Resistance Project Report

Simon de Sancha

21503324

Literature Review

Many methods exist for use within the game AI. A select few are discussed below.

Expert Bots

An expert bot is one which uses hard-coded and fairly simple rules to determine its outcome. It will not look into the future nor the past, acting based only on the current situation without persistent round-to-round memory.

The most simple type of bot AI, it can be fairly easily beaten by a more sophisticated AI. My implementation is discussed later.

Bayesian Updates

A bot which uses Bayesian Updating is one whose primary concern is with probabilities. Bayes' theorem is the following:

$$\Pr(\mathrm{A}|\mathrm{X}) = \frac{\Pr(\mathrm{X}|\mathrm{A})\Pr(\mathrm{A})}{\Pr(\mathrm{X})}$$

(Better Explained n.d.)

Where:

- Pr(A | X) is the probability of A given some newly observed 'evidence' X
- Pr(X | A) is the probability of observing the 'evidence' X given A
- Pr(A) the the 'prior' probability - the estimate from before the evidence was observed
- Pr(X) is the probability of the evidence X

In the context of The Resistance, 'A' may be the probability that a certain player is a spy, and the evidence 'X' may be the fact that the last mission the player went on failed. The various probabilities for each player may then be used for a resistance bot to choose whom to nominate for missions, what missions to vote for, etc. Details on my implementation are discussed later.

Monte-Carlo Search Tree

Monte-Carlo Search Tree (MCT) methods are based on simulated game playthroughs. For example, at each position in a game in which a bot needs to make a decision, the bot may simulate multiple playthroughs for each option. The bot needs to have it's own internal game simulation, and as such is mostly used for games which are complete - where the search algorithm has all the necessary information needed to solve the problem.

In the context of The Resistance, the most appropriate use for MCT methods would be in a spy bot - as a spy has no uncertainty over players actions/motives. My implementation is discussed later.

Opponent Modelling

Opponent modelling is based on multiple games with the same opponent, using a machine learning technique such as gradient descent-based learning or more sophisticated techniques such as neural networks or genetic algorithm's (described below).

In the context of the project, opponent modelling would be impractical. To train the agent would require a test bot to versus, and the result would likely be an over-fitted agent that isn't useful in competition with anything but the test bot.

Logic Bots

A logic bot uses a list of logical statements in order to calculate its' actions.

Neural Networks

Neural networks are a family of models inspired by biological neural networks. They typically consist of multiple 'layers' of neural units, with unit being connected to units in the previous layer by some weighting function. The units in the first layer usually represent the input data, and the last layer representing the output. The network is 'trained' to give approximately correct outputs based on training data.

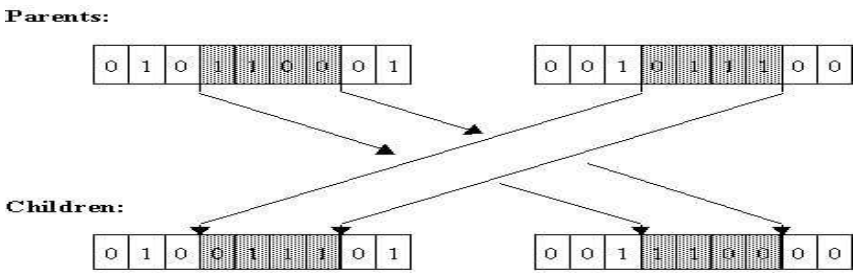
In the context of The Resistance, the input data may be each player's voting record, who they nominated or who was on failed missions, for example. The output may represent each player's spy probability.

Difficulties using this method in The Resistance would revolve around training the network. A set of accurate training data, which would come in the form of game logs, has been hard to find. Rather than use this form of supervised learning, some form of 'neuroevolution' could be used, in which neural networks are combined with genetic algorithms in order to train the network. The only feedback that is needed is a simple measure of the networks performance, which could be the number of times a bot wins, or the sum of the probabilities of players which turned out to be spies. (Lehman & Miikkulainen, 2013)

Genetic Algorithm's

Genetic algorithm's are loosely based on the natural evolution cycle, 'survival of the fittest'. A population of bots is created, with each bot being placed into a tournament, and the best bot from each tournament selected. These 'survivors' get to live to the next generation, and are 'mated' with another survivor, with the children replacing the failed bots from the previous generation. Some bots are 'mutated' to stop the algorithm converging on a local sub-optimal solution.

Each bot is represented by a set of 'genes', usually a simple vector of values. Mutation can be represented by some values being replaced by random numbers. Mating can take many forms, for example, two point crossover occurs when two crossover points are randomly chosen within the gene vector, and the child's genes in between these points are taken from the 'father', with the rest from the 'mother'.



(SlumpTown n.d.)

In the context of The Resistance, genetic algorithm's may be used to optimised other bots, for example, to calculate the optimal probability weightings to use in a Bayesian Updating bot. My implementation is discussed later.

Implementation

In my project I implemented 2 resistance bots and 2 spy bots:

- bayBot: A resistance bot using Bayesian updating
- expertResistanceBot: A resistance bot using simple expert rules
- smartSpy: A spy bot using Monte-Carlo Search Tree
- expertSpyBot: A spy bot using simple expert rules

The most successful bots, bayBot and expertSpyBot, are combined into a single bot - 'firstBot' - using a simple class that passes to either bot depending on whether the player is a spy or not. Similarly the worse bots - expertResistanceBot and smartSpy - are combined into 'secondBot'.

bayBot

For the resistance bot I implemented a Bayesian Updating agent. The rationale behind this decision was as follows:

- Monte-Carlo methods are hard to implement with incomplete information
- Neural networks or opponent modelling may have given a better result, but the difficulties described in Literature Review made the prospect unattractive in this project
- Bayesian updating seemed to give a middle-ground between ease of implementation and expected result.

The main process is as follows:

- Hold a suspicion value for every possible subgroup of players.
- For each round, use Bayesian updates to update our suspicion values for any groups in question
- Use the suspicion values for any nominated groups to determine if they likely contain spies
- If the suspicion values are of no help, revert to simple expert rules

Groups

The first effort made was to maintain suspicion values for each individual player. While this may be of use in certain circumstances, such as if our bot knows for certain a player is a spy, it is not of much use in general. The following situation highlights a major flaw in this method:

1) Players {A,B,C,D,E} start a game, player A is our resistance bot
2) {B,C,D,E} each have initial suspicion value of 0.5
3) {B,C} go on a mission which fails with 1 traitor
4) Assuming there is exactly 1 spy among {B,C}, each will have a suspicion of 0.5, where D and E will have 0.5

The suspicion values have not been updated at all - even though we have gained information about the spies. Even worse is that our bot may nominate {A, B, C} for the next mission - himself plus the next 2 bots with the lowest suspicion. This will clearly fail as we know there is a spy among {B, C}.

This led to suspicion values being assigned to groups, not individuals. In the above example, the suspicion values may be as follows:

{B, C} - 100%
{D, E} - 100%
All else - not yet known.

Now our bot can clearly see not to nominate {B,C} nor {D,E} as each will 100% contain one spy. He may now nominate some other combination {C,D} for example, which may contain from 0 to 2 spies - a much better choice.

The update process after each round is as follows:

$$P(G \text{ contains } N \text{ spies} \mid F \text{ traitors in } G) = P(F \text{ traitors in } G \mid N \text{ spies in } G) * P(N \text{ spies in } G) \div P(F \text{ traitors in } G)$$

Where:

- *G is the group that just failed a mission*
- *F is the number of traitors on the mission*
- $P(N \text{ spies in } G)$ *is the prior estimate*

$$P(F \text{ traitors in } G \mid N \text{ spies in } G) = (N, F) * P^F * (1-P)^{T-F}$$

$$P(F \text{ traitors in } G) = \sum_{N=0}^{T} P(N \text{ spies on mission}) * P(F \text{ traitors in } G \mid N \text{ spies})$$

Where:

- *T is the total number of spies in the game*
- *P is the probability that a spy will betray*
- *(N, F) is the binomial coefficient*

Note:

The above assumes that the number of betrays is a binomial distribution with respect to the number of spies on a mission. This will clearly not be true - if 2 spies are on a mission it is unlikely that both will betray with the same probability as if there was only 1 spy on the mission. For example, 'expertBot' will not betray at all if 2 spies are on the same mission.

Each time a group G fails a mission with F traitors, the following suspicion values are calculated:

- $P(G \text{ contains at least } 1 \text{ spy}) = \sum_{N=0}^{T} P(G \text{ contains } N \text{ spies} \mid F \text{ traitors in } G)$
- $P(U-G \text{ contains at least } 1 \text{ spy}) = \sum_{N=0}^{T} 1 - P(G \text{ contains } T-N \text{ spies} \mid F \text{ traitors in } G)$

  Where U-G are the players not in group G
- $P(A \text{ contains at least } 1 \text{ spy}) \approx P(G \text{ contains at least } 1 \text{ spy})$

  For all $G \subset A$, A supersets of G
- Any groups containing the leader of the failed mission has suspicions incremented by value L_v
- Suspicion values of individual players within group G are calculated as follows:

  $$P(S \text{ is spy}) = \sum_{N=1}^{T} \frac{P(K)*P(J)*P(\text{prior } S \text{ is spy})}{P(N \text{ spies in } G)}$$

  Where P(J) is the probability of N spies in G, given S is a spy, based on the binomial distribution:

  $$P(J) = P(N \text{ spies in } G \mid S \text{ is spy}) * P(F \text{ traitors} \mid N \text{ spies in } G)$$

  And P(K) is the probability of N spies in G calculated earlier.

  The values of players outside of G are calculated similarly.
- Approximate suspicion values of all other possible groups are calculated very crudely - simply the sum of each player's suspicion values. While this is certainly suboptimal, it returned better results than when compared to a more sophisticated method, for example, for the group H={A,B,C}

  $$P(\text{at least } 1 \text{ spy in } H) = \sum P(A \text{ is spy}) * P(B \text{ is resistance}) * P(C \text{ is resistance})$$

  Which is summed over all possible spy/resistance combinations which result in at least one spy in H.

The probability that a spy will betray - 'P' in the above figures - was calculated experimentally using genetic algorithms ('GeneticAlgorithm') training against expertSpyBot. A gene set of 5 values between 0 and 1 were used. Each represented the probability that a spy would betray on the corresponding mission. A 6th gene was used to represent L_v.

The genetic algorithm was run with a population of 100 bots, with 20 surviving each generation. Mating was done via one-point crossover. A mutation rate of 0.05 was used.

The fitness function was calculated by running 100 games consisting of 2 expertSpyBot's, and 3 identical bayBot's playing with the genes given. The fitness function then returns the number of wins for the resistance. A large number of games (100) was required to average out any randomness within the game - for example, in which order the spies are arranged, which agent gets to nominate first, etc.

Tournament selection was done from a set of 3 bots - with the fittest chosen from the 3 using the above fitness function.

This arrangement seemed to give the best results. Another possible fitness function was one which would pit all 3 different bots from the gene pool against 3 expertSpyBots, with the survivor being the one with the most accurate suspicion values. This did not produce useful results.

The GA quickly converges to an answer within ~20-30 generations. The results are:

0.94, 0.93, 0.99, 0.91, 0.62, 0.06

The first 5 values are the betrayal probability, and the 6th is the mission leader penalty. The 4th and 5th values had very poor convergence - the 4th being between 0.8-1 and the 5th being almost random. This result makes complete sense; the suspicion updates from after the 4th game are unlikely to heavily impact the result, with the 5th being totally useless. The low value for the leader penalty (0.06) is interesting, and is discussed later.

smartSpy

'smartSpy' implements a Monte-Carlo Search Tree to determine the optimal move out of a set of options. At each decision smartSpy needs to make, a number of playthroughs are simulated for each decision option; for example if a spy needs to decide whether to betray a mission, it will do a number of playthroughs in which he either votes 'betray' or 'not betray'. The option with the greater number of successful playthroughs is chosen.

The MCT search success is predicated on it's internal simulation. In my implementation, any spies are represented with 'expertSpyBots', and any resistance players are represented by 'bayBots'. In nominating a team, it follows the simple expert rules - nominating itself and a random selection of resistance members.

The MCT will have a branching factor of 2 - each node representing a smartSpy decision on whether to vote, or whether to betray, both of which are yes/no questions. The maximum depth would occur when every nomination takes the maximum 5 voting rounds, with a maximum of 5 missions, each containing the spy; giving a theoretical maximum depth of 30.

Due to time constraints the implementation of smartSpy needed to use a shortcut at the detriment of performance. Rather than performing a true tree search on each playthrough, a randomly chosen path is used for each playthrough. In comparison, in a depth-first search, which would be the optimal search technique (or it's iterative deepening variant), at each node the leftmost child would be explored, and once the search has returned only then the right child would be explored. In my implementation, upon reaching each node, a random child is explored. This is done as many times as possible within the time limit, and the decision option which leads to the most wins is used.

expertSpyBot, expertResistanceBot

A bot using simple expert rules, given below:

Spy rules:

- When nominating, include only himself and resistance members
- Approve missions if and only if they contain one spy
- Only betray a mission if itself is the only spy on the mission
- Betray any mission which will win the game for the spies

Resistance rules:

- When nominating, select itself and a random selection of others

- Reject missions of 3 not featuring self
- Always approve the first mission

Validation

Results from each bot competing against each other are given below. A modified Game.java was created which allowed the adding of Agent's specifically as spies or resistance. Each result is an average over 100 to 1000 games.

	bayBot	expertResistanceBot
smartSpy	~50%	99%
expertSpyBot	65%	100%

Percentage of spy wins

Reflection

The poor performance of smartSpy is quite clearly a reflection of the incomplete/poor MCT implementation.

The performance of bayBot likely could be improved by the following:

- Improving the calculation of betrayals based on number of spies on a mission, i.e.

  $$P(F \text{ traitors in } G \mid N \text{ spies in } G)$$

  The binomial distribution currently implemented is likely a poor approximation.

- Improving the suspicion approximation of groups unrelated to the one which went on the previous mission. As stated above the method used in bayBot is very crude, simply the sum of each player's individual suspicion value. A more sophisticated technique would likely give a better result.

- Improvements on the penalty applied to the leader of a failed mission. The very low value returned from the genetic algorithm - 0.06 - indicates a flawed method; especially when the expertSpyBot used to train against always leads a failed mission.

Bibliography

Better Explained n.d., *An Intuitive (and Short) Explanation of Bayes' Theorem*. Available from: https://betterexplained.com/articles/an-intuitive-and-short-explanation-of-bayes-theorem/ [25 October 2016]

Buckland, M., 2002. *AI Techniques for Game Programming*. Premier Publishing.

Lehman, J. and Miikkulainen, R., 2013. *Neuroevolution*, Available from: http://www.scholarpedia.org/article/Neuroevolution [25 October 2016]

SlumpTown n.d., *Chapter 2: Literature Review*. Available from: http://www.stumptown.com/diss/chapter2.html [25 October 2016]