***NARS in Table-Top Games. Avalon: The Resistance***

Jamison Peake, Department of Computer Science, Temple University

**Project Outline**

 Avalon: The Resistance is a five to ten player game that deals with team play, logical deductions, strategy, and deception; players can say anything at any time to convince you of their loyalty. My first test case was that of a most basic game: five players without any special characters involved. There are three ‘good’ players and two ‘bad’ players, and while the bad players know of each other, the good players are left in the dark. Through five events called “Quests”, either the good will prevail by gaining 3 successes or the bad will win by gaining 3 failures. The number of players a Quest requires (in order) are 2, 3, 2, 3, and 3. When a good player is on a Quest they need to pick a Success card, although a bad player can pick Success or Failure card. These cards are then shuffled and revealed, if there is one failure among them then the Quest fails. Players are selected for these Quests by Proposals which are voted upon by the group. Majority in acceptance means the proposal happens, but if the majority rejects then the proposal moves clockwise to the next player. For any further clarification on rules you can [visit here](http://www.terencel.com/docs/TR/The_Resistance_Avalon_Rules%28EN%29.pdf) for an online PDF version of the rule book. As far as generalized goals, first NARS should be able to watch a game and figure who is good/bad as an observer. Then NARS should be able to simulate a human playing the game efficiently enough to win.

**Project Stages**

 There are two ways to go about using NARS for this process. However it was narrowed to one choice with the current limitation of NARS. It is currently unable to process mass amounts of input so I will have to control the input NARS receives and record the output to use when necessary. Because of this my project is more of a recorder of the game than a player; it is currently logical intelligence rather than artificial human intelligence. It is important to recognize that this modification will not necessary when NARS can handle the input of an entire game.

 If NARS could watch an entire game then the game rules and logic rules I installed would remain absolute since nothing would contradict them. However, the frequency and belief NARS assigns to the logic rules would vary as it observes more games rather than being my fixed values as in the current method. This is what would result in an actual artificial intelligence along with the fact that the logic steps it uses would not be at absolute efficiency. Time control would also be there for proposals and quests which would help guide NARS’ thinking progress with the game.

**Avalon: The Resistance Logic Rules**

 This took up the main part of the project. First I had to dissect Avalon: The Resistance into a set of logic rules. Then I had to learn much of the language of NARS through trial and error with the different logic rules I was employing. It would take nearly a book to list all the rules in their entirety, however I shall show the main ideas in hopes of helping others understand the language more quickly than I did:

|  |
| --- |
| //Grouping many of my variables together to make complex statements simpler later on//Proposal# on Quest1 are Proposals<Q1Proposal1 --> Proposals>.<Q1Proposal2 --> Proposals>.<Q1Proposal3 --> Proposals>.<Q1Proposal4 --> Proposals>.<Q1Proposal5 --> Proposals>. |
| //If something is Bad then it cannot be Good and vice versa<<$1 --> Bad> <=> (--,<$1 --> Good>)>. <<$1 --> Good> <=> (--,<$1 --> Bad>)>.  |
| //If Player1, Player2, and Player 3 are Good then Player4 and Player5 are Bad <(&&, <{Player1} --> Good>, <{Player2} --> Good>, <{Player3} --> Good>) ==> (&&,<{Player4} --> Bad>, <{Player5} --> Bad>)>. |
| //If somebody Thinks themself as Bad then that person is Bad <<(\*, $1, <$2 --> Bad>) --> Thinks> ==> (--, <(\*, $1, <$2 --> Good>) --> Thinks>)>. |
| //If somebody is Good then Players Shouldknow that somebody is good <<$1 --> Good> ==> <(\*, Player, <$1 --> Good>) --> Shouldknow>>. |
| //Somebody Approves Something <(\*, $5, $4) --> Approve> |
| //There are two people on Quest1 which is part of some Proposal <(&&, <$1--> Q1>, <$2 -->Q1>) --> Proposal> |
| //If a Player approves a Proposal they Think the players on the Proposal are Good<(&&, <(&&, <$1--> Q1>, <$2 -->Q1>) --> $4>, <(\*, $5, $4) --> Approve>) ==> (&&,<(\*,$5, <$1 --> Good>)--> Thinks>, <(\*, $5, <$2 --> Good>)--> Thinks>)>.%1.00;0.40% |
| //If a Player Approves/Rejects something it gains the App/Rej property respectively<<(\*, Player, $1) --> Approve> ==> <$1 --> [App1]>>.<(&&, <(\*, Player, $1) --> Approve>, <$1 --> [App1]>) ==> <$1 --> [App2]>>.<(&&, <(\*, Player, $1) --> Approve>, <$1 --> [App2]>) ==> <$1 --> [App3]>>.<(&&, <(\*, Player, $1) --> Approve>, <$1 --> [App3]>) ==> <$1 --> [App4]>>.<(&&, <(\*, Player, $1) --> Approve>, <$1 --> [App4]>) ==> <$1 --> [App5]>>.<<(\*, Player, $1) --> Reject> ==> <$1 --> [Rej1]>>.<(&&, <(\*, Player, $1) --> Reject>, <$1 --> [Rej1]>) ==> <$1 --> [Rej2]>>.<(&&, <(\*, Player, $1) --> Reject>, <$1 --> [Rej2]>) ==> <$1 --> [Rej3]>>.<(&&, <(\*, Player, $1) --> Reject>, <$1 --> [Rej3]>) ==> <$1 --> [Rej4]>>.<(&&, <(\*, Player, $1) --> Reject>, <$1 --> [Rej4]>) ==> <$1 --> [Rej5]>>. |
| //Three or more Rej or App results in T.Rej or T.App<(&&,<$1 --> [App3]>, <$1 --> Proposal>) ==> <$1 --> [T.App]>>.<(&&,<$1 --> [Rej3]>, <$1 --> Proposal>) ==> <$1 --> [T.Rej]>>. |
| //If a Proposal gains T.App then that proposal happens<(&&, <(&&,<$1 --> Q1>,<$2 --> Q1>) --> $4>, <$4 --> Proposals>, <$4 --> [T.App]>) ==>(&&,<$1--> Q1>,<$2--> Q1>)>. |
| //If a quest is good the players on it are good<(&&,<$1 --> Q1>, <$2 --> Q1>, <$3 --> S1>) ==> (&&, <$1 --> Good>, <$2 --> Good>)>. |
| //If somebody is Bad it implies that somebody Needs F3 (3 Failures)<<$1 --> Bad> ==> <(\*, $1, F3) --> Needs>>. |
| //If somebody Needs F3 it implies they Need the Quest1 to be a failure (low confidence)<<(\*, $1, F3) --> Needs> ==> <(\*, $1, <Q1 --> F1>)--> Needs>>.%1.00;0.40% |
| //If somebody Needs F3 and the first quest was a success it implies they Need Quest2 to fail<(&&, <(\*, $1, F3) --> Needs>, <Q1 --> S1>) ==> <(\*, $1, <Q2 --> F1>) -->Needs>>.%1.00;0.80% |
| //If somebody Needs F3 and the second quest succeeded it implies they Need Quest3 to fail<(&&, <(\*, $1, F3) --> Needs>, <Q2 --> S2>) ==> <(\*, $1, <Q3 --> F1>) -->Needs>>.%1.00;0.99% |
| //If somebody Needs Quest1 to failure, that somebody Approves a Proposal for Quest1, and there are two people on that proposal ($2 and $3) then at least one of those people are bad<(&&, <(\*, $1, <Q1 --> F1>) --> Needs>, <(&&, <$2 --> Q1>, <$3 --> Q1>) --> $4>, <(\*, $1, $4) --> Approve>) ==> (||, (&&, <$2 --> Good>, <$3--> Bad>), (&&, <$2 --> Bad>, <$3-->Good>))>. |

 I then found the most efficient way of inputting my logic into NARS and found a general outline for voting logic and quest outcome logic. Calling these separation “Parts”, a long game could have upwards of 20 parts and calculating just one part would take nearly a day when writing it all out. For this purpose I created a Java program that I would give the data from the game and it would output NARS code. I’d copy and paste this code into NARS and then give my program NARS’ output which it would write in some file that it could read from when needing to use previous output. This shortened a Part from nearly a day to about an hour, which is a considerable difference but not yet near real time. My controlled logic was highly dependent on three main ideas: what the computer knows from observing (anything besides “Thinks” and “ShouldKnow” refer to this), what a player thinks (i.e. how a player is voting, who they accuse, etc), and what a player should know (compute this by running through every logic step the computer performed while observing the game again, only now the computer is assuming that player is good). Any contradictions, patterns, or reinforcements between these three types of knowledge indicate something that the computer would pick up, interpret, and assign a good or bad probability to the person performing the action. My code went as followed (if not otherwise specified, it is an output code to give to NARS and then accepts NARS’ output):

Figure 1. Controlled Logic Steps

The “\*” from this refer to general logic loops, which go as follow:

Figure 2. Logic Loop

 The “1” is the first arrow you take and then the “2” is the second arrow. This entire loop happens twice, the first run through is doing the logic for that Part, the next time through is that logic overall. “Probability” here means using probability based off the fact that there are only 3 good players and 2 bad players; “Logic of Bad Wanting to Win” means that if the computer decide a person was bad then it would go back through and use that players allegiance and past voting/proposals/quest outcome styles to determine the allegiances of other players. It cannot do this with good players as good players do not know of any other good players so their actions cannot be relied on.

**Game Input**

 Once again it would take nearly a tree to show all of the game input, so I will show the first part, whatever is in a box is what I gave to NARS from my program and NARS’ output, italics is my otherwise input, and regular text is my programs otherwise output:

Name each player:

*Player1*

*Player2*

*Player3*

*Player4*

*Player5*

Who proposed this quest?

*Player1*

Who is on this proposal?

*Player1*

*Player5*

Did Player1 Approve or Reject Q1Proposal1?

*Approve*

Did Player2 Approve or Reject Q1Proposal1?

*Reject*

Did Player3 Approve or Reject Q1Proposal1?

*Approve*

Did Player4 Approve or Reject Q1Proposal1?

*Approve*

Did Player5 Approve or Reject Q1Proposal1?

*Approve*

Here are your lines of code for what Player1 thinks for this part:

|  |
| --- |
| \*\*\* IN: <(\*,{Player1},Q1Proposal1) --> Approve>. %1.00;0.99% {0 : 1}  IN: <(&&,<{Player1} --> Q1>,<{Player5} --> Q1>) --> Q1Proposal1>. %1.00;0.99% {0 : 2}  IN: <(&&,<(\*,$1,$2) --> Approve>,<(&&,<$3 --> Q1>,<$4 --> Q1>) --> $2>) ==> (&&,<(\*,$1,<$3 –-> Good>) --> Thinks>,<(\*,$1,<$4 --> Good>) --> Thinks>)>. %1.00;0.40% {0 : 3}  IN: (&&,<(\*,{Player1},<{Player1} --> Good>) --> Thinks>,<(\*,{Player1},<{Player5} --> Good>) --> Thinks>)? {0 : 4}  IN: <(\*,{Player1},<{Player1} --> Good>) --> Thinks>? {0 : 5}  IN: <(\*,{Player1},<{Player5} --> Good>) --> Thinks>? {0 : 6} 82 OUT: (&&,<(\*,{Player1},<{Player1} --> Good>) --> Thinks>,<(\*,{Player1},<{Player5} --> Good>) –> Thinks>). %1.00;0.40% {81 : 1;2;3} 452 OUT: <(\*,{Player1},<{Player1} --> Good>) --> Thinks>. %1.00;0.36% {533 : 1;2;3} 172 OUT: <(\*,{Player1},<{Player5} --> Good>) --> Thinks>. %1.00;0.36% {705 : 1;2;3}  |

Enter the data NARS gave or enter E to stop entering data:

*<(\*,{Player1},<{Player1} --> Good>) --> Thinks>. %1.00;0.36%*

*<(\*,{Player1},<{Player5} --> Good>) --> Thinks>. %1.00;0.36%*

*E*

Those lines were written to Player1ThinksP.txt

Here are your lines of code for what Player2 thinks for this part:

|  |
| --- |
| \*\*\* IN: <(\*,{Player2},Q1Proposal1) --> Reject>. %1.00;0.99% {0 : 1}  IN: <(&&,<{Player1} --> Q1>,<{Player5} --> Q1>) --> Q1Proposal1>. %1.00;0.99% {0 : 2}  IN: <(&&,<(\*,$1,$2) --> Reject>,<(&&,<$3 --> Q1>,<$4 --> Q1>) --> $2>) ==> (||,(--,<(\*,$1,<$3 –> Good>) --> Thinks>),(--,<(\*,$1,<$4 --> Good>) --> Thinks>))>. %1.00;0.40% {0 : 3}  IN: (||,(--,<(\*,{Player2},<{Player1} --> Good>) --> Thinks>),(--,<(\*,{Player2},<{Player5} --> Good>) --> Thinks>))? {0 : 4} 100 OUT: (||,(--,<(\*,{Player2},<{Player1} --> Good>) --> Thinks>),(--,<(\*,{Player2},<{Player5} --> Good>) --> Thinks>)). %1.00;0.40% {99 : 3;2;1}  |

Enter the data NARS gave or enter E to stop entering data:

*(||,(--,<(\*,{Player2},<{Player1} --> Good>) --> Thinks>),(--,<(\*,{Player2},<{Player5} --> Good>) --> Thinks>)). %1.00;0.40%*

*E*

Those lines were written to Player2ThinksP.txt

*…This continues on, getting as complicated as….*

Here is what Player1 should know:

 <(\*,{Player1},<{Player2} --> Good>) --> Shouldknow>. %0.00;0.57%

 <(\*,{Player1},<{Player5} --> Good>) --> Shouldknow>. %0.88;0.58%

 <(\*,{Player1},<{Player4} --> Good>) --> Shouldknow>. %0.73;0.48%

 <(\*,{Player1},<{Player3} --> Good>) --> Shouldknow>. %0.73;0.48%

 <(\*,{Player1},<{Player1} --> Good>) --> Shouldknow>. %1.00;0.99%

Which player number should Player1 know most likely to be bad or 0 if none:

*2*

Here is your data for bad players wanting to win for this part:

|  |
| --- |
| \*\*\* IN: <(\*,{Player1},<{Player2} --> Good>) --> Shouldknow>. %0.00;0.57% {0 : 1}  IN: <(\*,{Player1},<{Player5} --> Good>) --> Shouldknow>. %0.88;0.58% {0 : 2}  IN: <(\*,{Player1},<{Player4} --> Good>) --> Shouldknow>. %0.73;0.48% {0 : 3}  IN: <(\*,{Player1},<{Player3} --> Good>) --> Shouldknow>. %0.73;0.48% {0 : 4}  IN: <(\*,{Player1},<{Player1} --> Good>) --> Shouldknow>. %1.00;0.99% {0 : 5}  IN: <<(\*,{Player1},<$1 --> Good>) --> Shouldknow> <=> (--,<(\*,{Player1},<$1 --> Bad>) –> Shouldknow>)>. %1.00;0.99% {0 : 6}  IN: (--,<(\*,{Player1},<{Player2} --> Bad>) --> Shouldknow>)? {0 : 7}  IN: <(\*,{Player1},<{Player2} --> Bad>) --> Shouldknow>? {0 : 8} 2 OUT: (--,<(\*,{Player1},<{Player2} --> Bad>) --> Shouldknow>). %0.00;0.57% {1 : 1;6} 35 OUT: <(\*,{Player1},<{Player2} --> Bad>) --> Shouldknow>. %1.00;0.57% {36 : 1;6}  |

What is NARS output:

*<(\*,{Player1},<{Player2} --> Bad>) --> Shouldknow>. %1.00;0.57%*

Further codes!

|  |
| --- |
| \*\*\* IN: <(\*,{Player1},<{Player2} --> Bad>) --> Shouldknow>. %1.00;0.57% {0 : 1}  IN: <<(\*,{Player1},<$1 --> Bad>) --> Shouldknow> ==> <(\*,{Player1},<(\*,$1,F3) --> Needs>) --> Shouldknow>>. %1.00;0.99% {0 : 2}  IN: <(\*,{Player1},<(\*,{Player2},F3) --> Needs>) --> Shouldknow>? {0 : 3} 9 OUT: <(\*,{Player1},<(\*,{Player2},F3) --> Needs>) --> Shouldknow>. %1.00;0.57% {8 : 1;2}  |

What is NARS output or E to stop:

*<(\*,{Player1},<(\*,{Player2},F3) --> Needs>) --> Shouldknow>. %1.00;0.57%*

*E*

Here are your lines of code for what players Player1 should know are good based on bad players wanting to win:

|  |
| --- |
| \*\*\* IN: <(\*,{Player1},<(\*,{Player2},F3) --> Needs>) --> Shouldknow>. %1.00;0.57% {0 : 1}  IN: <<(\*,{Player1},<(\*,$1,F3) --> Needs>) --> Shouldknow> ==> <(\*,{Player1},<(\*,$1,<Q1 --> [Fail]>) --> Needs>) --> Shouldknow>>. %1.00;0.40% {0 : 2}  IN: <(\*,{Player1},<(\*,{Player2},<Q1 --> [Fail]>) --> Needs>) --> Shouldknow>? {0 : 3} 15 OUT: <(\*,{Player1},<(\*,{Player2},<Q1 --> [Fail]>) --> Needs>) --> Shouldknow>. %1.00;0.23% {14 : 1;2}  |

What is NARS output or E to stop:

*<(\*,{Player1},<(\*,{Player2},<Q1 --> [Fail]>) --> Needs>) --> Shouldknow>. %1.00;0.23%*

*E*

Further codes!

|  |
| --- |
| \*\*\* IN: <(\*,{Player1},<(\*,{Player2},<Q1 --> [Fail]>) --> Needs>) --> Shouldknow>. %1.00;0.23% {0 : 1}  IN: <(\*,{Player2},Q1Proposal1) --> Reject>. %1.00;0.99% {0 : 2}  IN: <(&&,<{Player1} --> Q1>,<{Player5} --> Q1>) --> Q1Proposal1>. %1.00;0.99% {0 : 3}  IN: <(&&,<(\*,$1,$2) --> Reject>,<(&&,<$3 --> Q1>,<$4 --> Q1>) --> $2>,<(\*,{Player1},<(\*,$1,<Q1 --> [Fail]>) --> Needs>) --> Shouldknow>) ==> (&&,<(\*,{Player1},<$3 --> Good>) --> Shouldknow>,<(\*,{Player1},<$4 --> Good>) --> Shouldknow>)>. %1.00;0.99% {0 : 4}  IN: (&&,<(\*,{Player1},<{Player1} --> Good>) --> Shouldknow>,<(\*,{Player1},<{Player5} --> Good>) --> Shouldknow>)? {0 : 5}  IN: <(\*,{Player1},<{Player1} --> Good>) --> Shouldknow>? {0 : 6}  IN: <(\*,{Player1},<{Player5} --> Good>) --> Shouldknow>? {0 : 7} 147 OUT: (&&,<(\*,{Player1},<{Player1} --> Good>) --> Shouldknow>,<(\*,{Player1},<{Player5} --> Good>) --> Shouldknow>). %1.00;0.23% {146 : 4;1;3;2} 103 OUT: <(\*,{Player1},<{Player1} --> Good>) --> Shouldknow>. %1.00;0.21% {249 : 4;1;3;2} 488 OUT: <(\*,{Player1},<{Player5} --> Good>) --> Shouldknow>. %1.00;0.21% {727 : 2;1;3;4}  |

What is NARS output or E to stop:

*<(\*,{Player1},<{Player1} --> Good>) --> Shouldknow>. %1.00;0.21%*

*<(\*,{Player1},<{Player5} --> Good>) --> Shouldknow>. %1.00;0.21%*

*E*

Those lines were written to Player1ShouldknowP.txt

**Project Results**



Table I. Parts 1 and 2 Results

 Because of the enormous amount of time spent per part, I only recorded the first couple parts and merely went through the further parts to check that my program’s code was working. Knowing that the logic rules I installed work, and since I am controlling everything, I was able to assume that the trend of NARS figuring out good and bad players only improves by running through approximations of what the logic rules would yield.

 I have been unable simulate a human player in Avalon: The Resistance with NARS as of now. I have made parameters on how NARS should act depending on which character and allegiance it is assigned. However, since my usage of NARS creates logical intelligence, not artificial human intelligence, the game play will never be varied and thus give away which character it is every game to the other players. NARS will also have to have time copulas integrated so it can recognize there is only so much it can do and then has to wait on more input.

**Project Problems**

 The obvious one that I had to work around (unavoidable for how young NARS is) is its inability to process large amounts of data and simply learning the NARS language. As I mentioned before, I learned much of what I know through trial and error with my logic codes and also from looking at different examples of codes people had done in the past. Also I was unable to use true probability; this is possible in NARS but I would have had to spend another half of a semester writing the rules of scalar quantities. This is understandable though as true probability is a concept learned, not a method used for learning. Another problem was there was some difficulty with NARS processing complex higher-order statements which will mostly be address in the next update.

 A problem that only I will probably encounter is NARS’ refusal to accept a complete belief in any concept. The rules of the game would need absolute belief as they are what we operating under in a controlled environment. I overcome this one with my self control; NARS did not have to process any game rules, only the logic. Finally, the time inefficiency of the method was a huge problem. It was shortened from a day per part to an hour per part but still inefficient to compute when a game can have upwards of twenty parts. This is something that I cannot fix until either my first problem was fixed or there becomes some quicker way to communicate between programs with NARS.

**Project’s Future**

 As stated previously, I will not be able to continue on with this specific idea until NARS is becomes compatible with other written programs or in years down the line when NARS is able to handle large amounts of input at once, along with time copulas being implemented. When this is done I would greatly enjoy being able to confirm the results of NARS observing a 5 player and 7 player game. Also if I wish to continue as far as implementing NARS as a human player then that would most likely take another semester of research.

 The way I proceeded about this project does not allow it to be applied to many other projects, however the codes I used for what a player should know and what a player thinks has great potential in strategy games for AI. Companies are quickly having AI technology replace otherwise logic based codes. The reasoning behind this is that no matter how strategic a game is, unless a game is as simple and limited as checkers, the game will come down to more than straight logic of the best move. Understanding this concept is what separates avid strategists and otherwise normal players, along with separating logical intelligence systems and the potential of the AI game systems. An example of this comes from a [Go board game](http://en.wikipedia.org/wiki/Go_%28game%29). Some may consider its form limited because there is only one type of game piece, but the number of combinations with those pieces are enormous, having more possible board states than there are atoms in the universe. However, everybody has a characteristic choice; in the subconscious there are methods of attack one prefers and others one dislikes. This naturally shapes one’s game play. In experts it may be so fleeting that it cannot even be called a pattern, but recognizing that flow of game play, which attack an opponent will choose from the numerous branches of techniques, will allow one player to control the game. This is how strategists win games, most people are so caught up in winning that they do not notice that winning is only a byproduct of the entire game. It is always board state (i.e. controlling it) that results in winning and that is what one should strive for, controlling your opponent rather winning. A logical intelligence game will only strive to win a game, while AI can strive to understand your opponent and defeat that specific opponent. Although the codes of “ShouldKnow” and “Thinks” were just a byproduct of my project, I believe them along with some true probability code has great potential in this field that can create an A.I. program capable of picking up and excelling in any strategy game.