as a spy are much higher than that of those when playing as a resistance. However, from Figure 4 and 5 we can see that each agent only plays as a spy roughly 1/3 of the time. Therefore, the overall win ratio is not the average of the two values.

From the above table when all other players are random, we can see that both MCTS agents perform significantly better than *randomAgent* and slightly better than *basicAgent* when playing as a spy. However, the win ratio when playing as a resistance member is considerably lower for the MCTS agents when compared to both "simple" agents. Overall, the performance of the MCTS sits between that of *randomAgent* and *basicAgent*. It appears the more complex simulation in *basicAgentMCTS* has had little impact performance.

| % Wins for Tested Agent with "Basic Agent" Opponents | | | |
|---|---|---|---|
| **Agent Name** | **%Wins as Spy** | **% Wins as Resistance** | **Overall Win %** |
| randomAgent | 55% | 29% | 39% |
| basicAgent | 62% | 38% | 47% |
| randomMCTSAgent | 63% | 28% | 41% |
| basicMCTSAgent | 64% | 28% | 42% |

When playing with *basicAgents*, all agents tested win less often than when playing against *randomAgents*. Now however, the performance benefit of MCTS agents when playing as a spy when compared to *basicAgent* has reduced dramatically to 1 and 2% for *randomMCTSAgent* and *basicMCTSAgent* respectively. Again, the MCTS agents perform worse than our simple agents when playing as resistance members. Relative to one another the order of overall performance is the same as with *randomAgent* opponents.

To create one final agent, which we will call *mixedAgent,* we will use the MCTS with random simulation from *randomMCTS* to make decisions if the agent is a spy, and the logic from *basicAgent* to make decisions if the agent is a resistance member (as these are the best performing agents for each role type against *randomAgent* ). As a final test, we have played the five agents against each other for 5000 games. The results of which are shown below.

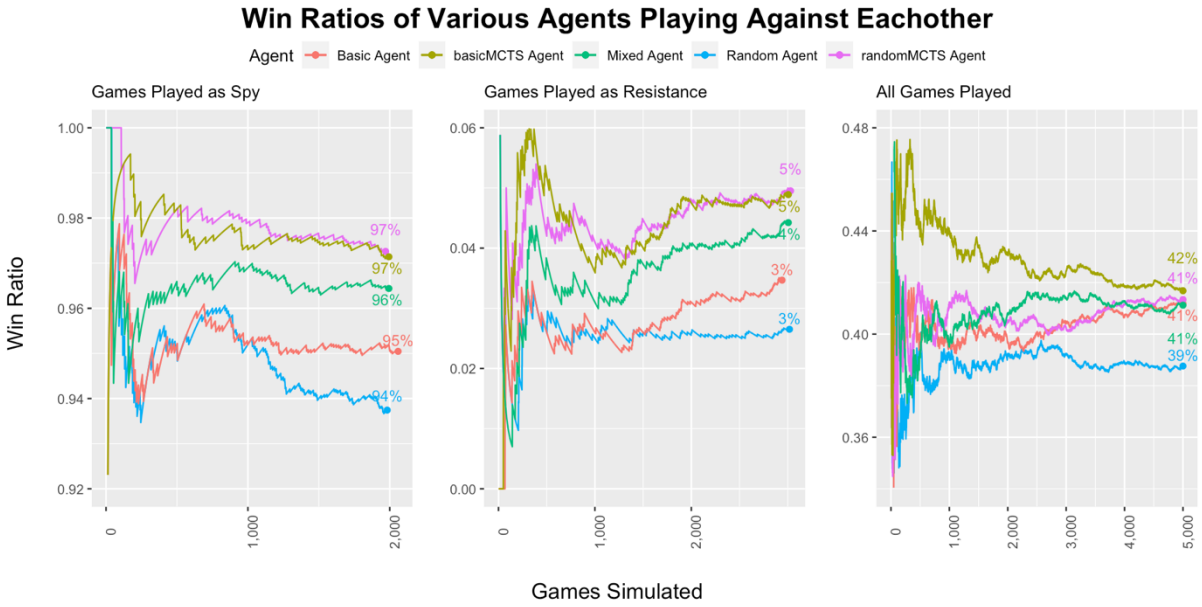


Figure 6. Tournament Results

From figure 6 we can see that each of our five agents performs roughly the same when playing against one another. However, if we were to treat this as a competition, the winner would be our *basicMCTSAgent* as it has the highest overall win percentage.

In summary it is clear from the results above that more is needed than a simple MCTS of a simplified game tree for an agent to perform better than a reflex agent such as *basicAgent* in The Resistance. If we were to attempt this again, incorporating elements of bayes theorem into the simulation part of MCTS may be an interesting avenue of investigation.

# 5.0 References

[1] D. Thompson, “Teaching Validity and Soundness of Arguments Using the Board Game:The Resistance,” *PRIMUS*, vol. 25, no. 6, pp. 542–552, May 2015, doi: 10.1080/10511970.2015.1038859.

[2] E. Heit and C. M. Rotello, “Relations between inductive reasoning and deductive reasoning.,” Journal of Experimental Psychology: Learning, Memory, and Cognition, vol. 36, no. 3, pp. 805–812, 2010, doi: 10.1037/a0018784.

[3] S. D. Palma and P. L. Lanzi, “Traditional Wisdom and Monte Carlo Tree Search Face-to-Face in the Card Game Scopone,” *IEEE Transactions on Games*, vol. 10, no. 3, pp. 317–332, Sep. 2018, doi: 10.1109/tg.2018.2834618.

[4] R. Kelly and D. Churchill, “Comparison of Monte Carlo Tree Search Methods in the Imperfect Information Card Game Cribbage,” 2017.

[5] E. S. Steinmetz and M. Gini, “More Trees or Larger Trees: Parallelizing Monte Carlo Tree Search,” *IEEE Transactions on Games*, pp. 315–319, Sep. 2021, doi: 10.1109/tg.2020.3048331.

[6] M. C. Fu, “Monte Carlo Tree Search: A Tutorial,” *IEEE Xplore*, pp. 222–236, Dec. 2018, doi: 10.1109/WSC.2018.8632344.

[7] D. P. Taylor, “Investigating Approaches to Ai for Trust-Based, Multi-Agent Board Games with Imperfect Information with Don Eskridge’s ‘The Resistance,’” 2014.

[8] A. Gusmão and T. Raiko, “Towards Generalizing the Success of Monte-Carlo Tree Search beyond the Game of Go,” Frontiers in Artificial Intelligence and Applications, vol. 242, pp. 384–389, 2012, doi: 10.3233/978-1-61499-098-7-384.

[9] D. Whitehouse, “Monte Carlo Tree Search for games with Hidden Information and Uncertainty,” Aug. 2014.

[10] Z. Ghahramani, “Probabilistic machine learning and artificial intelligence,” Nature, vol. 521, no. 7553, pp. 452–459, May 2015, doi: 10.1038/nature14541.

[11] J. L. Puga, M. Krzywinski, and N. Altman, “Bayes’ theorem,” Nature Methods, vol. 12, no. 4, pp. 277–278, Mar. 2015, doi: 10.1038/nmeth.3335.

[12] K. B. Korb and A. E. Nicholson, Bayesian Artificial Intelligence, 2nd ed. CRC Press, 2010, pp. 156–163.