Resistance AI

1 Introduction

The Resistance is a is a multi-player party game released in 2010, involving between five and ten players. Each player is part of a resistance group, aiming to overthrow the government. However a third of the players, rounded up, are secretly assigned to be government spies. In each round of the game, a team (a subset of a specified size of the players) is chosen by an assigned leader and voted on by the rest of the team. If the mission passes, it is carried out by the team. If not, another leader is allocated. When a mission is chosen and passes the vote, each spy on the team has the option to sabotage the mission. If the mission is sabotaged by enough spies (in most cases only one sabotage is required) it fails. If the mission passes, the resistance members get one point. If the mission fails, the spies get one point. However, the spies must be careful not to incriminate themselves by sabotaging too eagerly, since then they may be identified by the true resistance members and will not be chosen for further missions. The first group to 3 points wins the game. *The Resistance* is partially observable for the true resistance members and fully observable for the government spies. The game also has a very minor stochastic element, since the leader of a mission is allocated randomly at the start of the game.

In this report, approaches to constructing artificially intelligent agents for playing *The Resistance* are investigated. Alongside the report, source code for two different agents are provided: one using a Bayesian inference technique and another using a Monte Carlo Tree Search technique.

2 Literature Review

Three works on artificially intelligent agents that refer explicitly to *The Resistance* are identified and discussed in this report. In 2012, members of the Dagstuhl Seminar "Artificial and Computational Intelligence in Games" [9] arranged a competition whereby programmers developed an artificially intelligence agent to play *The Resistance*. This tournament appeared to be the catalyst for several analyses on *The Resistance*. The 2012 competition put an emphasis on learning-based approaches [10] over search-based approaches. The agents in the competition relied predominantly upon learning and *expert rules*: human-specified condition-action rules based on intuition (as an example, hard-coding a spy to sabotage if the spy team already has two points).

2.1 Bayesian Inference

Bayes' rule for updating beliefs was used as a component of several bots entered in the Dagstuhl competition, and was also discussed by Whitehouse [15] in particular with respect to *opponent modelling* and by Taylor who focussed on integrating Bayesian Inference into Monte Carlo Tree Search [13]. Bayes' rule is given by

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)} \tag{1}$$

where A and B are events. $P(A|B)$ is the *posterior* belief given observations, $P(B|A)$ is the *likelihood* of observation of B given A, $P(A)$ is the *prior* belief, and $P(B)$ is the *evidence* [3]. Bayesian inference is an important concept in *The Resistance*, since it is advantageous for resistance players to identify which players are most likely to be spies. Assigning a probability for player k to be a spy as $P(k \text{ spy})$ - initially equal for all other players - and having just completed a mission with n sabotages, the belief for player k to be a spy can be updated by

$$P(k \text{ spy} \mid n \text{ traitors}) = \frac{P(k \text{ spy})P(n \text{ traitors} \mid k \text{ spy})}{P(n \text{ traitors})} \tag{2}$$

Equation (2) contains a dependency on an opponent model. To calculate $P(n \text{ traitors} \mid k \text{ spy})$ and $P(n \text{ traitors})$, the probability with which each spy s chooses to sabotage the mission must be known in order to exactly determine the probability of n sabotages occurring. Clearly each agent does not have access to the policy of any other agent, so an opponent model must be developed to estimate this sabotage probability. As discussed by Cowling and Whitehouse [3], a common practice for Bayesian inference systems for analogous unknown-opponent-policy games such as Poker and Skat compute an *offline* opponent model. This opponent model is then tuned in response to player behaviours while the game is in progress.

If an accurate opponent model is achieved, Bayesian inference can be used to exploit the opponent. In the similar imperfect-information, stochastic multi-agent game of Poker, acting on accurate Bayesian inference can give advantages to the player such as influencing the dealt cards [11]. In *The Resistance*, an accurate opponent model would make choosing/nominating teams without spies more effective for the resistance agent, and could also be used to bluff as a spy agent by exploiting the resistance members' inferences.

2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a strategy for game-playing, and is the most extensively discussed technique in the literature relating to *The Resistance*. MCTS works by representing all possible states of the game in a game tree, where the root node is the initial state. Every possible move creates a child node which represents the new state of the game.

MCTS works to solve the same problem as the Minimax algorithm. However, Minimax can take an infeasible amount of time when the branching factor is high. *The Resistance* can be considered to be one of these cases, considering that the branching factor for the nomination component of the game can reach up to $\binom{10}{5} = 252$ for a ten-player game with a five-person mission. Junxing Wang [14] places the number of nodes in the game tree at more than 32 million for a five-player game, clearly making a 'brute-force' Minimax-style search infeasible even with alpha-beta pruning. MCTS avoids this problem by using random sampling. The MCTS algorithm randomly plays out games by choosing an arbitrary move at each node. Next, the nodes in the tree are weighted based on the result of this randomly simulated game, where nodes which have led to a winning state more often are considered more frequently in future simulations.

In each iteration of MCTS, four steps are carried out. First, in the selection step, child nodes are chosen from the start node down to a leaf node in the game tree. This step balances exploration with exploitation by selecting the child node that maximises the 'Upper Confidence Bound 1 applied to trees' (UCT) formula [7]:

$$UCT(i) = \frac{w_i}{n_i} + \sqrt{\frac{2 \ln \sum_i n_i}{n_i}} \tag{3}$$

where w_i is the number of wins from the ith node and n_i is the number of simulations that node i has been involved in. The first term in (3) is the exploitation parameter - proportional to the win ratio - and the second term is the exploration parameter, which decreases each time the node is selected. Next is the expansion step, where a random child node of this leaf is chosen. Then for the simulation step a random 'playout' occurs from this node, where completely random moves are chosen at each stage until the game ends. Finally the result of the final game is back-propagated up the tree, with each node storing the number of won playouts as well as the total number of playouts that involved that node. These four steps can be repeated for as long as the time allocated for making a move. The move chosen by MCTS is the one associated with the node having the highest number of simulations. This is called the robust child selection policy.

MCTS converges to Minimax as the number of iterations becomes large; that is, it approaches optimal play against an optimal opponent [1]. Two further advantages of Monte Carlo Tree Search are that is can be quite easily parallelised [2] and that the search algorithm can be terminated at any time, returning the best-found move so far.

Pure MCTS is only applicable in perfect-information scenarios, so it does not apply to *The Resistance* when playing as a resistance member. However in 2001 Ginsberg [6] demonstrated an adaptation of Monte Carlo methods to imperfect-information games, by sampling states from the current *information set* (the set of possible states of the game from the perspective of a particular player). This was later adapted to MCTS [4] to form Information Set Monte Carlo Tree Search (ISMCTS), which stores information sets in each node of the game tree. Then MCTS is extended by choosing an arbitrary 'determinisation' from the information set at the selection step of the algorithm. This technique can also be used to bluff using self-determinisation methods [3], where game states that the player knows to be incorrect can still be sampled from. These are states where the resistance members have not yet eliminated the possibility of the sampled game state being the correct one.

2.3 Genetic Algorithms

Playing well as a learning agent in *The Resistance* relies on choosing a set of parameters that define the choices made by an agent at each game step. For example, an agent purely constructed from expert rules will require some 'threshold values' for making decisions. As another example, a resistance agent performing inference on other players needs to estimate the probability that a spy will sabotage a particular mission given the opportunity.

In addition, these values are not necessarily independent. The problem is then how to find the set of parameters that optimise the win rate of the agent. One technique for solving this optimisation problem is to use a genetic algorithm. A genetic algorithm is a technique inspired by natural selection: it works by maintaining a population of candidate solutions, with varying parameter values [5]. Usually the set of parameters are encoded using a binary string. Then, at each stage, a new *generation* is formed using the most 'fit' (successful) solutions. This selection choice is based off a 'fitness function' - potentially in the case of *The Resistance*, the win rate of the given agent. These new solutions are generated using a combination of two techniques, mutation and crossover. Mutation involves flipping some arbitrary bits in a small proportion of newly created agents, aiming to maintain diversity in the set of solutions. Crossover is analogous to reproduction: two 'parent' solutions produce two new solutions via selecting an arbitrary position in their bit strings and swapping the values of the two strings up to this position. A genetic algorithm can continue producing new solutions indefinitely - a suitable termination point may be to stop the algorithm once an agent with a win rate above some minimum point is produced.

Genetic algorithms and other optimisation techniques have not been discussed in the literature in the context of *The Resistance*.

2.4 Knowledge-based Techniques

Since the state of the system is partially observable in *The Resistance*, reasoning about the state given evidence is important for optimal play. As a resistance member, facts about which opponents are spies can be deduced - i.e. identifying the correct state of the game. As a spy player, the amount of deduction that can be performed by the resistance members must be kept to a minimum. For both these techniques, various proof and knowledge-related techniques can be used. Knowledge-based techniques are not discussed extensively in the literature on *The Resistance*, but variations on knowledge/logic-based methods are used extensively in agents from the Dagstuhl competition [13].

Knowledge-based agents perform reasoning on an internal representation of knowledge (a knowledge base) to infer facts. A language used for internally representing knowledge requires associated semantics - defining the truth of sentences in each possible model of the world. Inference rules (deriving new sentences from old ones) can be used to prove new sentences using an inference engine and therefore derive new knowledge. Automated theorem proving can be used for general game-playing where a good strategy, and potentially even the rules of the game, is initially unknown [12]. Although this may not be strictly applicable to *The Resistance*, theorem-proving is closely related to the problem of constraint satisfaction - finding a solution to a set of constraints that must be satisfied by the variables. Constraint satisfaction can be applied to partially observable games like minesweeper and is a good fit for *The Resistance* in identifying which states of the game (spy combinations) are possible given the evidence from mission sabotages. The agent `Rebounder` [13] from the 2012 Dagstuhl competition uses a knowledge-based/constraint satisfaction technique as a resistance member: it enumerates all possible combinations of spies, and at each stage eliminates the combinations which contradict the history of the game. This is done by considering which models of the world (which players are spies) are consistent with the sabotage history of the game. However, no explicitly knowledge-based techniques (i.e. automated theorem proving) have been found in *The Resistance*-playing agents so far.

Proof/reasoning-based techniques can also be extended to include reasoning about the agent's knowledge itself using epistemic modal logic. If agent a knows that ϕ holds, then we can write $K_a\phi$. If agent a believes that ϕ holds, then $B_a\phi$ [8]. Inference rules can be naturally extended to work with an epistemic logic to infer what agents do and don't know in a multi-agent system. This is particularly useful in *The Resistance*, since it can be used as a spy agent to choose moves which minimise the knowledge that resistance agents have.

3 Rationale

Certainty/knowledge-based techniques such as epistemic logic, logic programming, and theorem proving are less powerful than Bayesian inference for *The Resistance.* Each of these concepts relies on facts gained by the number of sabotages. However, if no sabotages occur, no firm conclusions at all can be drawn. Yet with Bayesian inference, given an opponent model where a spy is more likely to sabotage than not, this situation implies that the members of the team are less likely to be spies. In addition, Bayesian inference can infer at least as much as logic-based techniques: if B and E are on a mission and two sabotages occur, both methods will successfully deduce the spies. Therefore, using Bayesian inference is a more general and more powerful method for this game since it provides information about who the *likely* spies are even if the spies aren't known to the agent with certainty.

Genetic algorithms may be challenging to use in the context of *The Resistance* due to difficulties mainly involving the fitness function. Clearly one basic fitness function that could be used is to compute the win rate of the agent over some number of games. However, this requires using a specific set of other agents to play against/with. So the final agent would be excellent at playing against that specific set of agents, and may not adapt well to new styles of playing. In addition, in general a genetic algorithm will need to iterate hundreds of thousands or even millions of times to consistently produce a good result, so the fitness function needs to evaluate quickly. However the win rate itself needs hundreds of games to eliminate random noise, leading to a large amount of training being required for an agent which may end up still performing poorly against players with a different play-style than those it has been trained against.

When the agent is a spy it has perfect information, so MCTS is an excellent choice. An advantage is that no expert rules at all need to be implemented - MCTS 'discovers' advantageous moves in each situation which no longer need to be hard-coded. For example, it was noticed that the MCTS spy agent implemented in the following section continually voted down missions if it was optimal for the mission on the fifth nomination attempt to be automatically accepted (i.e. if the leader of that mission is a spy). These and similar edge cases are hard to spot and can be messy to implement, so the advantage of the MCTS agent is that these smart plays no longer need to be hard-coded to produce a strong agent.

From considering the above arguments, the two techniques decided on were Bayesian inference and Monte Carlo Tree Search. The implementation of these two *Resistance*-playing agents are discussed in the next section.

4 Implementation

4.1 Agent 1: `BayesAgent`

This agent uses Bayesian inference, as per the previous section, as both a resistance and government team member. The most important section of code is in the `ResistancePerspective` class. This class represents the perspective of a resistance agent and maintains a spy probability on each other player in the game. Initially the spy probabilities of other players are equally distributed as $\frac{\texttt{nspies}}{\texttt{nplayers}-1}$. In addition, a sabotage probability is held for each player: this represents the probability that the player, if a spy, will sabotage a given mission if it has the opportunity. After each round, the following sequence of operations are executed:

1. For each player k, $P(n \text{ traitors} \mid k \text{ spy})$ is calculated. First, the spy probability is temporarily set to 1.0 to indicate the assumption that k is a spy. Then the probability of this scenario is calculated by iterating over all possible combinations of spies. Each spy combination is then further broken down into component probabilities by iterating over all possible combinations of which spies chose to sabotage. For each of these combinations which is valid (i.e. the number of spies who choose to sabotage in this combination matches with the number of spies which *did* sabotage), the probability of this spy sabotage combination is

$$P_c = \prod_{p=0}^{\texttt{nplayers}} \begin{cases} \texttt{spy\_prob}(p) \times \texttt{betray\_prob}(p, m) & p \text{ spy, on mission, sabotaged} \\ \texttt{spy\_prob}(p) \times (1 - \texttt{betray\_prob}(p, m)) & p \text{ spy, on mission, didn't sabotage} \\ \texttt{spy\_prob}(p) & p \text{ spy, not on mission} \\ \texttt{1 - spy\_prob}(p) & p \text{ not spy} \end{cases} \tag{4}$$

   where c is the spy-sabotage combination and m is the mission.

2. P_c is then summed for all possible combinations of spies - these events are independent, so each component probability can be added to give the resulting total probability for the

number of sabotages to have occurred given that k is a spy. This value is returned and the spy probability of player k is reset from 1.0 to the original value.

3. Each player's spy probability is updated using (2) with the denominator $P(n \text{ traitors})$ set to 1.0. Then the spy probabilities are normalised so that the total spy probability sums to the number of spies in the game. This is much more efficient than computing $P(n \text{ traitors})$ using the same method as above, since it is just a constant for all players.

As well as storing a spy probability computed using Bayesian inference, some other behaviour-tracking techniques are used to calculate the overall 'spyness' of each player. These behaviour-tracking techniques are based on those implemented in `PandSBot`, an agent from the 2012 Dagstahl competition [9]. The three behaviours tracked are:

- `helpedSpies`: This tracks how consistently the player has assisted the spy team. Each time a mission is completed, the agent considers who voted for the mission. If the player voted yes for a mission which was sabotaged, this is a piece of evidence for the premise that the player is a spy. Likewise if the player voted no for a mission which was not sabotaged. If neither of the previous statements hold, then a piece of evidence is added to the variable *against* the argument.

- `behavedLikeSpy`: This tracks general suspicious behaviour. It considers cases including when a leader nominates a team not including him/herself, when a player votes no to the very first nomination of the game (this move would only make sense if the person was a spy who knew that there were no spies on the nominated team), and when a player not on the team votes yes to a team with the same number of members as the number of resistance agents (if they're resistance, they must know that there is at least one spy on the team in that case).

- `behavedLikeResistance`: This tracks behaviour that helps the resistance team. In particular, it tracks members who vote against teams which they are on - it seems likely that only a resistance member would do this in most cases.

Each of these variables are converted to probabilities by dividing the number of cases in which the situation held by the number of cases considered (e.g. in the case of `behavedLikeResistance`, dividing the number of times the player voted against teams they were on by the total number of times they voted). Then, the total spyness probability in `ResistancePerspective.Player#spyness()` is obtained by simply multiplying together the probabilities small weightings for each behaviour.

Finally, the suspicion of potential spy combinations is tracked. Each possible combination of spies is stored along with a mapping to the overall probability of the combination being the correct spy combination. This is essentially computed by multiplying together the spyness of each player in the combination. However, the strength of the connection between players in the combination is also considered: a probability of a player being 'friends' with each other player is stored. This friendship probability is increased for every other player that the player nominates as part of a team, or votes for. The final spyness for the group is updated by multiplying it with every friendship probability of player pairs within the combination.

From these combination values, the agent can now make decisions: it nominates a team by taking all players except those in the highest-spyness combination, sorts them in ascending order by spyness, and picks players from the start of the list until reaching the required selection count. It votes yes only for teams which do not contain any players in the highest-spyness combination.

The spy component of this agent uses Bayesian inference as well. It stores the `ResistancePerspective` of the game from every resistance member's point of view. The team nomination is chosen by iterating over all possible team nominations containing enough spies to win a point, assuming that every spy on the team will sabotage, and then minimising the certainty of the resistance players' inferences following a simulated mission using each team. This 'certainty minimisation' is done by finding the minimum of the function

$$\sum_{\substack{p \\ k \neq \texttt{me}(p)}} \left(\texttt{spy\_prob}(p,k) - \frac{\texttt{nspies}}{\texttt{nplayers}-1} \right)^2 \tag{5}$$

where p is over the perspectives and k is over each player. This is done to bring the spy probabilities of each player as close as possible to having the evenly distributed spy probabilities of $\frac{\texttt{nspies}}{\texttt{nplayers}-1}$. Then the nomination which is makes the resistance members most uncertain is selected. Voting is done purely by expert rules - voting yes for any team which contains enough spies to win a point. The choice of whether to sabotage is done by considering the 'worst-case' scenario if all spies sabotage: a mission where all spies on the mission sabotage is simulated, and the resistance perspectives are temporarily updated. If any of the resulting perspectives have a spy probability which has increased to 1.0, the mission is not sabotaged (since it has a change of revealing a spy to a resistance member). If not, the mission is sabotaged.

4.2 Agent 2: `SearchAgent`

For this agent, first the Monte Carlo Tree Search algorithm was implemented in the `MCTS` class. This was implemented in a fairly abstract manner - i.e. not *The Resistance*-specific. The `MCTS` class contains two primary subclasses: the `State` interface and the recursive `Node` class. The `State` interface is implemented by a specific game to store the game state at any particular node in the game tree. It must implement methods to return all possible transitions (moves) from the current state, the current player, whether the game is over and if so, the scores of each player, and several other methods required for MCTS. The `Node` class holds information about a particular node of the game tree - number of simulations involving that node, back-propagated scores, children (visited/unvisited), the parent, and the current player at that node. Then a straightforward implementation of MCTS was developed as per the algorithm specification. The public interface exposes the three method. `#search()` starts an asynchronous loop which runs the selection-expansion-simulation-backpropagation steps repeatedly. `#transition()` interrupts the search and returns the best move found so far. `#state(State)` updates the state to search from. Next, the game state for *The Resistance* was 'discretised' by storing the entire state of the game at any point (current player, phase, number of votes so far, etc.), and required methods from `State` were implemented to transition between states.

After the basic implementation of pure MCTS was completed, it was realised that the assumption made previously - that pure MCTS would work for *The Resistance* as a spy player - was incorrect. This is because MCTS assumes that *all* players have perfect information, not just the start player. This is clearly untrue for *The Resistance.* The reason for the requirement that all players have perfect information is that when the tree is considering some game tree node where the current player is a resistance member, it randomly samples possible moves the resistance player could make to maximise its score by performing playouts to the end of the game. This gives the resistance player information that it would not otherwise have in a real game - that is, the pure MCTS algorithm gives the resistance player at the current node perfect information on which is the best move to take. Hence, MCTS implemented naïvely gives the simulated resistance members the ability to choose a team without spies every time, and thus an almost 100% win rate no matter the choices made by the spies. This resulted in entirely random moves being made by the initial `SearchAgent`, since the moves it made were inconsequential in the MCTS simulation. So in fact, pure MCTS does not work even if the current player has perfect information - only if *all* players have perfect information.

To 'patch' the MCTS algorithm to work with *The Resistance*, an opponent model was integrated into the search. When performing a playout, instead of picking a random move using a uniform distribution the moves were weighted by their likelihood of being chosen in a real game. All spy moves have equal weights, but moves by the resistance members are weighted in proportion to Bayesian suspicion values (using a lightweight version of the Bayesian inference code in `ResistancePerspective` from `BayesAgent`). The suspicion values are updated in the game state each time a mission is completed. Then various functions for weighting moves were explored. The final weighting function for nominations ranks each team by total suspicion, and then linearly assigns weights based on this. This technique is not based in theory and may or may not be correct, but it appears to work well in practice by making the agent attempt to appear less spy-like by discouraging aggressive sabotaging. The opponent model 'patching' is not ideal - ideally an imperfect information MCTS algorithm such as ISMCTS [3] would be used as a spy player alongside self-determinisation [3].

Once the opponent model was integrated, the implementation works exactly as in the pure-MCTS case. Whenever the agent receives new information it updates the game state. At each step when the agent is required to make a move, it sets up a search by passing the current game state. It then runs the search for slightly less than the time limit of 1000 ms and chooses the transition associated with the most-visited child node returned from the Monte Carlo Tree Search.

5 Validation

The Bayes and MCTS agents (considering the resistance and spy components of each separately) were developed using an iterative approach. After building the base of both agents, the weakest component of the agents was incrementally improved until it improved above another component. This process was repeated until the submission of this project.

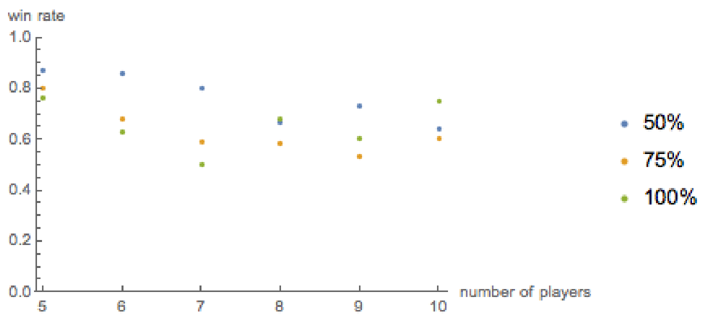

Figure 1: Win rate of a team of `BayesAgents` playing against a team composed of `RandomAgents`, over 1000 games. The legend indicates the probability that a `RandomAgent` will sabotage the mission given the opportunity.

To examine the performance of the inference engine of `BayesAgent`, a team of resistance `BayesAgents` played against a spy team composed of `RandomAgents`. As per figure 1, the agent maintains a win rate above 50% in all cases. As a baseline, when random agents play against random agents the resistance members win 3% of the time when spies sabotage with 100% probability, 22% of the time at 75% sabotage-probability and 51% of the time at 50% sabotage-probability. The agent appears to perform best against an opponent that sabotages approximately 50% of the time, and worst against an opponent which sabotages 100% of the time - this is surprising since it was expected that this would make it easier to perform inference and identify spies quickly, however this may have been offset by the fact that by the time the spies are *all* identified they have managed to sabotage three missions. The agent appears to work best in a 5-player game, likely because of the fewer possibilities for spy combinations. It performs poorest in a 7-player game, which may be because the spies have an intrinsic advantage in this case - the 7-player game has the highest proportion of spies to resistance members (3:4).

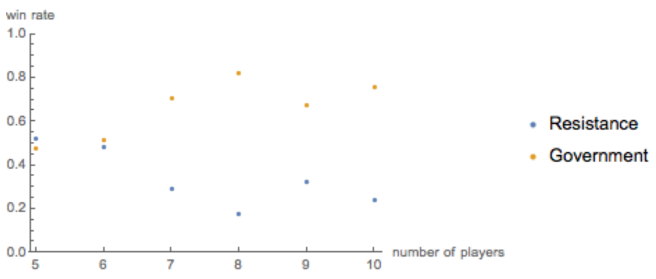

Figure 2: Win rate of `BayesAgent` playing against itself. Each data point is averaged over 1000 games.

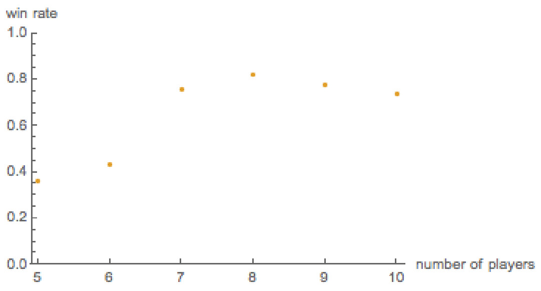

Figure 3: Win rate of `SearchAgent` spies playing against `BayesAgent` resistance members. Each data point is averaged over 100 games.

The `SearchAgent` has a 100% win rate when playing as a spy against `RandomAgent`s, which is expected since the random agents perform no inference to influence their voting/nomination choices. Initially, when the `BayesAgent` *only* performed Bayesian inference without any behaviour-tracking, the `SearchAgent` would dominate, winning almost 100% of games. However, with behaviour tracking included the `BayesAgent` resistance team performs better. This is because the `SearchAgent` opponent model was suited for agents only doing basic inference and worked to exploit that - however, this exploitation is 'flagged' by the behaviour tracking techniques described in the implementation section, making it very hard for the `SearchAgent` to win. Refer to figure 3 for the approximate win rate of the MCTS agent - note that the performance improves as the number of players increase. This is possibly due to the difficulty of correct resistance inference increasing with the number of players.

6 Conclusion

In this report, approaches to creating artificially intelligent agents for playing the game *The Resistance* have been investigated. Two new agents were developed using the techniques of Bayesian inference and Monte Carlo Tree Search. The theory, implementation and performance of these two agents were also discussed. The Bayesian inference agent was found to a strong player as both a resistance and spy player against various opponents, and the Monte Carlo Tree Search also performed relatively well as a spy. However, further work could to be done on implementing Information Set Monte Carlo Tree Search with self-determinisation to produce a stronger MCTS spy agent.

References

[1] C. B. Browne et al. “A Survey of Monte Carlo Tree Search Methods”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (Mar. 2012), pp. 1–43. ISSN: 1943-068X. DOI: `10.1109/TCIAIG.2012.2186810`.

[2] Guillaume M J B Chaslot, Mark H M Winands, and H Jaap van den Herik. *Parallel Monte-Carlo Tree Search.* 2008.

[3] P. I. Cowling, D. Whitehouse, and E. J. Powley. “Emergent bluffing and inference with Monte Carlo Tree Search”. In: *2015 IEEE Conference on Computational Intelligence and Games (CIG).* Aug. 2015, pp. 114–121. DOI: `10.1109/CIG.2015.7317927`.

[4] Peter I Cowling, Edward J Powley, and Daniel Whitehouse. “Information Set Monte Carlo Tree Search”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 2 (2012).

[5] Kalyanmoy Deb. “An introduction to genetic algorithms”. In: *Sadhana* 24.4 (1999), pp. 293–315. ISSN: 0973-7677. DOI: `10.1007/BF02823145`. URL: `http://dx.doi.org/10.1007/BF02823145`.

[6] M L Ginsberg. “GIB: Imperfect Information in a Computationally Challenging Game”. In: arXiv:1106.0669 (June 2011). URL: `http://cds.cern.ch/record/1356388`.

[7] Levente Kocsis and Csaba Szepesvári. “Bandit based Monte-Carlo Planning”. In: *In: ECML-06. Number 4212 in LNCS.* Springer, 2006, pp. 282–293.

[8] E. J. Lemmon. In: *The Philosophical Review* 74.3 (1965), pp. 381–384. ISSN: 00318108, 15581470. URL: `http://www.jstor.org/stable/2183362`.

[9] Simon M. Lucas et al. “Artificial and Computational Intelligence in Games (Dagstuhl Seminar 12191)”. In: *Dagstuhl Reports* 2.5 (2012). Ed. by Simon M. Lucas et al., pp. 43–70. ISSN: 2192-5283. DOI: `http://dx.doi.org/10.4230/DagRep.2.5.43`. URL: `http://drops.dagstuhl.de/opus/volltexte/2012/3651`.

[10] Simon M. Lucas et al. “Artificial and Computational Intelligence in Games: Integration (Dagstuhl Seminar 15051)”. In: *Dagstuhl Reports* 5.1 (2015). Ed. by Simon M. Lucas et al., pp. 207–242. ISSN: 2192-5283. DOI: `http://dx.doi.org/10.4230/DagRep.5.1.207`. URL: `http://drops.dagstuhl.de/opus/volltexte/2015/5040`.

[11] Marc Ponsen et al. “Bayes-relational Learning of Opponent Models from Incomplete Information in No-limit Poker”. In: *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3.* AAAI’08. Chicago, Illinois: AAAI Press, 2008, pp. 1485–1486. ISBN: 978-1-57735-368-3. URL: `http://dl.acm.org/citation.cfm?id=1620270.1620314`.

[12] Stephan Schiffel and Michael Thielscher. “Automated Theorem Proving for General Game Playing”. In: *Proceedings of the 21st International Jont Conference on Artifical Intelligence.* IJCAI’09. Pasadena, California, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 911–916. URL: `http://dl.acm.org/citation.cfm?id=1661445.1661590`.

[13] Daniel P Taylor. *Investigating Approaches to AI for trust-based, multi-agent board games with imperfect information with Don Eskridge’s ”The Resistance”.* Tech. rep. University of Derby, 2014.

[14] Junxing Wang. *Mastering Non-Perfect Information Games: Resistance And Mafia.* Tech. rep. Carnegie Mellon University, May 2014.

[15] Daniel Whitehouse. “Monte Carlo Tree Search for games with hidden information and uncertainty”. PhD thesis. University of York, 2014.