UNIVERSITY OF BERGEN

MASTER THESIS

Department of Information Science and Media Studies

Reasoning about Knowledge and Action in Cluedo using Prolog

Author:
VEMUND INNVÆR AARTUN
Supervisor:
THOMAS ÅGOTNES

June 1, 2016

Abstract:

This thesis will look at how to define a representation for the implementation of Cluedo and dynamic epistemic logic into Prolog. It looks at basics of modal logic, and definitions of dynamic epistemic logic. The thesis goes through the implementation in great detail, and shows a suggestion for how to best represent the game world of Cluedo. The thesis will look at how an agent using modal logic will compare against one that does not. And try to find out if Prolog is a programming language that is suitable for this kind of implementation.

Acknowledgements:

I would like to express my gratitude to everyone who have helped me through this master thesis. Without the support and motivation from friends and family this would never be possible. A special thanks goes to:

My parents Marit and Bjarte Aartun for their undying support, and encouragement.

To Kristoffer Kjøsnes Kongsvik for believing in me, when I did not.

And finally to my supervisor Thomas Ågotnes, for giving me the idea for the thesis, and helping me finish it.

Table of Contents:

1 Introduction **9**
1.1 Research questions 9
1.2 Methodology 10
1.3 Thesis structure. 10

2 Theory: **11**
2.1 Cluedo 11
2.2 Cluedo article 12
2.3 Dynamic epistemic logic 14
2.3.1 Basics expressions of modal logic 15
2.4 Definitions of dynamic epistemic logic 15
2.4.1 Definition 1(Dynamic epistemic logic - LA) 16
2.4.2 Definition 2 (Knowledge actions - KAA) 16
2.4.3 Definition 3 (Semantics of LA) 17
2.6 Implementation of AI in games. 19
2.6.1 The environment of Cluedo.
20

3 Development **22**
3.1 Development tools
23
3.1.1 Prolog 23
3.1.2 Games in prolog 24
3.2 Implementation of the game mechanics 24
3.2.1 Mental representation of the agent 24
3.2.2 Kripke 26
3.2.3 cluedo.pl 27
3.2.4 dealer.pl 28
3.2.5 subtracter.pl 33
3.2.6 query.pl 35
3.2.7 unity.pl 38
3.3 Implementation of dynamic epistemic logic 43
3.3.1 First out is showcard/3 43
3.3.2 Now let’s look at leggetil/3 45
3.3.2 Now let’s look at the nocard/3 predicate 49
3.3.3 Analyzer/5 51
3.3.3.1 anSC/5 55
3.3.3.2 Now let’s look at assemble/4 57
3.3.3.3 remover/5 58
3.4 Implementing modal.pl into untiy.pl. 61

4 Discussion: 64
4.1 Cluedo mechanics
64
4.2 Definition 1 - Language 65
4.3 Definition 2 - Knowledge actions. 66
4.3.1 Nocard 66
4.3.2 Showcard 67
4.3.3 Endmove and win 68
4.4.1 Analyzer 69
4.4 Comparing agent using modal logic vs not. 69
4.5 Prolog 71
4.6 Strategies and further development 71
5 Conclusion 74

References 76

Appendix A - Query of player(6). before modal.pl
Appendix B - Query of player(6). after modal.pl
Appendix C - cluedo.pl
Appendix D - dealer.pl
Appendix E - query.pl
Appendix F - subtracter.pl
Appendix G - unity.pl
Appendix H - modal.pl

1 Introduction:

Information-carrying events, such as a message or a request in an electronic social medium, can lead to complex changes of belief- or knowledge states of the involved agents. The field of epistemic and doxastic logic develops frameworks for formalising such states and for analysing their dynamics [3, 7]. Knowledge dynamics have often subtle and sometimes non-intuitive properties. An example is the so-called Moore sentence, “p is true but you don't know it", which illustrates that a fact can become false immediately after it was communicated even if it was true immediately before. Knowledge dynamics become even more interesting, and subtle, in strategic settings, where the agents have, often competing, goals involving each other's belief- and knowledge states, such as “Ann knows my secret but Bob does not". Strategic interaction is analysed in the field of game theory.

However, knowledge games, games where actions are described in the terms of their informational content and players have preferences over knowledge states, is an emerging research area in the intersection of epistemic logic and game theory [5, 1]. Knowledge games are generally games where the shared knowledge of all the players make up a knowledge base, but only some of the knowledge is know to each of the players. Some knowledge is shared, and the endgame is to acquire a specific set of knowledge that is unknown to all the players in the beginning of the game. Each player has a prefered knowledge state that changes as the game progresses, until they have certainty of having the winning knowledge state. In order to get the right knowledge state the player must be able to reason with the knowledge they are gradually gaining more of.

1.1 Research questions:

It turns out that many parlour games such as card or board games are useful case studies of modeling frameworks, because they involved precisely defined information-carrying actions and goals. One such example is the board game Cluedo, which has proved to be a useful case study in the development of dynamic epistemic logic [6]. While dynamic epistemic logics are beginning to be well understood theoretically, very few computer implementations exist. Such implementations would be useful for a number of reasons. One of them is that researchers need practical tools to develop and test new theoretical conjectures. The topic of the masters project is the implementation of dynamic epistemic logics, in particular for reasoning about Cluedo, in Prolog. In particular, the goal of the thesis is to answer the following research questions:

- Is Prolog suitable for implementations of dynamic epistemic logic[4]?
- How can Cluedo's knowledge actions, formalised in dynamic epistemic logic, be implemented in Prolog?
- Is Prolog suitable for implementations of Cluedo game mechanics?
- How does an agent using dynamic epistemic logic compare to a navie agent that does not?

1.2 Methodology:

The purpose of this thesis is to implement and test the functionality of dynamic epistemic logic, and the game mechanics of Cluedo in prolog. In order to do this, we must first get the proper theoretical basis to do so. This is done using literature that directly covers the topic of the thesis. Which is dynamic epistemic logic connected to Cluedo, but also the rules and environment of Cluedo. After the theory is learned. We will start implementing the game mechanics and the logic into Prolog. To do this we must find ways to represent the logic in prolog. After this is done, we use the program to compare the efficiency of the agents.

1.3 Thesis structure.

This thesis will start out by looking at the relevant theory and rules for Cluedo, and dynamic epistemic logic in chapter 2. We will look at the rules and mechanics of Cluedo, and at relevant literature for Modal and dynamic epistemic logic. In chapter 3, the development process of the program will be looked at in detail. First at the development of the Cluedo mechanics, then at the development of the modal logic. In chapter 4 we will revisit the theory, and compare it to the implementation. We will also briefly look at possible strategies for Cluedo, and at possibilities for further development for the program. The thesis ends with the conclusion in chapter 5

.

2 Theory:

2.1 Cluedo:

Cluedo is a board game that first came out in 1949, it was originally created by Anthony E. Pratt but have since had various remakes and spinoffs.[13] The game is played with two to six players and is centered around a murder mystery that the player will have to try to solve. Cluedo is a knowledge game. And like other knowledge games it is a competitive game, that seeks to gain the knowledge necessary to win in the least amount of time.

At the start of the game it is announced that Dr. Black has been found murdered in the stairway. The body has been moved since the murder, and it is clear that Dr. Black has been murdered in one of the nine rooms in his mansion, with one of six potential weapons, by one of the six guests. Each of the rooms, weapons and persons are represented by a card in the game, which gives 21 unique cards. One random card from each category is put in an envelope and placed in the middle of the board. The rest of the cards are evenly distributed between the players.

In order to figure out which cards are in the envelope, players must call out suggestions of three cards that might be in the envelope. Each of the players call out their suggestion when it is their turn every round. The the player must roll the dice and try to get to a room on the board, one square on the board represent one step from the number of steps given by the dice. It the player cannot reach a room, they cannot call for a suggestion, and must end their turn, if they do reach the room the player can call out his suggestion.

This is done by calling out the following. “I suggest that X, used Y to murder in the Z”. Where X must be a person card, Y must be a weapon card, and Z must be the room the player is currently in. If the player to the right has any of the cards X, Y or Z they have to show one of them to the player whose turn it is, and he has to decide if he wants to end the round or make an accusation. If the player to the right does not have any of the cards X, Y or Z he has to announce that to everyone, and the query moves to the next player. This goes on until one and only one of the cards mentioned has been shown or all players have been queried.

If the player decide to make an accusation he must state to all players the cards he thinks are in the envelope. One room, one person and one weapon. He will then open the envelope and see if the cards in the envelope are also the cards he has accused them of being. If he is correct he will have won the game, and the game ends. Each player can only accuse one time, so if he is wrong, he must place the cards back in the envelope without showing the other players the cards, and can no longer win the game. The player must however stay in the game and answer the other players suggestions, until one of the players get the accusation right.

After every player has called out their suggestion and gotten answers, we say that we are done with round 1, and round 2 starts. The game goes on for however many rounds it takes for a player to get an accusation right, or until all players have gotten the accusation wrong.

2.2 Cluedo article:

One of the main sources of literature for this thesis comes from Hans P. van Ditmarsch's case study "*The description of game actions in Cluedo".* Which was first published as a submission to "*Game Theory and Applications VIII"* in 2001. In this case study Ditmarsch uses dynamic epistemic logic to describe the different game actions in Cluedo, and how the actions influence the knowledge states of the players. Ditmarsch presents four different game actions known as *nocard*, *showcard*, *endmove* and *win*. To explain this he uses a simpler game with the same game mechanics, and the same game actions as Cluedo. The game consists of three players and three cards. Each card has a different colour, red, blue or white. The players have one card each whose colour is known to them, and the objective of the game is to figure out what colour the other players cards are.

In order to do this they use the different game actions to change the knowledge states of the players. An obvious example of this would be the *nocard* game action, if player 2 announces that he does not have the red card, the whole state of the knowledge base will be changed. To show this in an easier way Ditmarsch uses a Kripke-model.[14] Assuming that player 2 tells the truth, all knowledge states where player 2 has the red card are no longer valid.

Originally with 3 different cards you can have 3! = 6 different combinations, two of them where player 2 has the red card. This means that with one game action the the number of possible knowledge states has been reduced to 4, and the knowledge of the different players has been changed drastically.

Even though there are six combinations there are only two different combinations that are plausible for each player. If the game state is rbw, (meaning player 1 has the red card, player two has the blue card and player 3 has the white card) the two states plausible for player 1 will be rbw, and rwb. Likewise for player 2 the plausible states will be rbw and wbr, and player 3 rbw and brw. Notice that the states wrb and bwr are not plausible for any of the players, however it is not common knowledge that they can be disregarded. An example is that player 3 can imagine that state bwr is plausible for player 1, so in order to keep track of what player 1's knowledge is it must be accounted for. So let's see what happens when player 2 announces that he does not have the red card:

The result of this is that it becomes common knowledge that the states wrb and brw can be disregarded. Because of this player 3 can disregard brw and now knows that rbw is the win state of the game. Seemingly nothing happens to player 2's knowledge because he already knew those states could be disregarded. But actually player 2 now knows that either player 1 or player 3 knows the answer. Before the announcement he knew that one of the players already knew that he didn't have the red card, because they were holding it themselves. But after the announcement all players knows that player 2 does not have the red card, and the one who gained knowledge has the final answer. Player 2 however does not have the final answer and must end his move. Or player 2 could try to guess the right answer having a 50/50% chance of success.

Player 1 who is in possession of the red card is now aware that player 3 knows the solution, and he knows that player 2 knows that either him or player 3 has the solution, and he knows that player 3 knows that he knows, that player 3 knows the solution. And so on, all of this is valuable information in a game that continues for longer than one round. The article continues with the other game actions, and then goes over to explain how it would work in a game of Cluedo. Which is more complex, but still similar. But the example above shows that even what seems like a simple action in a simple game, can have huge consequences to the knowledge states of the participants and also to the knowledge base as a whole.

2.3 Dynamic epistemic logic:

Dynamic epistemic logic is a form of modal logic that is used for reasoning about knowledge. It is a relatively new form of logic and was developed by C.I. Lewis in the early 1900s. But the modern form that will be used in this thesis was developed by Saul Kripke in the 1960s. The uses for dynamic epistemic logic are spread to many different fields of research and can be found in economics, artificial intelligence, linguistics and any other field where reasoning about knowledge is useful.The constant change in knowledge and the reasoning behind getting to the different states of knowledge, also known as knowledge states, is what dynamic epistemic logic is all about.

One of the ways used by Kripke to explain and illustrate the knowledge states is using Kripke models. These models illustrates how different knowledge states are connected, and what can be derived from having knowledge about them. For instance in the three cards example, for how player three can know that player two has the blue card. Player three can know that because after removing the states where player two has the red card, the only viable options for the winstate is a state where player two has a blue card. And therefore knows player two has a blue card.This might sound overly obvious, but to a computer model it is very valuable information. And Cluedo who has 21 cards, is way more complex and knowing how to get the most out of the information at hand is a must for the agent.

In Ditmarch's article he uses dynamic epistemic logic to describe the way the knowledge is spread in a game of Cluedo. Because of this it will be vital to learn the basics of this type of logic to be able to start an implementation to prolog. Since dynamic epistemic logic is a subgroup of modal logic, which expresses modality in logic form. The basic expressions of modal logic must be learned. One article that goes through these basics is "*Playing Cards with Hintikka - An Introduction to Dynamic Epistemic Logic*" by Hans Van Ditmarsch, Wiebe van der Hoek and Barteld Kooi.[14] The article can be read without any previous knowledge of the topic, and covers it to a satisfactory extent. That the article is partially written by the same author as the Cluedo article as it provides similar examples explained in more detail using the same methods.This article also uses the Kripke-model that is explained in length, but is assumed to be pre-knowledge in the Cluedo article.

Keep in mind that the main reason for learning dynamic epistemic logic is to be able to find a way to implement the concepts into prolog, and not necessarily learn in depth about this type of logic.

2.3.1 Basics expressions of modal logic:

To be able to explain this we first have to go through the basics expressions of modal logic. In modal logic two of the most basic expressions are [] and ◊, where [] means that something is known to be true, and ◊ means that there is a possibility that something is true. In the article [] has been replaced by K and ◊ has been replaced by ˆK. [1]. This thesis will use the same connotations. Another expression is C, which is an expression of common knowledge. That is knowledge a group of players share, that others do not know. A final expression to be aware of is general knowledge. This is knowledge all players knows, and is represented with E.

2.4 Definitions of dynamic epistemic logic:

The articles previously mentioned provides several definitions for the concepts of dynamic epistemic logic. In this thesis however, we will only go through the first three of them:

The cluedo article defines the language L_A used in the explanations and the knowledge actions KA_A that are available as the following definitions. These definitions are when we have a set of agents A, and a set of P atoms. The first definition shows the dynamic modal operators that we call the language L_A:

2.4.1 Definition 1(Dynamic epistemic logic - L_A):

$L_A(P)$ *is the smallest set such that, if* $p \in P, \varphi, \psi \in L_A(P), a \in A, B \subseteq A, \alpha \in$ $KA_A(P)$,

then $p, \neg\varphi, (\varphi \wedge \psi), K_a\varphi, C_B\varphi, [\alpha]\varphi \in L_A(P)$. [6]

So what does this mean? To explain it in terms of Cluedo we can say that the dynamic modal language we use is the most condensed set of where , p is a single atom from the set of atoms. In cluedo that means that p is either a single card or a combination of cards, i.e a winstate. $\varphi, \psi \in L_A(P)$ means that φ *and* ψ are expressions or announcements of P in the language L_A.

$a \in A, B \subseteq A,$ means a is an element in A, which is an agent, which is a single player. And B is a subset of A meaning a group of players. $\alpha \in KA_A(P)$, is that α is an expression of a knowledge action, which we will look at later. If all of this is true. We can induce that the following is also true, and we can use it in the language L_A. [6] The following being $p, \neg\varphi, (\varphi \wedge \psi), Ka\varphi, CB\varphi, [\alpha]\varphi \in LA(P)$. That is p which is already explained. $\neg\varphi$ is that an expression can be not true. $,(\varphi \wedge \psi)$ multiple expressions are true together. $Ka\varphi$, agent a knows expression φ to be true. $CB\varphi$ a group of agents B has common knowledge that φ is true. And finally $[\alpha]\varphi$, meaning after the announcement of α, φ is true. These are the different operators that is used to create a model of Cluedo.

Next definition is of KA_A, the knowledge actions available to the agents.

2.4.2 Definition 2 (Knowledge actions - KA_A):

Given a set of agents A and a set atoms P, the set of knowledge actions $KA_A(P)$
is the smallest set such that, if $\varphi \in L_A(P), \alpha, \alpha' \in KA_A(P)$, $B \subseteq A$, *then* :
$?\varphi, L_B\alpha, (\alpha ; \alpha'), (\alpha \cup \alpha'), (\alpha ! \alpha'), (\alpha \cap \alpha') \in KA_A(P)$. [6]

A, P, B and φ are the same as in definition 1, what is new here are α *and* α'. These are different epistemic states. Where a knowledge action moves a set of agents from state α to state α'. This knowledge can be changed in a number of ways. Using this definition for

Cluedo we can get a set of four different knowledge actions available to us. nocard, showcard, endmove and win. [6]

They are defined as the following:

$\text{nocard}_b^{a,\{c,c',c''\}}$	$L_{123456}?(\neg c_b \wedge \neg c'_b \wedge \neg c''_b)$
$\text{showcard}_{b,c}^{a,\{c,c',c''\}}$	$L_{123456}?(!L_{ab}?c_{\mathbf{b}} \cup L_{ab}?c''_{\mathbf{b}} \cup L_{ab}?c''_{\mathbf{b}})$
endmove^a	$L_{123456}?\neg K_a \delta^0$
win^a	$L_{123456}?K_a \delta^0$

In nocard a player ‘a’ asks a player ‘b’ if he has one of the cards c,c’ or c’’.If player b use nocard it means he does not have any of those cards. With this information player 1-6 all learn that the player ‘b’ does not have card c, c’, and c’’. This might seem trivial, but it is important information to the model. And can be used further into the game.

In showcard the same query is done by a player ‘a’ but the player ‘b’ does have card ‘c’. He shows the card only to the player ‘a’. With this information player 1-6 learns that player a and b learns that b has card c or c’ or c’’.

In endmove, when a player ends his round without winning. Player 1-6 learns that ‘a’ does not know the winning move. Similarly in win, player 1-6 learns that ‘a’ does know the winning move.

2.4.3 Definition 3 (Semantics of L_A):

Let $(M, w) = s \in S_A$ *and* $\varphi \in L_A$, *where* $M = \langle W, \{\sim_a\}_{a \in A}, V \rangle$.

We define $s \models \varphi$ *by induction on the structure of* φ. [6]

$$M, w \models p \quad :\Leftrightarrow \quad w \in V(p)$$

$$M, w \models \neg\varphi \quad :\Leftrightarrow \quad M, w \not\models \varphi$$

$$M, w \models \varphi \wedge \psi \quad :\Leftrightarrow \quad M, w \models \varphi \text{ and } M, w \models \psi$$

$$M, w \models K_a \varphi \quad :\Leftrightarrow \quad \forall w' : w' \sim_a w \Rightarrow M, w' \models \varphi$$

$$M, w \models C_B \varphi \quad :\Leftrightarrow \quad \forall w' : w' \sim_B w \Rightarrow M, w' \models \varphi$$

$$M, w \models [\alpha]\varphi \quad :\Leftrightarrow \quad \forall S \subseteq S_{\subseteq A} : (M, w)[[\alpha]]S \Rightarrow \exists (M', w') \in S : M', w' \models \varphi$$

As you can see these are building on the same operators that was given in definition 1. The expression $M, w \models p$ should be read as p is satisfied in state w of model M. (M,w) is an epistemic state from a set of states S_A, and φ is an expression from the language L_A. The model is defined as $M = \langle W, \{\sim_a\}_{a \in A}, V \rangle$, and is a model developed by Saul Kripke in the 1960s [6] Now let's look closer at definition 3.

First is $M, w \models p :\Leftrightarrow w \in V(p)$ which says that p is satisfied in state w of model M, if and only if state w is an element in the valuation of p. If we again say that p is $card_1$ that means that this expression holds true for every state w that contains card number 1. For instance $s(1,7,13) \models card_1$ holds true.

Next is $M, w \models \neg\varphi :\Leftrightarrow M, w \not\models \varphi$. Meaning not expression φ satisfies state w in model M, if and only if expression φ does not satisfy M,w. Let's look at this a bit closer. If φ = $Player3_{card2}$, and M,w is a state showing that the Player3 has been dealt the cards [[3],6,17,3], then this does not satisfy that player3 holds card number 2. And therefore the state saying Player 3 has the cards [[3],6,17,3] is satisfied that $Player3_{card2}$ is not correct. Meaning $Player3_{card2}$ can be eliminated from the model.

$M, w \models \varphi \wedge \psi :\Leftrightarrow M, w \models \varphi$ *and* $M, w \models \psi$. This is also a basic expression saying that both expressions φ and ψ satisfy M,w, if and only if both of them satisfy M,w by themselves as well.

$M, w \models K_a\varphi :\Leftrightarrow \forall w' : w' \sim_a w \Rightarrow M, w' \models \varphi$. This says that it is true that *a* knows φ in M,w if and only if, for every state w' where w' relates to w through *a* it is implied that φ satisfies M,w'. An example would be in state [[3],6,17,3] you can say $K_3Player3_{card6}$, meaning player 3 knows he has card 6. This is only true if every other state player 3 can consider also says that he has card 6.

$M, w \models C_B\varphi :\Leftrightarrow \forall w' : w' \sim_B w \Rightarrow M, w' \models \varphi$. This is very similar to the previous expression. The difference is $C_B\varphi$ which means that a group of agents 'B' share the common knowledge φ, if all states concludes with φ being true.

The last one is $M,w \models [\alpha]\varphi \ :\Leftrightarrow\ \forall S \subseteq S_{\subseteq A} : (M,w)[[\alpha]]S \Rightarrow \exists (M',w') \in S : M',w' \models \varphi$.
This sentence can also be written as follows. $M,w \models [\alpha]\varphi \ :\Leftrightarrow M,w \models \varphi \ \Rightarrow M|\varphi, w \models \alpha$.
This means that after every announcement of knowledge action α, φ holds if and only if φ implies that φ holds when α has been annonced.

2.6 Implementation of AI in games.

In the field of artificial intelligence, game implementations are quite common. This is because of the problem solving nature of games. Methods developed for finding solutions for agents to solve a game, can help in finding solutions for problems in different areas of artificial intelligence, or in completely different fields of research.

Typically games in AI are deterministic, two-player games that are fully observable and turn-based. Such as chess or tic-tac-toe. One of the reasons for this is that it is the most simple type of game to create an agent for. It has perfect information, meaning that both players know everything there is to know about the game at all times. Imperfect information is the opposite of perfect information and means that some players have information that is exclusive to them or set of players. It has only two players so there is only one other player to keep track of, even simpler would be to have a single agent game, but because games have multiple agents more often than not, a two player game is more natural. That it is turn-based means that each player take turns to do game actions, and can only act when it is their turn. As opposed to being played in real-time, where all agents can act freely at any given time, which is a lot more complex. If the game is deterministic the outcome of the game can theoretically be predicted with a 100% certainty. A deterministic game means that the next state of the game is completely dependent on what the act in the previous state was, and it is therefore easier to predict how a move will affect the game. If the game is deterministic the outcome of the game can theoretically be predicted with a 100% certainty. The opposite would be a non-deterministic or stochastic game. These kind of games have some element of randomness to them that can not be predicted. This makes it harder for an agent to predict how a game action will affect the game, and what the outcome of the game will be.

2.6.1 The environment of Cluedo.

Cluedo is of a game with imperfect information, it is turn-based and has up to six players, and is arguably a stochastic game. It is definitely a way more complex game that tic-tac-toe, and aside for the turn-based aspect it is the opposite in terms of its qualities. It is obvious from the start that this is a game with imperfect information. The whole game is based around figuring out a card combination that is not known to you, and the only cards that are known to you are the cards you were dealt at the start of the game. Though you do gradually gain more knowledge of the cards as the game progresses.

Equally obvious is the turn-based aspect of the game. Each player has their turn to roll the dice, move around on the map the amount of steps they got from the dice, and if they landed in a room they can voice a suggestion. When one player's turn is over the next player turn begin, until everyone has had their turn in the round. When that round is over a new round is started with the first player again and each player takes their turn in the same order as the previous round. This continues for as many rounds it takes for the game to come to a conclusion. But this does not mean that the only time in the game that is relevant for the player is when it’s their turn. The players gain more knowledge continuously from reasoning based on the information they get when it’s another player’s turn.

Since it is a multi-agent game with more than two agents the amounts information the agents must keep track of in order to play optimally increases drastically. In Cluedo you can play with two to six players. For each added player the challenge of finding the win state increases. There is always the same amount of cards in play. But in a Cluedo game with two players the players have nine cards each, instead of the three cards each of the players will have in a game with six players. To reach the win state the agent must find the three cards that none of the players have. If player 1 in the game’s first move makes a suggestion and none of the other players have the cards he asks for, he has suggested the win state and can make an accusation. In a game with only two player there is a higher chance of suggesting the win state in the first move, but it also comes with a higher risk. If the agent always only asks for cards it doesn't hold itself it could be disadvantageous. For instance if the other player holds one of the cards being suggested, but not the two others, the asking agent will only get to know the one card that is not in the win state, while the other agent will get to know two cards in the win state. Making an implementation where the agent makes an bluff suggestion (that is asking for a card it already holds itself), would seem to have more importance from the start in a game with fewer players. This is not to say that bluffing does not have a vital part in six-player games as well.

When judging if a game is deterministic or stochastic the best way is to look at the game from the players perspective. In a game like Cluedo that is only partially observable there are many things that seem to be random, like the dealing of the cards, and the rolling of the dice. At the start of the game there is no way of knowing where any of the cards have been dealt, other than the ones in your hand. The part where players can bluff also seem like a random element in the game, and unless the agent has the information to see through the bluff it is hard to assume anything with confidence. For these reasons I would judge Cluedo to be a stochastic game rather than a deterministic one. Which also mean that finding an optimal solution for it would be challenging.

In order to make the implementation go more smoothly some of the elements of the game will be removed from the programming bit. The focus of the work is the modal element, and therefore I have decided to drop the player board. By doing this only two aspects of this is removed. Those two aspects are the dice and the ability to move on the board. Since you cannot move on the board the restriction of only being able to query a state containing the room you are currently in is removed. An agent can then ask for any state containing a weapon, person and location. By removing this aspect, the game is reduced to a query game where every player takes it’s turn to ask for a state. While it does remove much of the gameplay, it does not remove any of the relevant dynamics to the knowledge game. These aspects can also be implemented at a later point if preferable.

3 Development

Cluedo is a game where you need to keep you focus, and pay attention to the other players decisions in order to make you own. The most observant player who can ask the right questions at the right time, will be the one who comes out victoriously. From the start this sounds like a job a computer can do much better than a human. But whether or not prolog is the right language to program this kind of agent in is the question this thesis intends to find out.

Dynamic epistemic logic(DEL) does not translate directly into the language used in prolog. Therefore the program is made with dynamic epistemic logic as a basis, and borrows its concepts, it is not meant to be a copy of the logic, or to be a complete representation of the logic in the DEL articles. The key concepts of DEL, will be used in the program in order to make it as similar as possible to the logic used in the cluedo article.

The development part is divided into two categories, the implementation of the game mechanics, and the implementation of the modal logic. In the first part we start by looking at how the game world is represented, and what the world looks like tot the agent. This is done in cluedo.pl. We then move onwards to the game mechanics of dealing the cards in dealer.pl. Where we will create the winstate by creating a state that contains one card from each of the card categories, the rest of the cards will be divided equally amongst the players. After dealer.pl comes subtracter.pl. This is a predicate that will help out the modality part of the program later, it will help remove potential winstates, that are no longer viable after gaining a set of information. Next is query.pl, this is a predicate where the query of the other players take place. A potential winstate will be queried, and an answer will be provided by one of the players.

The previously mentioned files will all come toghether in unity.pl where they will be used to create a game of cluedo where the cards are dealt, a winstate is selectedl.Then the game starts with our agent doing a query and learn a card, next the the other players will do a query. After this a new round will start with the agent doing the query. This goes on until the agent finds the correct winstate. Before the implementation of the modal logic, this basic agent can search for the correct winstate in two ways, either by randomly asking for 1 of the 324 potential winstates. Or by using subtracter.pl to eliminate potential winstates when new cards are learned from a query, when randomly selcting a winstate from the new list of potential winstates. Unity.pl focuses on the latter type of agent.

In the second part of the implementation be be created as an enhancement of the unity.pl agent. We will create predicates for the knowledge actions nocard and showcard, so that we can use the information of all queries to gain more knowledge, not just the one query the agent got knowledge from earlier. The showcard and nocard predicates will give us knowledge of the potential deals and knowledge of the other players. We will then create a predicate that can analyze this information to make potential knowledge into secure knowledge. This will then be implemented into unity.pl to create a better agent. The new agent will then be compared to the old one.

3.1 Development tools:

3.1.1 Prolog:

Prolog is a declarative programming language. Meaning that you program to tell the computer what it should do, as opposed to functional programming that program the computer to know how it should do a task, every step of the way. Declarative programming can be a bit unintuitive and harder to grasp than functional programming, but often requires a considerably less amount of code to operate.

Prolog was developed from research made at the University of Aix-Marseille in the late 60s and early 70s. The language itself was created between 1971 and 1973 in a collaboration between people at the university of Aix-Marseille and the University of Edinburgh. It was one of the first programming languages that used declarative programing. The choice to use Prolog as the programming language, was partly because of all the logic associated with the project and partly because of my previous knowledge and competence in using the language. For the development of the program SWI-Prolog v.7.2.X for Windows OS has been used, the editor used is SWI-Prolog-Editor v.4.23

3.1.2 Games in prolog:

Prolog is a logic programming language that uses first-order logic. It is mostly associated with artificial intelligence programming, but can also be used for general purpose programming. Since Prolog is a declarative programming language, and is mainly represented by facts and rules. It should be possible to implement a Cluedo game in prolog, but it is yet to be seen how suitable the programming language will be for this kind of task.

A version of Cluedo has already been implemented in prolog.[9] But this version does not play like a traditional game of Cluedo, and there are no reasoning agents in the game. There is only one player, who gets clues from the game and from them must reason what person, weapon and location is correct. This is done by a human player outside of the program. The goal of the master thesis is to implement a rational reasoning agent that can do the same in the program, playing against other agents.

Most of the examples of games that have been implemented in prolog are two player games where the user play vs the computer. Tic-tac-toe[10] and chess[11] are examples of this. But there are some examples of multi-agent games that have been implemented in prolog. The card game Hearts is one such example.[12]

3.2 Implementation of the game mechanics:

3.2.1 Mental representation of the agent

The way the agent looks at the world is a list of the 324 winstates. The goal of the agent is to reduce that list to 1. The way this is done is by using concepts from dynamic epistemic logic, in order to exclude possible states in the list. The first of these exclusions are done right after the cards have been dealt. From the relation rules of DEL, the agent knows that any state containing a card that is on his hand, cannot be the final winstate. And will therefore remove all those states from the list. If there are 6 players each of them get 3 cards each from the dealer. Depending on the overlap of states this typically leaves the agent with 180-200 possible winstates after the dealer has dealt the cards. The states can be removed further when doing a query and learning a card another player holds. After learning a card the agent will always remove all states containing that card from the list of possible states. But this list is only the representation of the agent's own explicit knowledge. It must also keep track of other players knowledge.

The agent's knowledge of the other player's knowledge is divided into two. The first is a list containing representations of all the players in the game, and the possible cards they might have on their hand. The other is a list of cards a player might know are not in the winstate, but are not on their hand explicitly. All of this is seen from the agent's, perspective. There are also possibilities of implementing representations of what other players know of other players, but the information gain is small, and memory usage is big. More of this later. The list of cards a player does not have on their hand is represented by a list containing the cards 1-21. Cards the agent have been dealt can be removed from this list, giving 18 cards in a 6 player game. More information is gained using the knowledge action *nocard*. Where you can learn 1-3 cards a player does not have. Another way to gain knowledge is the knowledge action *showcard*. With showcard you learn that a player learns one of three cards, you also learn the the player showing the card has one of those three cards on his hand. You don't know which but by looking at all the information from all the players you can get a pretty good idea. For instance if player 4 answers to player 2 that he has one of the cards s(1,7,13), and you know from an earlier query that player 4 answered nocard to s(1,7,14), then you have now learned that player 4 holds card 13, and that player 2 knows it. As a sidenote you have also learned that player 3 does not hold either of cards 1,7 or 13, because he must have answered nocard in order for player 4 to answer.

Now that the agent knows card 13 is not one of the cards in the envelope, it can remove all states containing card 13 from the list of winstates. The program analyzes with this information at the end of each round, not continuously. But since the agent has no input in the game more than once per round, this is of no consequence to the performance of the agent.

3.2.2 Kripke:

The Kripke model contains three components. These are W, ~ and V. W is the domain or the world of the model. This is where all the parts that come into play are. In the case of Cluedo this is all the combination of the cards from different categories, which are called winstates in this thesis, but also single cards which is used in queries and all combinations the card can be dealt to the players. ~ sometimes also called R is the relations between the elements in W. In Cluedo a state is related to all other states that contain one or more of the same cards as the given state. V is valuations regarding an expression. That means that it checks for truth or falsehood of a expression. For instance V($card_1$) will give us every state that contains card number 1.

I did consider creating an explicit kripke model for the program. And if we only dealt with the winstates that would have been fine. As mentioned earlier there are 324 unique winstates in Cluedo. Though this is a relatively large number, all states can be explicitly written in the program as facts. But it is when you deal the cards to the players the problems arise. After having dealt 3 of the 21 cards to the winstate there are 18 cards left to deal to the players. The different categories of cards does not play a role when dealing the cards to the players. If we assume there are 6 players, the 18 cards can be dealt in a huge number of ways :

$$18! / 3!3!3!3!3!3! = 137\ 225\ 088\ 000$$

When all of these different card deals are combined with a unique winstate there are even more combinations of card deals:

$$137\ 225\ 088\ 000 * 324 = 44\ 460\ 928\ 512\ 000$$

That is almost 44,5 trillion different states. And this is not considering that there are really 21 different cards each player can receive not 18. But the reasoning for choosing 18, instead of 21 which gives considerably more states, is that the winstates is not like the others since it cannot be dealt completely random. But must have 1 card from each category. And therefore I remove 3 cards when calculating the number for the deal to the players, because no matter which cards are available there will always be exactly 18 cards. The agent can know that there are 137,2 billion different card deals, but he does not know which three cards are missing from the deal. What the agent can know is that it is one card from each category which means the deal to the players consist of 5 person cards, 5 weapon cards, and 8 location cards. So since the three missing cards can be in 324 different combinations, that means that the 18 cards also can be in 324 different combinations. Therefore the 137,2 billion is multiplied by 324 and we get the total of 44,46 trillion unique states. This number is far too high to have explicit in the program, or in the model. And another way must be found to represent this knowledge.

3.2.3 cluedo.pl

The very start of the programming process was to put in all the facts of the game explicitly into the program. There are 21 unique cards in Cluedo, divided into 3 different categories. In prolog the cards are represented by a number. From 1-21. There is no difference made between the categories at this point. However the three cards on the table which makes the winstate are 3 cards, 1 from each category. Therefore It is better to explicitly write down all the winstates and mark the cards into categories that way. A winstate contains three different cards and looks like this: “s(1,7,13)”, the first number “1” represent a person card and can be any card from 1-6. The second card “7” is a weapon card, represented by a number from 7-12, the third and final card “13” is a location card, and can vary from 13-21. As mentioned earlier there are 324 unique combinations between these cards. That is, there are 324 different winstates. All of them are explicitly represented in the program as a state like “s(1,7,13)”. Representing the states and categories this way, makes it so that the program can differentiate between times the categories matter, which is in the winstates, and when they don’t matter, which is when dealing the cards, and what kind of card a player holds.
In cluedo.pl all the explicit information needed in the game is stored. Even though some of it is not used it is still there to show what a card number is representing.

The predicates in cluedo.pl are the following.

person/1
weapon/1

location/1

cards/1

cards2/1

winstates/1

The three predicates person/1, weapon/1 and location/1 are used to show what number represents which card. It can also be used in combination with cards/1 and cards2/1 to explicitly state what type of card a number is. This however has not been needed so far in the implementation.

cards/1 is a list containing the string name of the different cards, while cards2/1 is a list containing the number representation of the cards, that is a list containing the numbers 1-21.

winstates/1 are all the 324 different winstates written explicitly into the program.

3.2.4 dealer.pl

The next thing that needed to be implemented was the card dealer. A implementation that is able to deal the cards to 2-6 players and to deal one card of each category to the "table" in order to create a winstate. This is done in *dealer.pl*, by the cooperation of several predicates. The predicates of dealer.pl are:

dealer/3

dealer2/3

winselector/2

test2/3

list/3

deal/4

rev2/3

dealer/3 is the predicate that starts the process of dealing the cards. The full predicate is:

```
dealer(L,X,Win):- winselector(ND,Win),dealer2(ND,X,L).
```

Here L and Win are the variables that we are interested in getting. L is the list of cards that are dealt to the players and Win is the winstate that contains one card from each category. X is the number of players, which must be given at the query.

An example of a query and answer is as follows:

```
Q:
dealer(L,6,W).

A:
L = [[[1], 3, 19, 4], [[2], 15, 11, 7], [[3], 1, 21, 14], [[4], 9, 20, 12], [[5], 10, 18, 16],
    [[6], 5, 6, 13]],
W = s(2, 8, 17);
false.
```

As we can see the winstate W is s(2,8,17). It contains one card from each category. In this specific case it is ('Professor Plum','Dagger','Billiard Room'). The rest of the cards has been divided equally between the other players. For instance player [1] which is considered our player, has been dealt the cards 3,19 and 4. Now let’s look at the technical details of the program:

The first thing dealer/3 does is to call to winselector/2:

```
winselector(ND,Win):- cards2(OD), test2(X,Y,Z), subtract(OD,[X,Y,Z],ND),s(X,Y,Z)=Win.
```

winselector/2 starts out by creating a variable containing all cards and calls it OD. cards2/1 is the predicate from cluedo.pl. Next it calls on test2/3 which selects a random member from all the unique winstates, it then use subtract/3 to remove the cards used for the winstate and return the 18 remaining cards as ND, and then returns the winstate as Win.

Example:

```
Q:
winselector(ND,Win).
```

```
A:
ND = [1, 2, 3, 4, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20, 21],
Win = s(5, 7, 18);
false.
```

It then comes back to dealer/3 where it calls on dealer2/3 with the cards (ND) and the number of players (X) in order to get a list of the cards dealt to X players. The full dealer2/3 looks like this:

```
dealer2(C,X,L):- list(X,[],L1), deal(C,L1,[],L2), rev2(L2,[],L).
```

It starts out by calling to list/3. list/3 is a predicate that creates a list with X amount of lists. One for each player. Each list gets a new list with a number starting with 1 then being the previous +1 from there on. This way each list gets an identifier that can be used for the representation of the players. The list of list is known as L1 is then used in deal/4.

Example of list/3:

```
Q:
list(6,[],L1).
A:
L1 = [[[1]], [[2]], [[3]], [[4]], [[5]], [[6]]] ;
false.
```

deal/4 is used in four different instances and is the predicate that does the dealing of the cards:

```
deal([],[],L,L2):-  L2=L,!.
deal([],L1,L,L2):-rev(L1,R), append(R,L,L2),!.
deal(C,[],L,L2):- rev(L,R),deal(C,R,[],L2).
deal(C,[L0|L1],F,L4):- random_select(Y,C,R),add(Y,L0,L2), add(L2,F,F1),deal(R,L1,F1,L4).
```

Example query of list/4:

```
Q:
```

```
deal([1, 2, 3, 4, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20, 21],[[[1]], [[2]], [[3]], [[4]],
  [[5]], [[6]]],[],L2).
A:
[[20, 21, 3, [6]], [15, 10, 2, [5]], [9, 4, 12, [4]], [14, 19, 1, [3]], [13, 6, 17, [2]],
[16, 11, 8, [1]]]
```

The process starts out in the last of the four options:

```
deal(C,[L0|L1],F,L4):- random_select(Y,C,R),add(Y,L0,L2), add(L2,F,F1),deal(R,L1,F1,L4).
```

Here a random card is chosen from C, we call that card Y. The rest of the cards is now called R. Y is then added to L0, which is the first of the lists created in list/3. In other words this first list is a representative of the first player, and has the [1] identifier in it.The new list containing Y and L0 is called L2. L2 is then added to F, which was just an empty list, and creates F1. It then starts a new iteration of deal/4 where C is replaced by R and F is replaced by F1, and L1 replaces [L0|L1]. This process goes on until all of the players have gotten one card each. When there are no more players to give a card to the next rule of deal/4 comes into play:

```
deal(C,[],L,L2):- rev(L,R),deal(C,R,[],L2).
```

This rule simply reverses the list of the players and starts deal/4 over again. The reason the lis needs to be reversed is. Because player 6 was the last player to receive a card, but is also fist in the list. We need to put player 1 at the top of the list again. The reversed list is called R. R is then used for a new iteration of dealing cards, so that every player gets their second card. This continues until all the cards have been divided equally to all players. When all cards have been dealt one of two rules will be used.

```
deal([],[],L,L2):-  L2=L,!.
```

Or

```
deal([],L1,L,L2):-rev(L1,R), append(R,L,L2),!.
```

The difference between the two is whether the cards have been dealt equally or not. When there are 2,3 or 6 players the cards will be dealt equally and deal([],[],L,L2):- L2=L,!. Will be used. Both the list of cards and the list of players are empty, so now all that is needed is to give the goal list (L2) the same value as the list with all the cards dealt (L). and then cut the process by using !. In the case of 4 or 5 players the cards can not be dealt equally. As you can see if you try to divide 18 to 4 or 5 it does not become natural numbers. (4,5 and 3,6 respectively). When this happens not all the players will have made it to L, but some are still in L1. We must then reverse the players in L1 for the same reason as earlier, and append it with L in order to create the full list L2. When we have L2 we can cut the process using !.

When deal/4 is finished dealer2/3 continues to rev2/3:

```
rev2([],L1,L) :- L = L1.
rev2([H|T],L1,L) :- rev(H,H1), add(H1,L1,L2), rev2(T,L2,L).

Example:
Q:
rev2([[20, 21, 3, [6]], [15, 10, 2, [5]], [9, 4, 12, [4]], [14, 19, 1, [3]], [13, 6, 17, [2]],
    [16, 11, 8, [1]]], [],L).
A:
L = [[[1], 8, 11, 16], [[2], 17, 6, 13], [[3], 1, 19, 14], [[4], 12, 4, 9], [[5], 2, 10, 15],
   [[6], 3, 21, 20]] ;
false.
```

rev2/3 is a predicate that takes the list created in list/4 and reverses the first list in it, and then puts them into a new list. It does this to every list until there are no more lists to reverse. Then it returns the reversed lists. The reason for doing this is that before reversing, the identifier is at the back of the list. In order for it to be an effective identifier it is important that it is the first element in the list, both for identifying purposes and in order to easily remove the identifier from the list. Also the order of the cards does not have any significance at this point. After rev2/3 has finished, dealer2/3 is also finished and will give the deal to dealer/3. dealer/3 will then return the values.

3.2.5 subtracter.pl

Now that the cards have been dealt we need to use the new information we have learned in the game. After learning the three cards the agent was dealt it can remove all winstates containing these 3 cards. This is done by *subtracter.pl*. In *subtracter.pl* there are several predicates that are of importance:

```
run/2
run2/3
run1/3
subs2/4
subs/4
```

This process startes with run/2:

```
run(L,N):- winstates(WS),run2(L,N,WS).
```

run/2 starts by creating a variable containing all 324 winstates and names it WS. It then calls on run2/3 to continue the process. In run2/3 there are 3 parameters. The first is L, which is the goal list containing the updated amount of winstates after the process is finished, and N is a list containing a player, like the ones created in dealer/3, in our example the list of player 1 looked like this: [[1], 3, 19, 4]. The last parameter is WS, which was made just prior. Let's look at run2/3 next:

```
run2(L1,[H|T],WS):-run1(L1,T,WS).
```

run2/3 is a very simple predicate, and it's purpose is to remove the identifier from the list of cards, as it will not be relevant in the next process. run2/3 then calls for run1/3:

```
run1(L,[],WS):- L=WS.
run1(L1,[H|T],WS):-subs(H,WS,WS,L), run1(L1,T,L).
```

This is where most of the work happens, in order to show how the predicate works in an example I will use a smaller set of winstates and a single card used for removing winstates:

```
Q:
run1(L,[16],[s(1,9,13),s(1,9,14),s(1,9,15),s(1,9,16),s(1,9,17),s(1,9,18),s(1,9,19),s(1,9,20),
      s(1,9,21)]).
A:
L = [s(1, 9, 13), s(1, 9, 14), s(1, 9, 15), s(1, 9, 17), s(1, 9, 18), s(1, 9, 19), s(1, 9, 20),
    s(1, 9, 21)] ;
False.
```

As you can see the state containing card 16 has been removed. If I was to replace 16 with 1. For this example I would get L = [].

run1/3 is an iterative predicate that uses subs/4 in order to remove all winstates that are no longer viable. That is, all the winstates containing cards we already know is not in the winstates. It calls to subs/4 (subs(H,WS,WS,L)) and sends H, which is the first card on player 1's hand and the winstates WS two times, L is a list with the updated winstates and will be used in the next iteration.

```
subs(N,L,L1,L3) :-L=[H|T],(H=s(N,_,_);H=s(_,N,_);H=s(_,_,N)) ->
                        subtract(L1,[H],L2),subs(N,T,L2,L3);subs2(N,L,L1,L3).
```

What subs/4 does is that it goes through every winstate and checks if the card at hand is a part of the current winstate. The way it does this is that it takes out the first winstate from the first list of winstates L, It calls this winstate H the rest of the winstates are called T. It then checks if card N is in any of the three positions of the winstate. That is s(pos1,pos2,pos3). If state H does contain N, H will be removed from L1 which is the second list of the winstates.The updated list L2, will then be sendt to do subs/4 again together with T. However if H does not contain N, subs2/4 will be called:

```
subs2(_,[],L1,L1).
subs2(N,[H|T],L1,L3):- subs(N,T,L1,L3).
```

subs2/4 contains the win criteria of the predicate, which is when there are no more states left. But as long as the win criteria is not fulfilled it will keep the iterations going. It removes H from the states that are being asked about, but does not remove it from L1, and then restarts the subs/4 predicate. Since N was not in state H, H is not removed from L1 and continue with the rest of the still possible winstates. The program will do this until all states have been checked. It will then go back to run1/3 who will send the next card to be inspected. This continues until all cards have been checked up against all winstates. And every winstate what contained one or more of the cards has been removed from the possible winstates. The new list of winstates is then returned to run2/3. We will later see how all this connects together in unity.pl.

3.2.6 query.pl

Next after we have removed the unnecessary states we want to be able to do a query to the other players. This is done in *query.pl*. Predicates in this file is:

```
choice/4
choice1/4
win/2
```

choice/4 is the predicate that starts the query:

The task of this predicate is to do a query to the other players and return a list with the player who answered and the card he answered with.

Example:

```
Q:
choice( [[[1], 3, 19, 4], [[2], 15, 11, 7], [[3], 1, 21, 14], [[4], 9, 20, 12], [[5], 10, 18, 16],
        [[6], 5, 6, 13]],s(5,6,20),A,s(2, 8, 17)).
A:
A = [[4], 20]
```

The state being queried is s(5,6,20). Player [1] is the player doing the query of the deal we dealt earlier, and player [4] is the first player to use showcard, showing card 20. Now let’s look at the details:

```
choice([H|T],s(X,Y,Z),[A2,A1],W):-      H=[H1|T1],A2=H1,
                                        choice1(T1,s(X,Y,Z),[A2,A1],W),!;
                                        once(choice(T,s(X,Y,Z),[A2,A1],W)).

choice([],s(X,Y,Z),[A2,A1],Win):- s(X,Y,Z)==Win,A2=[0],A1=(X,Y,Z);!.
```

choice/4 has 4 parameters. The first one are the cards that have been dealt. The cards of the player doing the query are not included in this list. That way the player can ask for a card even if he already owns it, without the query failing. Already from the start the dealt cards are divided into H and T, H being the player next in line after the player doing the query, and T being a list of the remaining players. In the second parameter we find the state being queried. The three cards in the state are being represented by X,Y and Z. Next parameter is for the answer. The answer is divided into A2 and A1. A2 is the identifier of the player giving the answer and A1 is the card that has been given. This way we can keep track of who has given knowledge of which card. The fourth and final parameter is the winstate, named W. It has no relevance in this part of the predicate, but need to be known for later use.

choice/4 starts out by dividing H into its identifier H1 and the cards T1, it then marks A2 as the same as H1. Next it calls to choice1/4 which we will look into in a bit. If choice1/4 is true it will stop the process with !. If not it will run choice/4 again now replacing H with T.

If choice/4 runs through all players and doesn't get an answer from any of them, that means one of three things. Either we have asked for the winstate and now know the final answer, or we have asked about the cards we already own, it can also be a combination of the two. Of course a player should never ask only for the cards he already owns, but it is still a possibility. Because of this possibility the state asked for is checked against the winstate. If it is the same A2 gets the identifier [0], which is only given to an answer containing the winstate. A1, is marked as all three cards X,Y and Z. Now let's look at choice1/4

```
choice1([H|T],s(X,Y,Z),[A2,A1],W):-   H==X -> A1=X;H==Y -> A1=Y;H==Z ->
                                      A1=Z;choice1(T,s(X,Y,Z),[A2,A1],W).
```

choice1/4 takes the first card from the hand of the player and calls it H, the rest of the cards are called T, it then compares H with all of the cards in question (X,Y and Z). If any of them is the same card as H, A1 will be given the same value as that card. If not choice1/4 will be called upon again now using T and removing H from the equation. This continues until a card has been identified or none of the players cards are the same as the ones in question.

Example of player 4 being queried:

```
Q:
choice1([ 9, 20, 12],s(5,6,20),[[4],A1],s(2, 8, 17)).
A:
A1 = 20 ;
false.
```

Please note that as of yet there is no predicate that will choose the order of the cards on each of the players' hands. This is a problem because the player would normally want to decide which card it will show if there are several options. The way it is now the order of the cards never changes, and the first card will always be shown if there are options. However the program is designed with the possibility of creating a predicate can change the order of the cards, given by a different strategies. But the implementation of this has not taken place in this version of the program..

The last predicate in this file is win/2:

```
win(W,(X,Y,Z)):- W==s(X,Y,Z),true;fail.
```

win/2 is simply a check so see if a state is indeed the winstate. If it is it will return true, and the program will finish, if not it will fail and the program continue.

```
Example:
Q:
win(s(2, 8, 17),(2,8,17)).
A:
true.
```

3.2.7 unity.pl

Next we will look at unity.pl this file puts together all the predicates from the other files and put them into action. The predicates of note in this file are the following:

```
players/1
trekkfra/4
trekkfra2/6
sporring/4
sporring2/5
sporring2/6
selecter/2
```

players/1 starts the game:

```
players(X):-	dealer(L,X,Win),L = [P|T],winstates(WS),trekkfra(P,WS,T,Win).
```

X is the number of players that will participate in the game. With that information we can call on dealer/3 to deal the cards L, and get a winstate Win. We then separate the first player P, from the rest of the players T. And make a variable containing all possible states WS. The the program then calls on trekkfra/4.

In most of unity.pl the only value that is returned is *true*. However in the program the many values are written in the console throughout the process, and the result of a query of players/1 can be seen in Appendix A. (The write/1 predicates have been removed from the presentation, In order to see the placements look at the original coding in Appendix G Now let’s look at the deal and the winstate in that query.

In this case the deal is:
Deal:[[[1],15,8,9],[[2],19,10,18],[[3],17,13,6],[[4],11,14,1],[[5],21,7,16],[[6],3,2,5]]

And winstate is: ‘Win:s(4,12,20)’

The program then writes the cards of the agent ‘my cards[[1],15,8,9]’, it then removes all states containing those cards from the possible winstates, and write how many states are left after the first removal. ‘states left:192’. The agent then goes on to do the query. ‘P: [[1],15,8,9]’ shows us that it is player [1] doing the query. ‘QM:s(4,7,16)’ means that the state being queried is s(4,7,16), and A:[[5],7] means that the answer was card 7 from player [5]. The program then goes on to remove the states containing card 7, and writes how many states are left. ‘states left:144’. Then the rest of the players do their queries. ‘Q:[[s(4,7,16),[1],[5]],[s(5,11,16),[2],[4]],[s(6,12,21),[3],[5]],[s(6,8,14),[4],[1]],[s(1,11,17),[5],[3]], [s(4,12,19),[6],[2]]]’ This is the common knowledge from the query. [[s(4,7,16),[1],[5]] meaning state s(4 ,12, 19) was queried by [1] and [5] answered. This cycle continues for however many rounds it takes for the agent to do a query of the winstate ‘QM:s(4,12,20)’. When it does the win/2 predicate will be called and if true, the program stops.

The agent is a very simple one that only eliminates states based on the information it gets from its own query. And principles from dynamic epistemic logic has not been used. In addition the competition is non-existent since the other players have been designed in a way so that they cannot win. The program is not fully developed and has only been designed to test its functionality.

But we will later look at how the information we get from unity.pl can be used to implement modality to the program. But for now, let’s continue looking at trekkfra/4 and unity.pl.

```
trekkfra(P,WS,T,W):- run2(L,P,WS),sporring(L,T,P,W).
```

Example of the parameters filled out:

```
P= [[1],15,8,9]
WS = [s(1, 7, 13), s(1, 7, 14), s(1, 7, 15), s(1, 7, 16), s(1, 7, 17), s(1, 7, 18), s(1, 7, 19),
        s(1, 7, 20), s(..., ..., ...)|...] ,
T = [[2],19,10,18],[[3],17,13,6],[[4],11,14,1],[[5],21,7,16],[[6],3,2,5]]
W = s(4,12,20)
```

There is not room to write out every winstate, but to begin with WS contains all 324 of them.

Here run2/3 from *subtracter.pl* is called, and will remove all states containing the cards player 1 were dealt. The list of updated states will be called L. It will then be sent to sporring/4 which uses L, T, P and Win:

```
sporring(L,T,P,W):-random_select(Q,L,R),choice(T,Q,[A1,A2],W),
                        trekkfra2([A1,A2],L,T,P,W,[[Q,[1],A1]]).
```

No algorithm for choosing a card has been created yet, so sporring/4 starts out by randomly selecting a state from L that will be the used as the quered state in choice/4.After querying the rest of the players T about Q it will get an answer [A1,A2]. And will be added as a parameter in trekkfra2/6, we also want the common knowledge from this query registered this is done in [[Q,[1],A1]].

```
trekkfra2(A,WS,T,P,W,Q):- A=[A1,A2],A1==[0],win(W,A2),
                               run2(L,A,WS),sporring2(P,T,L,W,Q).
```

Parameters of trekkfra2/6:

```
 A =[[5],7]
WS =  L = [s(1, 7, 13), s(1, 7, 14), s(1, 7, 16), s(1, 7, 17), s(1, 7, 18), s(1, 7, 19), s(1, 7, 20),
          s(1, 7, 21), s(..., ..., ...)|...]
T = [[[2],11,15,16],[[3],19,7,18],[[4],12,8,6],[[5],3,14,13],[[6],20,21,4]]
P = [[1],1,10,2],
W = s(5,9,17)
Q = [[s(4,7,16),[1],[5]]]
```

Take note that L has been renamed to WS in this predicate. Here all states containing cards 15, 8 and 9 have been removed from WS. As you can see it state ‘s(1, 7, 15)’, is no longer in the list.

trekkfra2/6 will start by separating the identifier and the card given as an answer. It then checks if the identifier is [0], which is the identifier for a win, it then runs win/2 and the game should finish. If the identifier is not [0], it will call on run2/3 and remove all states containing A2. The new list of states is called L. It then calls on sporring2/5, in a similar fashion as we did with sporring/4.

```
sporring2(P,T,L,W,Q1):-          add(P,[],L1), T=[T1|T2], append(T2,L1,T3), selecter(Q,T1),
                                  choice(T3,Q,[A1,A2],W),T1=[S|C],Q2=[[Q,S,A1]],
                                  append(Q2,Q1,Q3), sporring2(T1,T2,L1,Q3,L,W).
```

The first thing that is done in sporring2/5 is that we change the player that will be the one doing the query. It does this by adding P to an empty list, then separating the next player in T(T1) from the rest of the players (T2). It then puts T2 in the same list as P, so that P gets in the last place in that list. This is done using the append predicate. It then calls on selecter/2:

```
selecter(Q,T1):- winstates(WS), random_select(Q,WS,L).
```

selecter/2 is the predicate that will be used to select a state to query for the other players. It is not fully developed and is now just a predicate that selects a random state from all states.

After selecting a state in selecter/2, sporring/4 continue by getting an answer in choice/4. The common knowledge of the query is saved in Q2 and is appended with the common knowledge from the first query. This list is called Q3. The program then calls on sporring2/6:

```
sporring2(T1,[],L1,Q,L,W):-      append(L1,[T1],L2),L2=[P|T],rev(Q,R),sporring(L,T,P,W).

sporring2(T1,T,L1,Q,L,W):-       append(L1,[T1],L2), T=[T2|T3], append(T3,L2,T4),
                                  selecter(Q1,T2), choice(T4,Q1,[A1,A2],W), A =[Q1,A1],
                                  add(A,Q,Q2),sporring2(T2,T3,L2,Q2,L,W).
```

At this point the parameters look like this:

```
T1 = [[2],19,10,18]
T = [[3],17,13,6],[[4],11,14,1],[[5],21,7,16],[[6],3,2,5]]
L1 = [[1],15,8,9]
Q = [[s(5,11,16),[2],[4]],[s(4,7,16),[1],[5]]]
L = [s(1, 10, 13), s(1, 10, 14), s(1, 10, 16), s(1, 10, 17), s(1, 10, 18), s(1, 10, 19), s(1, 10, 20),
    s(1, 10, 21), s(..., ..., ...)|...] W =  s(5,9,17)
```

Variables of interest are:

```
A = [s(6,12,21),[3],[5]]
```

Q2 = [[s(6,12,21),[3],[5]],[s(5,11,16),[2],[4]],[s(4,7,16),[1],[5]]]

L2=[[[1],15,8,9],[[2],19,10,18]]
T4 = [[[4],11,14,1],[[5],21,7,16],[[6],3,2,5],[[1],15,8,9],[[2],19,10,18]]

Now that all states containing cards 7, 8, 9 and 15 have been removed, the effect of the removal of winstates is much clearer. Before the first state in the list was s(1, 7, 13), now it is s(1, 10, 13).

sporring2/6 is where all other players than our agent is doing the queries. It puts the list of players in the right order and separates the next player T2, from the rest. It then call to selecter/2 to get a state to ask for, and then to choice/4 to do the query. The state being queried, the player doing the query and the player showing the card are all common knowledge, and is saved in A so that the agent can use the knowledge later. It adds A to Q which are all the similar information from previous rounds. And finally it calls on sporring2/6 again do do the next player. When all the players have done their query it puts the list in the right order, separates the agent from the rest of the players and call to sporring/4 so that the agent can do the query again. This process continues until the winning state has been quired by the agent.

After all the players have done their queries for a round the reversed list of queries looks like this:

R=[[s(4,7,16),[1],[5]],[s(5,11,16),[2],[4]],[s(6,12,21),[3],[5]],[s(6,8,14),[4],[1]],[s(1,11,17),
[5],[3]],[s(4,12,19),[6],[2]]]

In the modality part of the program this information will be used to gain more knowledge if the other players.

3.3 Implementation of dynamic epistemic logic:

Now we will look at modal.pl this is the part of the program that tries to deal with modality and come up with strategies for what card to pick next and how states relates to other states, but also to find out which cards other players has. modal.pl contains the following predicates:

```
showcard/3
showcard2/3
nocard/3
nocard2/5
leggetil/3
leggetil2/4
first_last/3
first_last2/3
semi_perm/5
analyzer/5
```

3.3.1 First out is showcard/3:

```
showcard([],L,L):-!.
showcard(Q,M,L) :- Q= [[s(X,Y,Z),P1,P]|R],  L=[L1|L2],L1=[N|B], [P]==L1 ->
                    leggetil(L1,L3,Q),add(L3,L2,L4),showcard(R,M,L4);
                    Q= [[s(X,Y,Z,P1,P]|R],  L=[L1|L2],L1=[N|B], P==N ->
                   leggetil2(N,B,L3,[[X],[Y],[Z]]),add(L3,L2,L4),showcard(R,M,L4);
                   L=[L1|L2], first_last([L1],L2,NL),showcard(Q,M,NL).
```

Example query:

```
 Q:
showcard([[s(4,7,16),[1],[5]],[s(5,11,16),[2],[4]],[s(6,12,21),[3],[5]],[s(6,8,14),[4],[1]],
        [s(1,11,17),[5],[3]],[s(4,12,19),[6],[2]]],
M:
[[[1]], [[2]], [[3]], [[4]], [[5]], [[6]]]).

A:
M =[[[2], [4], [12], [19]], [[3], [1], [11], [17]], [[4], [5], [11], [16]], [[5], [16, 21], [7, 21],
    [4, 21], [16, 12], [7, 12], [4, 12], [16, 6], [7, 6], [4, 6]], [[6]], [[1], [6], [8], [14]]]
```

This predicate has three parameters, Q, M and L. Q is the same Q that is used in sporring2/6, and contains the common knowledge of the queries for every round. It contains the knowledge of which player performed the query, what state was queried, and the player that answered. M is a list of possibilities we will later use in analyze/3. L is a list containing a representation of every player, and the knowledge we already have of them.

With this information we can use the principles from the knowledge actions nocard, and showcard. Let us look at those knowledge actions again:

$\text{nocard}_b^{a,\{c,c',c''\}}$ $\qquad L_{123456}?(\neg c_b \wedge \neg c'_b \wedge \neg c''_b)$

$\text{showcard}_{b,c}^{a,\{c,c',c''\}}$ $\qquad L_{123456}?(!L_{ab}?c_{\mathbf{b}} \cup L_{ab}?c''_{\mathbf{b}} \cup L_{ab}?c''_{\mathbf{b}})$

So if a player does the nocard action, our agent can know that the player does not have any of the cards in question. showcard/3 however focuses on the showcard action. Which lets us know that both the player that has been asked and the player asking might know any of the cards in question.

The program starts by taking out the first query from Q, and then the first player in L. This player is called L1, and is divided into the identifier N and the previous state of possible knowledge B. In case L1 does not have any previous knowledge it only contains the identifier. Therefore the program first checks to see i [P]==L1, if it is the program calls on leggetil/3 which we will look at later. It then takes L3, which is the updated version of L1, and adds it to the rest of L. Making a new list L4. And starts a new round of showcard/3 with the next query. If L1 already contains information, we will see if P is the same as N, if it is the program calls on leggetil2/4, then starts a new round of showcard/3 with a similar fashion as earlier. If it neither N or L1 is the same as P we have the wrong player, and the program will put L1 at the back of L and start showcard/3 over again.

3.3.2 Now let’s look at leggetil/3:

```
leggetil(L,L2,Q):- Q= [[s(X,Y,Z),P1,P]|R], L=[N|B], B==[] -> L2=[N,[X],[Y],[Z]].
```

Example:

```
Q:
leggetil([[1]], L2, [[s(4,7,16),[1],[5]]]).
A:
L2 = [[1], [4], [7], [16]] ;
false.
```

It takes out the first in Q and the first player in L like in showcard/3 and then checks to see that B is indeed empty. If it is it adds each of the cards that was asked about in the query, knowing that the player learned one of them. Each of the lists in L2 are representative of possible knowledge. L2=[N,[X],[Y],[Z]] would mean that player N has learned X or Y or Z. next is leggetil2/4:

```
leggetil2(N1,NR,L,Q):- semi_perm(NR,Q,[],L1,NR),add(N1,L1,L).
```

```
Example:
Q:
leggetil2([[1]], [[4],[7],[16]] ,L, [[2],[11],[17]]).
A:
L = [[[1]], [16, 17], [7, 17], [4, 17], [16, 11], [7, 11], [4, 11], [16, 2], [7, 2], [4, 2]] ;
false.
```

If we look at the information we sendt to leggetil2/4 from showcard/3 we can see here is that the identifier for the player is named N1, the previous knowledge is named NR, L is the new list we want to generate and Q are the three cards in the query. Now in order to understand more of leggetil2 we must first look at semi_perm/5:

```
semi_perm(L,[],L2,L2,BL).
semi_perm([],N,L2,FL,BL):-    N =[N1|NR], semi_perm(BL,NR,L2,FL,BL).
semi_perm(L,N,L2,FL,BL) :-    L=[H|T], N=[N1|NR], first_last(N1,H,H2), add(H2,L2,L3),
                               semi_perm(T,N,L3,FL,BL).
```

```
Example:
Q:
semi_perm([[4], [7], [16]], [[2],[11],[17]], [] ,FL, [[4], [7], [16]]).
A:
FL = [[16, 17], [7, 17], [4, 17], [16, 11], [7, 11], [4, 11], [16, 2], [7, 2], [4, 2]]
```

semi_perm/5 has 5 parameters, both L and BL is the same list of previous knowledge at the start of the process. N are the cards, L2 is an empty list and FL is the final list we want to generate. It starts by dividing first state of possible knowledge from the rest of the states in L. H is the first state the rest of the knowledge states are called T. Then the first of the three cards in question is called N1 the two other NR. After this fist_last/3, which we will look at later, is used to put card N1 at the back of state H. The new list of H plus N1 is called H1. We then add H2 to the empty list L2, and start semi_perm/5 over again with T instead of L. The program will now do the process over again until the first card has been added in all knowledge states in L. Notice that at this point the program does not consider that there will be duplicates of a card in the different knowledge states. We will come back to the reason for that later.

When all the knowledge states has gotten the first card added and L is an empty list The first card N1 will be taken out of the equation and semi_perm/5 will start over now the the empty list L replaced by the list of previous knowledge states BL. This will be done until all three cards have been added to the new possibilities. Where there are no more cards list the new list of knowledge states will be returned to leggetil2/4. As you can see the amount of knowledge states will always be three times as many as the previous amount of knowledge states. This is because only one of the three cards in question has been learned. So each knowledge state will be made into 3 new ones. For example if the previous knowledge state is [[1]] and the cards in question are [[2],[3],[4]]. The new knowledge state will be

[[1,2],[1,3],[1,4]]. And if in the next round cards [[5],[6],[7]] are queried, the new state will be [[1,2,5],[1,3,5],[1,4,5],[1,2,6],[1,3,6],[1,4,6],[1,2,7],[1,3,7],[1,4,7]] and so on.

showcard2/3 is almost exactly the same as showcard/3 except it adds to the knowledge states of P1 instead of P.

```
showcard2([],L,L):-!.
showcard2(Q,M,L) :- Q= [[s(X,Y,Z),P1,P]|R],  L=[L1|L2],L1=[N|B], [P1]==L1 ->
                    leggetil(L1,L3,Q),first_last([L3],L2,L4),showcard2(R,M,L4);
                    Q= [[s(X,Y,Z),P1,P]|R],  L=[L1|L2],L1=[N|B], P1==N ->
                    leggetil2(N,B,L3,[[X],[Y],[Z]]),add(L3,L2,L4),showcard2(R,M,L4);
                    L=[L1|L2], first_last([L1],L2,NL),showcard2(Q,M,NL).
```

This will give us all the common knowledge that can be gained from the showcard knowledge action. At the moment though it does not include the query of done by player 1 as our agent is more concerned about what it knows as opposed to what the other players knows.

```
showcard2([[s(4,7,16),[1],[5]],[s(5,11,16),[2],[4]],[s(6,12,21),[3],[5]],[s(6,8,14),[4],[1]],
          [s(1,11,17),[5],[3]],[s(4,12,19),[6],[2]]],M,[
             [[2], [4], [12], [19]],
             [[3], [1], [11], [17]],
             [[4], [5], [11], [16]],
             [[5], [16, 21], [7, 21], [4, 21], [16, 12], [7, 12], [4, 12],
                 [16, 6], [7, 6], [4, 6]],
             [[6]],
             [[1], [6], [8], [14]]
          ]).
```

Gives the answer:

```
M = [
[[1], [14, 16], [8, 16], [6, 16], [14, 7], [8, 7], [6, 7], [14, 4], [8, 4], [6, 4]],
[[2], [19, 16], [12, 16], [4, 16], [19, 11], [12, 11], [4, 11], [19, 5], [12, 5], [4, 5]],
[[3], [17, 21], [11, 21], [1, 21], [17, 12], [11, 12], [1, 12], [17, 6], [11, 6], [1, 6]],
[[4], [16, 14], [11, 14], [5, 14], [16, 8], [11, 8], [5, 8], [16, 6], [11, 6], [5, 6]],
[[5], [4, 6, 17], [7, 6, 17], [16, 6, 17], [4, 12, 17], [7, 12, 17], [16, 12, 17], [4, 21, 17],
    [7, 21, 17], [16, 21, 17], [4, 6, 11], [7, 6, 11], [16, 6, 11], [4, 12, 11], [7, 12, 11],
    [16, 12, 11], [4, 21, 11], [7, 21, 11], [16, 21, 11], [4, 6, 1], [7, 6, 1], [16, 6, 1], [4, 12, 1],
    [7, 12, 1], [16, 12, 1], [4, 21, 1], [7, 21, 1], [16, 21, 1]],
[[6], [4], [12], [19]]] ;
false.
```

Before we move on let's look at first_last/3:

```
first_last(L,L2,L3) :- rev(L2,NL2),NL2=[H|T], add(H,L,Ll) , first_last2(Ll,T,L3).
```

Example:

```
Q:
first_last([1],[2,3,4],L).
A:
L = [2, 3, 4, 1] ;
false.
```

This predicate puts an element at the back of the list. it starts out by using rev/2 to reverse the order of the elements in L2, which is the list we want to put element L into at the last position. After reversing L2 and calling it NL2 we take out the first element and adds it to the list containing element L. This way the previously last element in L2 is now in a list with L, where L is now the last element. But we want to put the rest of the elements into the list as well, so we call on first_last2/3:

```
first_last2(L,[],L).
first_last2(L,L2,L3) :- L2=[H|T], add(H,L,Ll) , first_last2(Ll,T,L3).
```

As you can see first_last2/3 does almost the exact same thing as first_last/3 except for not reversing the list at the start. After all elements in L2 has been added to L, it returns to first_last/3 where L3 has become the new list there L is at the last position.

3.3.2 Now let's look at the nocard/3 predicate:

```
nocard([],M,M):-!.
nocard(Q,M,L):- Q= [[s(X,Y,Z),P1,P]|R], L= [L1|LR],L1=[I|C], P1==I ->
                first_last([L1],LR,L2), nocard2([X,Y,Z],L2,P,[[]],M1),nocard(R,M,M1);
                L= [L1|LR], first_last([L1],LR,L2), nocard(Q,M,L2).
```

Example:

Q:

```
nocard([[s(4,7,16),[1],[5]],[s(5,11,16),[2],[4]],[s(6,12,21),[3],[5]],[s(6,8,14),[4],[1]],
        [s(1,11,17),[5],[3]],[s(4,12,19),[6],[2]]],M,[
                [[1],[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21],[
                [[2],[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]],
                [[3],[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]],
                [[4],[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]],
                [[5],[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]],
                [[6],[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]]
        ]).
```

A:

```
M = [
[[1], [2, 3, 5, 6, 7, 8, 9, 10, 13, 14, 15, 16, 18, 20, 21]],
[[2], [2, 3, 5, 6, 8, 9, 10, 12, 13, 14, 15, 18, 19, 20, 21]],
[[3], [1, 2, 3, 6, 8, 9, 10, 12, 13, 14, 15, 17, 18, 19, 20, 21]],
[[4], [1, 2, 3, 5, 8, 9, 10, 11, 13, 14, 15, 17, 18, 19, 20]],
[[5], [1, 2, 3, 4, 5, 7, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 21]],
[[6], [2, 3, 4, 5, 7, 9, 10, 12, 13, 15, 16, 18, 19, 20, 21]]] ;
false.
```

The nocard/3 predicate removes the cards that a player says he does not have, by not being able to answer, from the list of possible cards for that player. The parameter Q is the same as in showcard/3. And the cards in question are being extracted from Q in the same way as well. After that the first player will be extracted from L. The representation of this player will at first be [[1], [1, 2, 3, 4, 5, 7, 8, 9, 10, 12, 13, 14, 15, 17, 18, 19, 20, 21]]. This list will be called L1 and is again divided into I and C for the identifier and the cards respectively. I is then compared to P1 which is the identifier of the player who did the query. If P1 and I are the same player, we want the program and remove cards from the next players. The reason for this is that we want to remove cards X,Y and Z from all the players between player P1 and player P because they used nocard in the game. So we continue by putting L1 at the back of the list of players. Then we this information to nocard2/5:

```
nocard2([],L,P,NL,M):- NL=[H|T],append(T,L,M).
nocard2(C,[],P,NL,M):- NL=[H|T],nocard2(C,T,P,[[]],M).
nocard2([X,Y,Z],LR,P,NL,M):- LR=[N|R], N=[I|C], I==P ->
                             nocard2([],LR,P,NL,M);  LR=[N|R], N=[I,C],
                             subtract(C,[X,Y,Z],C2),D=[[I,C2]],
                             first_last(D,NL,R2), nocard2([X,Y,Z],R,P,R2,M).
```

The same thing is done as in nocard/3, and the first player is extracted, and divided into identifier I, and cards C. We check if I and P is the same player. P is the player that did answer the query in showcard. So if I and P are the same, the process will be ended and no cards are removed. The way we do this is to call on nocard2/5 again with the end criteria satisfied. If I and P are not the same player, we know that player I does not have cards X,Y and Z. So we want to remove them. This is done by calling on subtract/3, which removes the cards from C if they are there. Updated list of cards will be called C2. We then put I and C2 together, and call it D, then put it at the last place with the list NL. The first round NL is just a list containing an empty list.NL= [[]]. The new list is called R2 and is used when we call on nocard2/5 again to check the next player. This is so that no players are lost in the process. This continues until I is the same as P and the end criteria is called. The end criteria is then the parameter containing the cards are empty. When this happens, the empty list in NL will be removed and the players in NL will be appended with the players in L. This way all the players are brought back to nocard/3

So back in nocard/3, nocard2/5 will have given us an updated list of cards, M1. We want to do this with all the queries of the round so we start a new nocard/3, with the rest of the queries, R and M1. To recap this is what happens when the identifier of the player who did the query P1, is the same as I. If P1 is not the same as I, the L1 will be put at the back of the list of players, and nocard/3 will be called upon again until P1 is the same as I. This continues until all queries have been accounted for. When no more queries are available the end criteria is called and M is returned as the same value as the newest update of M1/L.

3.3.3 Analyzer/5

The predicate analyzer/5 uses several other predicates to help analyze the results we get from showcard/3 and nocard/3. The predicates we will look at are:

analyzer/5
anSC/5
remover/5
remover2/5
assemble/4

The input analyzer gets are Q, W and L. Where Q is the queries from the current round, W is a variable containing several bits of information we will look at later. Finally L is the list of possible winstates. The output of analyzer/5 is W2, and L2. Which are the updated values from W, and L respectively.

analyzer/5 calls on showcard/3 but not showcard2/3. The reason for this is that in showcard/3 we can analyze that cards a player might have on their hand, and in showcard2/3 we can analyze what card a player might know. Getting to know a card a player holds is much more helpful than knowing that a player knows, a player holds that card. In terms of modal logic it is $Hand_{P1}(Card_1)$ compared to $K_{P2}(Hand_{P1}Card_1)$. $K_{P2}(Hand_{P1}Card_1)$ will be helpful information when creating a strategy for which card to ask for, or when the opponents are reasoning players, but at this stage in the program all we need to know is $Hand_{P1}(Card_1)$. Therefore showcard2/3 has been taken out of the analyzing process.

By knowing $Hand_{P1}(Card_1)$, we also know $\neg Hand_{P23456}(Card_1)$ therefore we can remove the possibility of those players having $Card_1$. We can also remove the possibility of $Card_1$ being in the winstate. So our goal of analyzing will be to get to know as many $Hand_X(Card_Y)$, as possible. The way we do this is to compare the results from showcard/3 to the results from nocard/3. showcard/3 gives us a list with a list of cards each player might have after a query. In other words, ^K($Hand_X(A \wedge B \wedge C \wedge ...)$) for every player.The length of the list is 3 in the power of the number of queries a player answered. Unless the player answered 0 queries then the length will be 0. So if the player answered 1 query the number of possibilities will be 3^1 = 3.

nocard/3 gives us a similar yet different list. nocard/3 also gives a list of cards a player might have, but it has removed the cards a player cannot have. ^K($Hand_X$(List)). If we now compare this list to every possibility in showcard/3, we can remove all possibilities that contains cards that are not in nocard/3. This way we remove all possibilities we can know is wrong. After the removal, if there is only one possibility left we now know that is the card or cards the player holds. We can then proceed to remove the card or cards from the nocard/3 lists of the other players, and from the possible winstates.

One problem with this method is if a player answers 3 or more queries in a round, then different combinations of the right cards or duplicates of the same card might keep the list length from getting down to one, because there are several combinations that are correct. This issue has not been resolved in the current version of the program.

Now let's look closer at the predicates:

```
analyzer(Q,W,L,L2,W2):-         W=[Win,ML,E],
                                showcard(Q,M,[[[1]], [[2]], [[3]], [[4]],[[5]],[[6]]],
                                nocard(Q,M2,ML),add(M,E,E2),  anSC(E2,M2,M2,[],N),
                                remover(N,M2,NM2,L,L2),W2=[Win,NM2,E2].



Q:
Analyzer(
        [[s(4,7,16),[1],[5]],[s(5,11,16),[2],[4]],[s(6,12,21),[3],[5]],[s(6,8,14),[4],[1]],[s(1,11,1
        7),[5],[3]],[s(4,12,19),[6],[2]]]
[
        s(1,7,19),
        [[[1], [8, 9,  15]],
        [[2], [1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21]],
        [[3], [1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21]],
        [[4], [1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21]],
        [[5], [1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21]],
        [[6], [1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21]]],
        []
],
[
        s(1,8,13),s(1,8,14),s(1,8,15),s(1,8,16),s(1,8,17),s(1,8,18),s(1,8,19),s(1,8,20),s(1,8,21),
        s(2,7,13),s(2,7,14),s(2,7,15),s(2,7,16),s(2,7,17),s(2,7,18),s(2,7,19),s(2,7,20),s(2,7,21)
]

        L2,
        W2
).
```

The first thing you should notice is that the W from unity has been modified to bring with it additional information to be used in analyzer. Later we will look at changes done in the program in order to implement analyzer into the program. But for now take note that W consists of 3 variables W=[Win,ML,E], where Win is the winning state, ML is the previous ^K(Hand$_X$(List)) that will be updated in nocard/3 this round, and E are the previous results from showcard/3, which we also want to compare to the updated list from nocard/3. L are all the winstates that are still a possibility. In this example a limited sample has been provided. With this we get the result:

```
A:

L2 =[s(2, 7, 13), s(2, 7, 14), s(2, 7, 15), s(2, 7, 16), s(2, 7, 17), s(2, 7, 18), s(2, 7, 19),
    s(2, 7, 20), s(2, 7, 21)],

W2 =    [s(1, 7, 19),
        [[[1], [8, 9, 15]],
        [[2], [2, 3, 5, 6, 10, 12, 13, 14, 18, 19, 20, 21]],
        [[3], [1, 2, 3, 6, 10, 12, 13, 14, 17, 18, 19, 20, 21]],
        [[4], [1, 2, 3, 5, 10, 11, 13, 14, 17, 18, 19, 20]],
        [[5], [1, 2, 3, 4, 5, 7, 10, 11, 12, 13, 16, 17, 18, 19, 20, 21]],
        [[6], [2, 3, 4, 5, 7, 10, 12, 13, 16, 18, 19, 20, 21]]],
        [[
        [[2], [4], [12], [19]],
        [[3], [1], [11], [17]],
        [[4], [5], [11], [16]],
        [[5], [16, 21], [7, 21], [4, 21], [16, 12], [7, 12], [4, 12], [16, 6], [7, 6], [4, 6]],
        [[6]],
        [[1], [6], [8], [14]]
        ]] ;
        false.
```

As you can see from the results all states containing 8 has been removed from L, and every player but 1 has had 8 removed from their list of possible cards. In addition the new ML has gone through nocard/3 and various cards from the player has been removed. Now the result from showcard/3 has been put in E, so that we can compare it to the the updated ML next round. Now let’s go through the process in the program:

```
analyzer(Q,W,L,L2,W2):-       W=[Win,ML,E],
                              showcard(Q,M,[[[1]], [[2]], [[3]], [[4]],[[5]],[[6]]],
                              nocard(Q,M2,ML),add(M,E,E2),  anSC(E2,M2,M2,[],N),
                                        remover(N,M2,NM2,L,L2),W2=[Win,NM2,E2].
```

It first divide W into the three variables [Win,ML,E] , runs showcard/3 using Q, and nocard/3 using Q and ML. It adds the result from showcard/3, M to E and calls it E2. The result from nocard/3, M2 is used toghter with E2 in anSC, to get N:

3.3.3.1 anSC/5

```
anSC(M,[],M2,NewM,N):- anSC(M,M2,M2,NewM,N).
anSC([],_,M2,M,R):- rev(M,R),!.
anSC([[]|T],_,M2,M,N):- anSC(T,M2,M2,M,N).
anSC([M|Rm],[M2|Rm2],M4,NewM,N):- M=[H1|T1],H1=[H|T], M2=[H2,T2], H==H2 ->
                                    assemble(T,T2,[],L1),add([H|L1],NewM,NewM2),
                                    anSC([T1|Rm],Rm2,M4,NewM2,N);M=[H1|T1],
                                    first_last([H1],T1,NM),
                                    anSC([NM|Rm],[M2|Rm2],M4,NewM,N).
```

```
Q:
        anSC([[
                [[2], [4], [12], [19]],
                [[3], [1], [11], [17]],
                [[4], [5], [11], [16]],
                [[5], [16, 21], [7, 21], [4, 21], [16, 12], [7, 12], [4, 12], [16, 6], [7, 6], [4, 6]],
                [[6]],
                [[1], [6], [8], [14]]]],
                [
                [[1], [8, 9, 15]],
                [[2], [2, 3, 5, 6, 8, 10, 12, 13, 14, 18, 19, 20, 21]],
                [[3], [1, 2, 3, 6, 8, 10, 12, 13, 14, 17, 18, 19, 20, 21]],
                [[4], [1, 2, 3, 5, 8, 10, 11, 13, 14, 17, 18, 19, 20]],
                [[5], [1, 2, 3, 4, 5, 7, 10, 11, 12, 13, 16, 17, 18, 19, 20, 21]],
                [[6], [2, 3, 4, 5, 7, 10, 12, 13, 16, 18, 19, 20, 21]]],
                [
                [[1], [8, 9, 15]],
                [[2], [2, 3, 5, 6, 8, 10, 12, 13, 14, 18, 19, 20, 21]],
                [[3], [1, 2, 3, 6, 8, 10, 12, 13, 14, 17, 18, 19, 20, 21]],
                [[4], [1, 2, 3, 5, 8, 10, 11, 13, 14, 17, 18, 19, 20]],
                [[5], [1, 2, 3, 4, 5, 7, 10, 11, 12, 13, 16, 17, 18, 19, 20, 21]],
                [[6], [2, 3, 4, 5, 7, 10, 12, 13, 16, 18, 19, 20, 21]]],
                [],
                N).
```

A:

```
N =
        [[[1], [8]],
        [[2], [19], [12]],
        [[3], [17], [1]],
        [[4], [11], [5]],
        [[5], [4, 12], [7, 12], [16, 12], [4, 21], [7, 21], [16, 21]],
        [[6]]] ;
```

anSC/5 starts by taking out the first element in E2, calls it M and then divides M it up in H1 = [[2], [4], [12], [19]] and the rest of the list T1. H1 is then divided to H=[2] and T=[[4], [12], [19]]. The first element of M2 is divided into two parts as well. H2 = [1] and T2 = [8, 9, 15]. It then checks if H and H2 are the same identifier. In this case it is not the same [2]=/=[1], so the first part fails, and H1 will be put at the back of T1 using first_last/3. The new list is called NM, and anSC/5 is called again, now with NM replacing M. This continues until H1 = [[1], [6], [8], [14]]. Then H and H2 both will be [1], and the program proceeds to call on assemble/4. We will later look at assemble/4, but what it does is uses the input from T2 = [8, 9, 15] and T= [[6], [8], [14]], to remove the states from T that cannot be found in T2. The result is [[8]] and will be called L1. H the identifier for L1 will be put back in its place and [[1],[8]], will be added to NewM, which until now has been an empty list. The new list containing [[1],[8]] is called NewM2. Then anSC/5 will be called again like this:
anSC([T1|Rm],Rm2,M4,NewM2,N) compared to the earlier

anSC([M|Rm],[M2|Rm2],M4,NewM,N). The head of M has been taken out and replaced by the tail (T1), M2 is no longer being used so it has been taken out, Rm2 is next, and The head of M is now in NewM2. This process continues until one of three things happen:

```
anSC(M,[],M2,NewM,N):- anSC(M,M2,M2,NewM,N).
```

If M2 is empty but there are still elements in M, the program will use the backup list of M2 to start the process again.

```
anSC([],_,M2,M,R):- rev(M,R),!.
```

If M is empty the process is done. We will reverse the NewM and return the list as R.

```
anSC([[]|T],_,M2,M,N):- anSC(T,M2,M2,M,N).
```

If the first element in M is now empty but there is still more elements, we replace M with the tail of M and continue the process.

3.3.3.2 Now let's look at assemble/4:

```
assemble([],T2,L,L).
assemble([H|T],T2,L,L1):- length(H,N),intersection(H,T2,H2),length(H2,N2), N==N2 ->
                          add(H2,L,M), assemble(T,T2,M,L1);assemble(T,T2,L,L1).
```

Q:

```
assemble([[6], [8], [14]],  [8, 9, 15],[],N).
```

A:

```
N = [[8]] .
```

assemble/4 will take out the first element in the input H=[6], and look at its length. In this case N is 1. It then takes the intersection of [6] and [8,9,15]. The result is H2 =[]. intersection/3 in a inbuilt predicate in prolog. Now the length of H2 is measured. N2=0. If N and N2 are the same we want to keep H2, and put it in L, then brings it with us for the rest of the tests. If it is not the same, like in this case, we will not add it to L.

At the end the result is the list of the variable who made it through the test. This test also works when a player has been queried more than once in a round. For instance [8,9] would make it through but [6,8] would not.

Going back to analyzer/5. After getting an answer from anSC/5 we will will use that answer in remover/5:

3.3.3.3 remover/5

```
remover([],M2,M2,L,L).
remover([H|T],M2,NM2,L,L2) :-     length(H,Num), Num==2 ->
                                  H=[H1,T1],remover2(H1,T1,M2,M3,0),add(H1,T1,T
                              3),
                                  run2(L3,T3,L), remover(T,M3,NM2,L3,L2);
                                  remover(T,M2,NM2,L,L2).
```

Example:

```
Q:
        remover([
                [[1], [8]],
                [[2],[19],[12]],
                [[3],[17],[1]],
                [[4],[11],[5]],
                [[5],[4,12],[7,12],[16,12],[4,21],[7,21],[16,21]],
                [[6]]
                ],
                [[[1],[8,9,15]],
                [[2],[2,3,5,6,8,10,12,13,14,18,19,20,21]],
                [[3],[1,2,3,6,8,10,12,13,14,17,18,19,20,21]],
                [[4],[1,2,3,5,8,10,11,13,14,17,18,19,20]],
                [[5],[1,2,3,4,5,7,10,11,12,13,16,17,18,19,20,21]],
                [[6],[2,3,4,5,7,10,12,13,16,18,19,20,21]]],
                NM2,
                [s(1,8,13), s(1,8,14), s(1,8,15), s(1,8,16),  s(1,8,17),s(1,8,18), s(1,8,19),
                s(1,8,20), s(1,8,21), s(2,7,13), s(2,7,14), s(2,7,15), s(2,7,16), s(2,7,17),
                s(2,7,18), s(2,7,19), s(2,7,20), s(2,7,21)],
                L2).


A:
        NM2 = [
                [[1], [8, 9, 15]],
                [[2], [2, 3, 5, 6, 10, 12, 13, 14, 18, 19, 20, 21]],
                [[3], [1, 2, 3, 6, 10, 12, 13, 14, 17, 18, 19, 20, 21]],
                [[4], [1, 2, 3, 5, 10, 11, 13, 14, 17, 18, 19, 20]],
                [[5], [1, 2, 3, 4, 5, 7, 10, 11, 12, 13, 16, 17, 18, 19, 20, 21]],
                [[6], [2, 3, 4, 5, 7, 10, 12, 13, 16, 18, 19, 20, 21]]],

        L2 =    [s(2, 7, 13), s(2, 7, 14), s(2, 7, 15), s(2, 7, 16), s(2, 7, 17), s(2, 7, 18),
                 s(2, 7, 19), s(2, 7, 20), s(2, 7, 21)] ;
false.
```

The inputs here are the result from anSc/5 = [H|T], the result from nocard/3 = M2, and the list of possible winstates = L.

The first thing the predicate does is to check if the length of H is 2. If it is that means that we know a player holds the card or cards in the second list of H. If the length is more than 2 there are still other possibilities. And the program will move on and call remover/5 again without H. If the length is 2 it will divide H into the identifier H1, and the list of cards T1. Then call on remover2/5. In this case the length of [[1], [8]] is 2, so it will be sent to remover2/5 together with M2 and a 0:

```
remover2(_,_,M,M,5):- !.
remover2(H1,T1,[M|MR],M4,N):-   N1 is N+1,  M=[H,T], H1=\=H ->subtract(T,T1,T3),
                                            first_last([[H,T3]],MR,M3),remover2(H1,T1,M3,M4,N1);
                                            first_last([M],MR,MR2),remover2(H1,T1,MR2,M4,N).
```

Example:

Q:

```
        remover2(
                [1],
                [8],
                [
                [[1], [8, 9, 15]],
                [[2], [2, 3, 5, 6, 8, 10, 12, 13, 14, 18, 19, 20, 21]],
                [[3], [1, 2, 3, 6, 8, 10, 12, 13, 14, 17, 18, 19, 20, 21]],
                [[4], [1, 2, 3, 5, 8, 10, 11, 13, 14, 17, 18, 19, 20]],
                [[5], [1, 2, 3, 4, 5, 7, 10, 11, 12, 13, 16, 17, 18, 19, 20, 21]],
                [[6], [2, 3, 4, 5, 7, 10, 12, 13, 16, 18, 19, 20, 21]]
                ],
                M4,
                0).
```

A:

```
        M4 = [
                [[1], [8, 9, 15]],
                [[2], [2, 3, 5, 6, 10, 12, 13, 14, 18, 19, 20, 21]],
                [[3], [1, 2, 3, 6, 10, 12, 13, 14, 17, 18, 19, 20, 21]],
                [[4], [1, 2, 3, 5, 10, 11, 13, 14, 17, 18, 19, 20]],
                [[5], [1, 2, 3, 4, 5, 7, 10, 11, 12, 13, 16, 17, 18, 19, 20, 21]],
                [[6], [2, 3, 4, 5, 7, 10, 12, 13, 16, 18, 19, 20, 21]]] ;
                false.
```

In remover2/5 we want to remove T1, in this case [8], from all players in M2, except for the player who has the card on his hand, [1]. That means we want to stop the process after the maximum of 5 times. I say maximum 5, because if there are less than 6 players the number does not need to go as high as 5, but it does not matter if a player is checked more than once in those cases. Because of this max 5 number we add 1 to N for every player we check. N+1 is N1. Then we take out the first player in M2, and checks that the identifiers are not the same. If they are the same, this part of the program fails and it jumps to the next part when it puts the first player at the last place in the list using first_last/3 and calls for remover2/5 to go again with the updated M2. If the identifiers are not the same, the program will remove the cards in T1 from the list of possible cards. Before it put the updated list of the first player at the back of M2, and calls on remover2/5 again. This goes on until N reaches 5. N will only go one higher if the identifiers don't match. When N is 5 remover2/5 break off using !, and return the updated M2 to remover/5.

Now remover will put H1 and T1 together and call for run2/3 to remove all cards in T1 from the possible winstates. After that it will call on remover/5 to go again with the next state in N. Then N is empty it will return the updated M2 and the updated L back to analyzer/5. analyzer/ will then put W together with the updated values, and return it as W2 together with the updated L called L2.

One problem with this removal process is that it does not check to see if all the cards a player has on his hand is known, and then remove the rest. For instance in a 6 player game, only 3 cards can be held by a player. So if you know 3 cards for a player, the rest of the cards can be removed from the list. This does however not make a difference to the information gain of the program at this point, but it will matter if we want to know the cards a player knows of, instead of what he has on his hand. More of this later.

3.4 Implementing modal.pl into untiy.pl.

In order to implement the modal logic into unity.pl a few changes had to be made. The first change we can see in players/1, changes are shown in bold:

```
players(X):- dealer(L,X,Win),mMaker(X,M),L =[P|T],P=[H|T2],
remover2(H,T2,M,M2,0),W=[Win,M2,[]],winstates(WS),trekkfra(P,WS,T,W).
```

The first change is that we have a new predicate called mMaker/2, lets look at that firstly.

mMaker/2 is now a part of dealer.pl and looks like this:

```
mMaker(X,M):-list(X,L,L0),cDeal(L0,[],M).

cDeal([],M,R):- rev(M,R),!.
cDeal([H|T],M,M1):- cards2(C),append(H,[C],H1), add(H1,M,M2), cDeal(T,M2,M1).
```

The purpose of mMaker/2 is to make the list representing the players used in nocard/3. The input in mMaker is X, which is the amount of players, and the output is M which is the list used in nocard.

Example:

```
Q:
        mMaker(6,M).
A:

        M = [
                [[1], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]],
                [[2], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]],
                [[3], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]],
                [[4], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]],
                [[5], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]],
                [[6], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]]] ;
        False.
```

It does this with the use of list/3 and cDeal/3. list/3 makes the list with the identifiers, like in dealer/3, and cDeal/3 puts all the cards into each of the lists. Next change in players/1 is **P=[H|T2], remover2(H,T2,M,M2,0),** P is divided into identifier H and cards T2, and is then used in remover2/5 to remove all the cards the agent was dealt from the other players in M. The result of this is M2.

If the agent was has the deal [[1],13,17,16], M2 will be:

```
[[[1],[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]],
[[2],[1,2,3,4,5,6,7,8,9,10,11,12,14,15,18,19,20,21]],
[[3],[1,2,3,4,5,6,7,8,9,10,11,12,14,15,18,19,20,21]],
[[4],[1,2,3,4,5,6,7,8,9,10,11,12,14,15,18,19,20,21]],
[[5],[1,2,3,4,5,6,7,8,9,10,11,12,14,15,18,19,20,21]],
[[6],[1,2,3,4,5,6,7,8,9,10,11,12,14,15,18,19,20,21]]]
```

The final change in players/6 is **W=[Win,M2,[]]**. This is the new way W is put together. Before it was only Win, now we also have M2 and an empty list which we will use to put in results from analyzer/5. Next are the changes made in trekkfra2/6:

```
trekkfra2([A1|A2],WS,T,P,W,Q):- W=[X,M,Z],A1==[0],win(X,A2);run2(L,[A1|A2],WS),
                                W=[X,M,Z],remover2(A1,A2,M,M2,0),
                                W2=[X,M2,Z], sporring2(P,T,L,W2,Q).
```

trekkfra2/6 has had similar adjustments to the new W, and an adding of remover2/5 to remove the cards from all other players than the one who answered our query. win/2 has also had a small adjustment:

```
win(W,[(X,Y,Z)]):-W==s(X,Y,Z),true;fail.
```

[(X,Y,Z)] is now in a list, before i was not. Next is sporring2:

```
sporring2(T1,[],L1,Q,L,W):-     append(L1,[T1],L2),L2=[P|T],rev(Q,R),
                                analyzer(R,W,L,NewL,W1),sporring(NewL,T,P,W1).
```

This is where analyzer/5 has been implemented. And the updated L and W are given to sporring/4. How the console looks when running players(6). after the implementation of analyzer/5 can be seen in appendix B.

4 Discussion:

4.1 Cluedo mechanics:

When implementing the game mechanics for cluedo, many of the mechanics were cut out. Things that are crucial to the game, such as the game board and the dice was not implemented. Because of this the mechanic where a player has to walk to a room in order to do a query was also omitted. But the focus of the thesis was to see if prolog could be used to implement dynamic epistemic logic to work in the environment of Cluedo. And for this purpose many of the game mechanics of Cluedo were not necessary to implement.

Only the mechanics needed in order to create an environment where dealings and change of the epistemic state were focused on. The game board and the dice did not add to this, and were therefore cut out of the implementation. The dealing of cards and the query among the players has a much higher importance in regards to epistemic state. This is done in dealer.pl and query.pl. In order to be able to deal the cards they had to be implemented explicitly into the program as facts in cluedo.pl. Part of the dealing of cards, is to deal the winstate. Therefore also the winstates needed to be written as facts. In addition to the dealing of cards and queries, the program also needed a predicate to take the game from one epistemic state to another. This is done in subtracter.pl. With these three mechanics, and the facts to back them up, all that was needed in order to start implementing the dynamic epistemic logic was to bring it all together. Which is what unity.pl is for. In unity.pl a game of Cluedo is played successfully, which shows us that prolog is capable of implementing these kinds of mechanics.

One might argue that this is no longer Cluedo, since several of the key parts that make it into the game it is has been removed. This might be true, but the underlying game principles concerning knowledge, still holds true to the game. Therefore I will argue that this is a successful implementation of Cluedo. Now let us see to what extent the implementation of dynamic epistemic logic holds true to the theory.

4.2 Definition 1 - Language

Looking back at the first definition of dynamic epistemic logic we have:

$$\mathrm{L_A(P)}\ \textit{is the smallest set such that, if } p \in P, \varphi, \psi \in \mathrm{L_A(P)}, a \in A, B \subseteq A, \alpha \in \mathrm{KA_A(P)},$$
$$\textit{then } p, \neg\varphi, (\varphi \wedge \psi), K_a\varphi, C_B\varphi, [\alpha]\varphi \in \mathrm{L_A(P)}.$$

For P we have our 324 winstates plus the 21 unique cards. We also know there are 44 460 928 512 000 other potential values for p, but decided to exclude them because of the large size.

As for $\psi \in \mathrm{L_A(P)}$, we have the following predicates:

person(p)

weapon(p)

location(p)

s(p,p',p'')

But can also be represented as a list of individual cards:

When N is between 1-21: $[p_1,p_2,p_3...p_N]$

Essentially person(p), weapon(p) and location(p) are never really used in the program. This leaves us with s(p,p',p'') and $[p_1,p_2,p_3...p_N]$ as the expressions for P.

Next on the list is $a \in A$ this is represented in the program as a list that is an identifier. Being [1],[2],[3],[4],[5], and [6]. When it comes to $B \subseteq A$ which is a group of players it works in a similar way. Any combination of groups can theoretically be made between the players by combining the identifiers. For instance [[1],[2],[3]]. The knowledge actions for the players $\alpha \in \mathrm{KA_A(P)}$, we will look at in detail later.

From this the language is inductively defined as: $\textit{then } p, \neg\varphi, (\varphi \wedge \psi), K_a\varphi, C_B\varphi, [\alpha]\varphi \in \mathrm{L_A(P)}$.

Which we now know is for example:

$p = 1$

$\neg\varphi = \neg[2] = [1,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]$

$(\varphi \wedge \psi) = [[1] \wedge s(1,7,13)]$

$K_a\varphi = [[1],1,2,3]$

$C_B\varphi = [[1],[3],1,7,13]$

$[\alpha]\varphi = [nocard_2([1,7,13])]\ [[2],2,3,4,5,6,8,9,10,11,12,14,15,16,17,18,19,20,21]$

This shows us that the implementation in prolog has the possibility to use the full range of the language if needed.

4.3 Definition 2 - Knowledge actions.

Now let us look at the definition for KA_A:

$$\textit{Given a set of agents } A \textit{ and a set atoms } P,\ \textit{the set of knowledge actions } \mathsf{KA}_A(\mathsf{P})$$
$$\textit{is the smallest set such that, if } \varphi \in \mathsf{L}_A(\mathsf{P}), \alpha, \alpha' \in \mathsf{KA}_A(\mathsf{P}),\ B \subseteq A,\ \textit{then}:$$
$$?\varphi, \mathsf{L}_B\,\alpha, (\alpha\ ;\ \alpha'),\ (\alpha \cup \alpha'), (\alpha\ !\ \alpha'), (\alpha \cap \alpha') \in \mathsf{KA}_A(\mathsf{P})$$

Gives us these knowledge actions:

$nocard_b^{a,\{c,c',c''\}}$	$L_{123456}?(\neg c_b\ \wedge \neg c'_b\ \wedge \neg c''_b)$
$showcard_{b,c}^{a,\{c,c',c''\}}$	$L_{123456}?(!L_{ab}?c_{\mathbf{b}} \cup L_{ab}?c''_{\mathbf{b}} \cup L_{ab}?c''_{\mathbf{b}})$
$endmove^a$	$L_{123456}?\neg K_a\delta^0$
win^a	$L_{123456}?K_a\delta^0$

4.3.1 Nocard

$nocard_b^{a,\{c,c',c''\}}$ $\quad L_{123456}?(\neg c_b\ \wedge \neg c'_b\ \wedge \neg c''_b)$

Starting with the nocard action, and comparing it to the program, we can notice a few things. First if we put in values for the variable in the formula $(\neg c_b \wedge \neg c'_b \wedge \neg c''_b)$, one example of what we can get is $(\neg 1_1 \wedge \neg 7_1 \wedge \neg 13_1)$, from this using $\neg\varphi$ and $(\varphi \wedge \psi)$ we get:

([[1],2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21] $\wedge$
[[1],1,2,3,4,5,6,8,9,10,11,12,13,14,15,16,17,18,19,20,21] $\wedge$
[[1]1,2,3,4,5,6,7,8,9,10,11,12,14,15,16,17,18,19,20,21])

When all these three states are satisfied we get the answer:

[[1],2,3,4,5,6,8,9,10,11,12,14,15,16,17,18,19,20,21]

Which nocard/3 will provide. What nocard/3 does not do is make a representation for $L_{123456}?(\neg c_b \wedge \neg c'_b \wedge \neg c''_b)$, it only make the representation for $L_1?(\neg c_b \wedge \neg c'_b \wedge \neg c''_b)$, assuming the agent is player 1. The reason for this is not that the knowledge learned is not common knowledge in the program, but it is because the preexisting knowledge state that is brought into nocard/3 contains the agent's private knowledge. This can be avoided by creating a list for every player of how their knowledge state would look like, based on the agent's knowledge about the players. Having that knowledge could also have an impact when choosing states to query, or when combined with showcard/2. However at the current state of the program it will not make any difference, since the agent also learns the same knowledge.

4.3.2 Showcard

Now let's look at the showcard action:

$\text{showcard}_{b,c}^{a,\{c,c',c''\}}$ $\quad L_{123456}?(!L_{ab}?c_{\mathbf{b}} \cup L_{ab}?c''_{\mathbf{b}} \cup L_{ab}?c''_{\mathbf{b}})$

Setting values to the variable in this case $(!L_{ab}?c_{\mathbf{b}} \cup L_{ab}?c''_{\mathbf{b}} \cup L_{ab}?c''_{\mathbf{b}})$, becomes $(!L_{12}?1_{\mathbf{2}} \cup L_{12}?7_{\mathbf{2}} \cup L_{12}?13_{\mathbf{2}})$ which in the terms of the program looks like this:

(!L[[1],[2],7]$\cup$[[1],[2],7]$\cup$[[1],[2],13])

The answer showcard/3 gives is:

[[2],[1],[7],[13]]

And showcard2/3 gives:
[[1],[1],[7],[13]]

Again the program only gets the result that is the most relevant to the agent. It does learn that player 2 holds one of the three cards and that player 1 knows one of the three cards, but does not register now both player 1 and 2 knows that player 1 knows the card. This is not immediately a game changer for the agent, but if enough information about the other players is gained this can become relevant information. Since the opponents just pick cards at random in the program, this has not yet been implemented.

4.3.3 Endmove and win

The two remaining knowledge actions are then endmove and win

endmove^a	$L_{123456}?\neg K_a \delta^0$
win^a	$L_{123456}?K_a \delta^0$

These two actions are similar and opposite. One tells all players that a player does not know the winning hand, the other tells them that he does know. win is not a very interesting knowledge action to our agent in terms of the game, because if win is used the game is over. endmove however can be useful. Example:

> If the agent knows the winstate is s(X.7.13), and its knowledge about player 2's hand is [[2],1,2,5,8,11] and a query of player two was (s(1,8,13),[2],[0]). And player 2 then does the endmove action. This must mean that player 2 owns card 8 and that card 1 is in the winstate. And now the agent knows the full winstate.

The reason the agent can learn the winstate is because it already knows 7 and 13 is in the winstate, thereby knowing that 8 is not in the winstate, because it is in the same category as 7. And since no one could answer player 2's query, 1 must be in the winstate and 8 must be in player 2’s hand. Since the opponent always will use endmove even if they randomly hit the winstate this has not yet been implemented into the program.

4.4.1 Analyzer

For analyzer/5 let’s focus on the one that is most important for this predicate. And that is $M, w \models \varphi \wedge \psi$. What anSC/5 does is to take the results from showcard (M) and the result from nocard (M2) and seeing where $M \wedge M2$ is true. In other words $Mo, w \models M\ and Mo, w \models M2$. The states $m \in M$ where $m \wedge M2$ is true are kept and the others are discarded.

This shows us that all the knowledge actions can be implemented into prolog.

4.4 Comparing agent using modal logic vs not.

In order to test the difference in performance in when the program used the modal logic vs when it did not. A predicate designed to count the number of rounds a game would go on was created.

```
counter(X,0,N2,R,R2):- R2 is R/N2,!.
counter(X,N,N2,R,R2):-N3 is N-1,players(X,U), R3 is R + U, counter(X,N3,N2,R3,R2).
```

Unity.pl was slightly modified so that it would be able to cound the rounds. After this counter/5, would call on players/2 which is the modified players/1. It will call for X amounts of players for N amount of rounds, N2 is also the same amount as N. players/2 will return U every round, which is the number of rounds the game of cluedo took. U will be added to R and is called R3. N3 is the new counter, and is N-1. Then a new round is called for with N3 and R3 taking the place of N and R. This goes on until N reaches 0. The R which is the sum of all U’s will be divided by N2. This way we get the average number of rounds it took to complete a game of cluedo. The reason rounds was made the indicator of the agent’s performance is because the knowledge gain is calculated at the end of every round. And an agent using fewer rounds got to the right answer the fastest and therefore has the highest performance.

Result:
Without using modality.pl in the program the average rounds in 100 round was 13.43:

```
    Q:
            counter(6,100,100,0,R).

    A:
            R = 13.43 ;
            false.
```

When the modality.pl was used the average in 100 rounds was 8.05 :

```
    Q:
            counter(6,100,100,0,R).

    A:
            R = 8.05 ;
            false.
```

That means that player/2 with modality.pl activated on average use 5.38 less rounds to find the correct winstate. That is quite considerable since 5.38 is 66.8% of the total of 8.05, and 40% of the total of 13.43. Meaning the program using modal.pl uses 40% less rounds. And 1.668 times faster. Bear in mind that a program that fully uses the principles of dynamic epistemic logic would find the answer in even less amount of rounds. When testing an agent who operates completely random, we get these values:

```
    Q:
            counter(6,10,10,0,R).

    A:
            R = 134.3
```

The random agent has a smaller sample size because of memory issues with prolog. However when comparing this number to the agent using modal.pl, we see that it uses 94% less rounds than the random agent. That is more than 16.5 times faster. As you results show the agent using modality.pl has a significant increase in its performance over both the random agent and the agant not using modal.pl. This shows us that that the mechanics from dynamic epistemic logic has been successfully implemented into prolog.

4.5 Prolog:

One of the main problems I encountered when using prolog was the lack of memory to go through with the operation. A query would simply stop midway because all the memory allotted had been used. In a 32-bit system the default memory given to prolog is 128 MB per are of local, global and trail memory. And for a 64-bit system the double of that at 256 MB.This is not a lot and, in several cases it was not enough. Luckily there is an inbuilt predicate that gives you the option to increase the amount of memory available to the program. The set_prolog_stack/2 predicate has been very helpful in dealing with this problem.

4.6 Strategies and further development:

The program is still flawed in the sense that it still only chose a state to query randomly. Using the knowledge gained from analyzer/5, it is possible to pick a state that gives the agent the highest possible information gain, while keeping it low for the rest of the players. In order to do this, the information of what other players know about each other is of higher importance. Therefore an implementation of showcard/2 into analyzer/5 would be necessary.

One strategy in order to get the most knowledge from the nocard knowledge action, would be to ask only for unknown cards from player 6, that many players share. Assuming the agent is player 1, the most knowledge is gained from nocard if player 6 is the one who answers. If the query reaches player 6, 3 cards can be eliminated from each player. This is a risky strategy for several reasons. The first being that all the information gained in nocard is general knowledge. So all the players will learn the same as you. The second is that there is no way of knowing which player will actually answer the query..

A slightly more subtle strategy would be to design a query to give you an answer from a specific player, or for a specific card. For instance if you want an answer from player 3, you should pick a state where you know player 2 cannot answer it. Or if you can ask for a card mixed with cards from your own hand, or a card you know to be in the winstate. This strategy also has risks. You can give other players knowledge by making players use nocard. Or you might ask for a card that none of the opponents are carrying, letting them know the cards you queried is either in the winstate or on your hand.

Another thing the analyzer/5 should be able to do, is to check M2 for numbers all players are missing. If for instance, M2 after a few rounds of eliminating cards looks like this:

```
[[1], [ 8, 9, 15,]],
[[2], [3, 5, 6, 10, 14,  18, 19]],
[[3], [1, 2, 3, 6, 10, 13, 14, 17, 18],
[[4], [1, 2, 3, 5, 10, 11, 13, 14, 17, 20]],
[[5], [1, 2, 3, 5, 7, 10, 11, 13, 16, 17, 18, 21]],
[[6], [2, 3, 5, 7, 13, 16, 18]]]
```

The agent can go through the list and find out that card 4 and 12 are missing from all the players. That means that card 4 and 12 must be in the winstate. This is important, because the winstate is divided into categories. So knowing 4 and 12 means the agent know the person and weapon category. And can eliminate all other cards in those two categories. Meaning the agent can remove states containing cards [1,2,3,5,6,7,8,9,10,11] from the list of possible winstates.

If the game is with 6 players an agent can know that every player has 3 cards on his hand. An implementation should be made into analyzer where known cards are marked. This way the agent can check for the amount of known cards per player and if it is 3, the agent can remove all other cards from the player in M2. This does not have an immediate impact the way the program is now, but when the knowledge from showcard2/3 is better used, this will help the information gain, in knowing which cards players know other players have.

Another implementation that can be made is to start analyzer/5 over again every time M2 is updated in remover/5. The updated version of M2 can be used in anSC/5 and might get more known states that can be removed in remover/5, which can again give an updated version of M2. This process should go on until no more information can be gained. To illustrate this imagine the following scenario:

We have M2:

```
[[1], [1, 2, 8, 9, 12, 15]],
 [[2], [2, 3, 5, 6, 8, 10, 13, 14, 18, 19, 20, 21]],
 [[3], [1, 2, 3, 6, 8, 10, 12, 13, 14, 17, 18, 19, 20, 21]],
 [[4], [1, 2, 3, 5, 8, 10, 11, 13, 14, 17, 18, 19, 20]],
 [[5], [1, 2, 3, 4, 5, 7, 10, 11, 12, 13, 16, 17, 18, 19, 20, 21]],
 [[6], [2, 3, 4, 5, 7, 10, 12, 13, 16, 18, 19, 20, 21]]],
```

And the result of anSC/5 is:

```
N = [[[1], [8]],
[[2], [19],[8]],
[[3], [17], [1]],
[[4], [11], [5], [8]],
[[5], [4, 12], [7, 12], [16, 12], [4, 21], [7, 21], [16, 21]],
[[6]]] ;
```

After going through remover/5 M2 will now look like this:

```
 [[1], [1, 2, 8, 9, 12, 15]],
 [[2], [2, 3, 5, 6, 10, 13, 14, 18, 19, 20, 21]],
 [[3], [1, 2, 3, 6, 10, 12, 13, 14, 17, 18, 19, 20, 21]],
 [[4], [1, 2, 3, 5, 10, 11, 13, 14, 17, 18, 19, 20]],
 [[5], [1, 2, 3, 4, 5, 7, 10, 11, 12, 13, 16, 17, 18, 19, 20, 21]],
 [[6], [2, 3, 4, 5, 7, 10, 12, 13, 16, 18, 19, 20, 21]]],
```

Normally the program will move on now, but if we were to send the updated M2 to anSC/5 now, we would get:

```
N = [[[1], [8]],
[[2], [19]],
[[3], [17], [1]],
[[4], [11], [5]],
[[5], [4, 12], [7, 12], [16, 12], [4, 21], [7, 21], [16, 21]],
[[6]]] ;
```

Now we can also remove 19 from all players except player 2, and also from winstates. The program as it is now will wait until the next round before it does this. This is just some of the possibilities for further implementation.

5 Conclusion:

For the conclusion of the thesis let's go through each of the research questions one by one.

- Is Prolog suitable for implementations of Cluedo game mechanics?

In section 4.1 we have looked at the parts needed to make a game of Cluedo run. And how they were successfully implemented into prolog. This shows that Prolog is a suitable language for this kind of programming. However there are challenges described in 4.5 that shows that prolog might not be the best fit.

- Is Prolog suitable for implementations of dynamic epistemic logic?

Section 4.2 shows us that the language for dynamic epistemic logic can be represented in Prolog. Which does makes Prolog suitable for this kind of implementation.

- How can Cluedo's knowledge actions, formalised in dynamic epistemic logic, be implemented in Prolog?

Section 4.3 shows us how the knowledge actions can be represented. In addition the knowledge actions showcard and nocard has been successfully implemented into Prolog.

- How does an agent using dynamic epistemic logic compare to a navie agent that does not?

Section 4.4 shows us that the the agent using a limited form of dynamic epistemic logic increased the efficiency by 66.8% when compared to an agent that did not. In 100 games of Cluedo the average number of rounds to get the correct answer was 8.05 for the agent using dynamic epistemic logic. Compared to 13.43 rounds for the one that did not.

References:

[1] Thomas Agotnes and Hans van Ditmarsch. What will they say? - public announcement games. Synthese (Special Section on Knowledge, Rationality and Action), 179(1):57-85, 2011.
[2] A. Baltag, L.S. Moss, and S. Solecki. The logic of public announcements, common knowledge, and private suspicions. In I. Gilboa, editor, Proceedings of the 7th Conference on Theoretical Aspects of Rationality and Knowledge (TARK 98), pages 43-56, 1998.
[3] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. Reasoning About Knowledge. The MIT Press, Cambridge, Massachusetts, 1995.

[4] H. van Ditmarsch. Descriptions of game actions. Journal of Logic, Language and Information, 11:349-365, 2002.
[5] H.P. van Ditmarsch. Knowledge games. PhD thesis, University of Groningen, 2000. ILLC Dissertation Series DS-2000-06.
[6] H.P. van Ditmarsch. The description of game actions in cluedo. In L.A. Petrosian and V.V. Mazalov, editors, Game Theory and Applications, volume 8, pages 1-28, Commack, NY, USA, 2002. Nova Science Publishers.
[7] H.P. van Ditmarsch, W. van der Hoek, and B.P. Kooi. Dynamic Epistemic Logic, volume 337 of Synthese Library. Springer, 2007.
[8] Jan van Eijck. Demo - a demo of epistemic modelling. In Interactive Logic, pages 305-363. Amsterdam University Press, 2006.
[9] "Clue" Written in Prolog Gameplay - GrinCo.org, video, Grinko, 13 January 2013, viewed 31 May 2016, <https://www.youtube.com/watch?v=rZ34Df9S36I>.
[10] S. Tatimoto, A Tic-Tac-Toe program in Prolog, 2003, accessed31 May 2016 , <https://courses.cs.washington.edu/courses/cse341/03sp/slides/PrologEx/tictactoe.pl.txt>
[11] Martin Osermann,Example Prolog Chess Program, 1998, accessed 31 May 2016, <http://familie-ostermann.de/Martin/chess/>
[12]Lectures for CSE 319 (Spring 2002), Modelling Games with Prolog Expert Systems, 2002, accessed 31 May 2016, <http://ed.loper.org/presentations/cse391_games.pdf>
[13] List of versions and spin-offs ,accessed 31 May 2016, <http://www.theartofmurder.com/cluedo_games.html>
[14] H.P. van Ditmarsch, W. van der Hoek, and B.P. Kooi. Playing Cards with Hintikka - An Introduction to Dynamic Epistemic Logic, 2004.

Appendix A.

Query of player(6). Before modal.pl

```
31 ?- players(6).

deal:[[[1],15,8,9],[[2],19,10,18],[[3],17,13,6],[[4],11,14,1],[[5],21,7,16],[[6],3,2,5]]
Win:s(4,12,20)
my cards:[[1],15,8,9]
states left:192
P: [[1],15,8,9]
```

```
QM:s(4,7,16)
A:[[5],7]
states left:144
Q:[[s(4,7,16),[1],[5]],[s(5,11,16),[2],[4]],[s(6,12,21),[3],[5]],[s(6,8,14),[4],[1]],[s(1,11,17),[5],[3]],
[s(4,12,19),[6],[2]]]bf
P: [[1],15,8,9]
QM:s(2,11,17)
A:[[3],17]
states left:126
Q:[[s(2,11,17),[1],[3]],[s(6,10,17),[2],[3]],[s(4,7,18),[3],[5]],[s(4,12,20),[4],[0]],[s(3,11,13),[5],[6]
],[s(6,9,16),[6],[1]]]
P: [[1],15,8,9]
QM:s(5,11,19)
A:[[2],19]
states left:108
Q:[[s(5,11,19),[1],[2]],[s(6,7,14),[2],[3]],[s(1,10,20),[3],[4]],[s(1,7,16),[4],[5]],[s(1,10,19),[5],[2]],
[s(1,7,13),[6],[3]]]
P: [[1],15,8,9]
QM:s(3,10,18)
A:[[2],10]
states left:72
Q:[[s(3,10,18),[1],[2]],[s(6,11,17),[2],[3]],[s(6,7,21),[3],[5]],[s(6,7,18),[4],[5]],[s(5,8,15),[5],[6]],[s
(3,7,18),[6],[2]]]
P: [[1],15,8,9]
QM:s(6,11,13)
A:[[3],13]
states left:60
Q:[[s(6,11,13),[1],[3]],[s(4,10,16),[2],[5]],[s(4,12,20),[3],[0]],[s(2,7,18),[4],[5]],[s(5,12,15),[5],[6]
],[s(2,7,20),[6],[5]]]
P: [[1],15,8,9]
QM:s(3,12,21)
A:[[5],21]
states left:48
Q:[[s(3,12,21),[1],[5]],[s(6,11,17),[2],[3]],[s(4,11,16),[3],[4]],[s(4,7,21),[4],[5]],[s(6,7,18),[5],[2]],
[s(3,11,19),[6],[2]]]
P: [[1],15,8,9]
QM:s(6,12,20)
A:[[3],6]
states left:40
Q:[[s(6,12,20),[1],[3]],[s(6,10,16),[2],[3]],[s(6,12,19),[3],[2]],[s(1,11,17),[4],[3]],[s(3,8,19),[5],[6]
],[s(5,10,15),[6],[1]]]
P: [[1],15,8,9]
QM:s(4,12,18)
A:[[2],18]
states left:30
```

```
Q:[[s(4,12,18),[1],[2]],[s(5,12,21),[2],[5]],[s(5,12,19),[3],[6]],[s(2,11,20),[4],[6]],[s(4,12,16),[5],_
G9304],[s(3,12,15),[6],[1]]]
P: [[1],15,8,9]
QM:s(1,11,14)
A:[[4],11]
states left:15
Q:[[s(1,11,14),[1],[4]],[s(4,8,17),[2],[3]],[s(1,10,15),[3],[4]],[s(4,7,15),[4],[5]],[s(4,12,18),[5],[2]],
[s(3,7,16),[6],[5]]]
P: [[1],15,8,9]
QM:s(1,12,14)
A:[[4],14]
states left:10
Q:[[s(1,12,14),[1],[4]],[s(2,11,14),[2],[4]],[s(1,11,21),[3],[4]],[s(6,9,19),[4],[1]],[s(6,11,17),[5],[3]
],[s(4,8,13),[6],[1]]]
P: [[1],15,8,9]
QM:s(1,12,20)
A:[[4],1]
states left:8
Q:[[s(1,12,20),[1],[4]],[s(6,7,21),[2],[3]],[s(1,7,15),[3],[4]],[s(4,7,18),[4],[5]],[s(6,11,16),[5],[3]],[s
(5,8,14),[6],[1]]]
P: [[1],15,8,9]
QM:s(2,12,20)
A:[[6],2]
states left:6
Q:[[s(2,12,20),[1],[6]],[s(4,11,14),[2],[4]],[s(3,10,18),[3],[6]],[s(6,7,21),[4],[5]],[s(1,10,18),[5],[2]
],[s(5,12,19),[6],[2]]]
P: [[1],15,8,9]
QM:s(5,12,20)
A:[[6],5]
states left:4
Q:[[s(5,12,20),[1],[6]],[s(5,11,15),[2],[4]],[s(5,11,18),[3],[4]],[s(5,7,17),[4],[5]],[s(4,7,17),[5],[3]],
[s(3,10,15),[6],[1]]]
P: [[1],15,8,9]
QM:s(3,12,20)
A:[[6],3]
states left:2
Q:[[s(3,12,20),[1],[6]],[s(2,10,17),[2],[3]],[s(3,12,16),[3],[5]],[s(5,11,20),[4],[6]],[s(1,9,16),[5],[1]
],[s(4,7,13),[6],[3]]]
P: [[1],15,8,9]
QM:s(4,12,20)
true.
```

Appendix B

Query of player(6). Before modal.pl

```
16 ?- players(6).
deal:[[[1],5,13,12],[[2],8,10,4],[[3],17,3,20],[[4],14,18,6],[[5],19,1,9],[[6],16,15,7]]
Win:s(2,11,21)
my cards:[[1],5,13,12]
states left:200
P: [[1],5,13,12]
QM:s(2,9,18)
A1:[4]
s(2,11,21)
[[4],18]
A:[[4],18]
Statesleft OldL: 175
L:[[[1],5,13,12],[[2],8,10,4],[[3],17,3,20],[[4],14,18,6],[[5],19,1,9]]
Q:[[s(2,9,18),[1],[4]],[s(6,12,19),[2],[4]],[s(3,11,15),[3],[6]],[s(3,12,14),[4],[1]],[s(2,12,16),[5],[6]
],[s(2,11,14),[6],[4]]]
states left NewL: 175
P: [[1],5,13,12]
QM:s(3,7,15)
A1:[3]
s(2,11,21)
[[3],3]
A:[[3],3]
Statesleft OldL: 140
L:[[[1],5,13,12],[[2],8,10,4],[[3],17,3,20],[[4],14,18,6],[[5],19,1,9]]
Q:[[s(3,7,15),[1],[3]],[s(3,10,13),[2],[3]],[s(2,8,19),[3],[5]],[s(4,9,17),[4],[5]],[s(5,9,17),[5],[1]],[s(
5,12,14),[6],[1]]]
states left NewL: 140
P: [[1],5,13,12]
QM:s(1,10,15)
A1:[2]
s(2,11,21)
[[2],10]
A:[[2],10]
Statesleft OldL: 112
L:[[[1],5,13,12],[[2],8,10,4],[[3],17,3,20],[[4],14,18,6],[[5],19,1,9]]
Q:[[s(1,10,15),[1],[2]],[s(1,9,20),[2],[3]],[s(6,7,19),[3],[4]],[s(5,10,13),[4],[1]],[s(6,10,14),[5],[2]],
[s(3,8,18),[6],[2]]]
states left NewL: 112
P: [[1],5,13,12]
```

```
QM:s(2,7,15)
A1:[6]
s(2,11,21)
[[6],15]
A:[[6],15]
Statesleft OldL: 96
L:[[[1],5,13,12],[[2],8,10,4],[[3],17,3,20],[[4],14,18,6],[[5],19,1,9]]
Q:[[s(2,7,15),[1],[6]],[s(1,12,20),[2],[3]],[s(3,10,18),[3],[4]],[s(4,11,18),[4],[2]],[s(2,7,17),[5],[6]],
[s(1,10,14),[6],[2]]]
states left NewL: 72
P: [[1],5,13,12]
QM:s(4,11,20)
A1:[2]
s(2,11,21)
[[2],4]
A:[[2],4]
Statesleft OldL: 48
L:[[[1],5,13,12],[[2],8,10,4],[[3],17,3,20],[[4],14,18,6],[[5],19,1,9]]
Q:[[s(4,11,20),[1],[2]],[s(6,8,14),[2],[4]],[s(6,7,20),[3],[4]],[s(3,8,19),[4],[5]],[s(4,11,18),[5],[2]],[s
(1,11,15),[6],[5]]]
states left NewL: 30
P: [[1],5,13,12]
QM:s(2,11,17)
A1:[3]
s(2,11,21)
[[3],17]
A:[[3],17]
Statesleft OldL: 24
L:[[[1],5,13,12],[[2],8,10,4],[[3],17,3,20],[[4],14,18,6],[[5],19,1,9]]
Q:[[s(2,11,17),[1],[3]],[s(1,8,16),[2],[5]],[s(3,8,13),[3],[1]],[s(6,9,18),[4],[5]],[s(3,8,18),[5],[2]],[s(
5,9,15),[6],[1]]]
states left NewL: 6
P: [[1],5,13,12]
QM:s(2,11,14)
A1:[4]
s(2,11,21)
[[4],14]
A:[[4],14]
Statesleft OldL: 4
L:[[[1],5,13,12],[[2],8,10,4],[[3],17,3,20],[[4],14,18,6],[[5],19,1,9]]
Q:[[s(2,11,14),[1],[4]],[s(3,10,13),[2],[3]],[s(5,10,15),[3],[6]],[s(3,12,13),[4],[1]],[s(4,12,18),[5],[
1]],[s(4,11,14),[6],[2]]]
states left NewL: 4
P: [[1],5,13,12]
QM:s(2,11,21)
s(2,11,21)
```

```
A1:[0]
s(2,11,21)
[[0], (2,11,21)]
WE HAVE A WINNNER!!s(2,11,21)
true.
```

Appendix C

cluedo.pl

```
% Author:  Vemund
% Date: 24.03.2015

person('Miss Scarlett').       % 1
person('Professor Plum').      % 2
person('Mrs. Peacock').        % 3
person('Reverend Green').      % 4
person('Colonel Mustard').     % 5
person('Mrs. White').          % 6

weapon('Candlestick').         % 7
weapon('Dagger').              % 8
weapon('Lead Pipe').           % 9
weapon('Revolver').            % 10
weapon('Rope').                % 11
weapon('Wrench').              % 12

location('Kitchen').           % 13
location('Ballroom').          % 14
location('Conservatory').      % 15
location('Dining Room').       % 16
location('Billiard Room').     % 17
location('Library').           % 18
location('Lounge').            % 19
location('Hall').              % 20
location('Study').             % 21

cards(['Miss Scarlett','Professor Plum','Mrs. Peacock','Reverend Green','Colonel
Mustard','Mrs. White','Candlestick', 'Dagger', 'Lead Pipe', 'Revolver','Rope', 'Wrench',
'Kitchen','Ballroom','Conservatory' ,'Dining Room','Billiard
Room','Library','Lounge','Hall','Study']).
cards2([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]).
winstates([s(1,7,13),s(1,7,14),s(1,7,15),s(1,7,16),s(1,7,17),s(1,7,18),s(1,7,19),s(1,7,20),s(1,7
,21),
s(1,8,13),s(1,8,14),s(1,8,15),s(1,8,16),s(1,8,17),s(1,8,18),s(1,8,19),s(1,8,20),s(1,8,21),
s(1,9,13),s(1,9,14),s(1,9,15),s(1,9,16),s(1,9,17),s(1,9,18),s(1,9,19),s(1,9,20),s(1,9,21),
s(1,10,13),s(1,10,14),s(1,10,15),s(1,10,16),s(1,10,17),s(1,10,18),s(1,10,19),s(1,10,20),s(1,10
,21),
s(1,11,13),s(1,11,14),s(1,11,15),s(1,11,16),s(1,11,17),s(1,11,18),s(1,11,19),s(1,11,20),s(1,11
,21),
s(1,12,13),s(1,12,14),s(1,12,15),s(1,12,16),s(1,12,17),s(1,12,18),s(1,12,19),s(1,12,20),s(1,12
,21),

s(2,7,13),s(2,7,14),s(2,7,15),s(2,7,16),s(2,7,17),s(2,7,18),s(2,7,19),s(2,7,20),s(2,7,21),
s(2,8,13),s(2,8,14),s(2,8,15),s(2,8,16),s(2,8,17),s(2,8,18),s(2,8,19),s(2,8,20),s(2,8,21),
s(2,9,13),s(2,9,14),s(2,9,15),s(2,9,16),s(2,9,17),s(2,9,18),s(2,9,19),s(2,9,20),s(2,9,21),
s(2,10,13),s(2,10,14),s(2,10,15),s(2,10,16),s(2,10,17),s(2,10,18),s(2,10,19),s(2,10,20),s(2,10
,21),
s(2,11,13),s(2,11,14),s(2,11,15),s(2,11,16),s(2,11,17),s(2,11,18),s(2,11,19),s(2,11,20),s(2,11
,21),
s(2,12,13),s(2,12,14),s(2,12,15),s(2,12,16),s(2,12,17),s(2,12,18),s(2,12,19),s(2,12,20),s(2,12
,21),

s(3,7,13),s(3,7,14),s(3,7,15),s(3,7,16),s(3,7,17),s(3,7,18),s(3,7,19),s(3,7,20),s(3,7,21),
s(3,8,13),s(3,8,14),s(3,8,15),s(3,8,16),s(3,8,17),s(3,8,18),s(3,8,19),s(3,8,20),s(3,8,21),
s(3,9,13),s(3,9,14),s(3,9,15),s(3,9,16),s(3,9,17),s(3,9,18),s(3,9,19),s(3,9,20),s(3,9,21),
s(3,10,13),s(3,10,14),s(3,10,15),s(3,10,16),s(3,10,17),s(3,10,18),s(3,10,19),s(3,10,20),s(3,10
,21),
s(3,11,13),s(3,11,14),s(3,11,15),s(3,11,16),s(3,11,17),s(3,11,18),s(3,11,19),s(3,11,20),s(3,11
,21),
s(3,12,13),s(3,12,14),s(3,12,15),s(3,12,16),s(3,12,17),s(3,12,18),s(3,12,19),s(3,12,20),s(3,12
,21),

s(4,7,13),s(4,7,14),s(4,7,15),s(4,7,16),s(4,7,17),s(4,7,18),s(4,7,19),s(4,7,20),s(4,7,21),
s(4,8,13),s(4,8,14),s(4,8,15),s(4,8,16),s(4,8,17),s(4,8,18),s(4,8,19),s(4,8,20),s(4,8,21),
s(4,9,13),s(4,9,14),s(4,9,15),s(4,9,16),s(4,9,17),s(4,9,18),s(4,9,19),s(4,9,20),s(4,9,21),
s(4,10,13),s(4,10,14),s(4,10,15),s(4,10,16),s(4,10,17),s(4,10,18),s(4,10,19),s(4,10,20),s(4,10
,21),
s(4,11,13),s(4,11,14),s(4,11,15),s(4,11,16),s(4,11,17),s(4,11,18),s(4,11,19),s(4,11,20),s(4,11
,21),
s(4,12,13),s(4,12,14),s(4,12,15),s(4,12,16),s(4,12,17),s(4,12,18),s(4,12,19),s(4,12,20),s(4,12
,21),

s(5,7,13),s(5,7,14),s(5,7,15),s(5,7,16),s(5,7,17),s(5,7,18),s(5,7,19),s(5,7,20),s(5,7,21),
s(5,8,13),s(5,8,14),s(5,8,15),s(5,8,16),s(5,8,17),s(5,8,18),s(5,8,19),s(5,8,20),s(5,8,21),
s(5,9,13),s(5,9,14),s(5,9,15),s(5,9,16),s(5,9,17),s(5,9,18),s(5,9,19),s(5,9,20),s(5,9,21),
s(5,10,13),s(5,10,14),s(5,10,15),s(5,10,16),s(5,10,17),s(5,10,18),s(5,10,19),s(5,10,20),s(5,10
,21),
s(5,11,13),s(5,11,14),s(5,11,15),s(5,11,16),s(5,11,17),s(5,11,18),s(5,11,19),s(5,11,20),s(5,11
,21),
s(5,12,13),s(5,12,14),s(5,12,15),s(5,12,16),s(5,12,17),s(5,12,18),s(5,12,19),s(5,12,20),s(5,12
,21),

s(6,7,13),s(6,7,14),s(6,7,15),s(6,7,16),s(6,7,17),s(6,7,18),s(6,7,19),s(6,7,20),s(6,7,21),
s(6,8,13),s(6,8,14),s(6,8,15),s(6,8,16),s(6,8,17),s(6,8,18),s(6,8,19),s(6,8,20),s(6,8,21),
s(6,9,13),s(6,9,14),s(6,9,15),s(6,9,16),s(6,9,17),s(6,9,18),s(6,9,19),s(6,9,20),s(6,9,21),
s(6,10,13),s(6,10,14),s(6,10,15),s(6,10,16),s(6,10,17),s(6,10,18),s(6,10,19),s(6,10,20),s(6,10
,21),
s(6,11,13),s(6,11,14),s(6,11,15),s(6,11,16),s(6,11,17),s(6,11,18),s(6,11,19),s(6,11,20),s(6,11
,21),
s(6,12,13),s(6,12,14),s(6,12,15),s(6,12,16),s(6,12,17),s(6,12,18),s(6,12,19),s(6,12,20),s(6,12
,21)]).



%recon([X|T],Y):- dersom X=,s(Y,_,_);s(_,Y,_);s(_,_,Y). plasser X i liste Z der alle Y=Y, send
T videre og gjenta.
recon(X,Y,Z):- writeln(Z).
recon([X|T],Y,Z):- (X=s(Y,_,_);X=s(_,Y,_);X=s(_,_,Y)),recon(T,Y,[X|Z]).
```

Appendix D

dealer.pl

```
% Author:Vemund
% Date: 27.08.2015

test(X):-winstates(Select),random_member(X,Select).

%plukker ut en state som er winstate og gir de tre kortene individuelt
test2(X,Y,Z):-winstates(Select),random_member((s(X,Y,Z)),Select).


%del ut kort til en win state og resten til X antall spillere
%eksempelspørring: dealer(L,6,W).ND
```

```
%dealer(L,X,Win):- winselector(ND,Win),length(ND,C1),C is C1/X, rs(ND,L,C).

dealer(L,X,Win):- winselector(ND,Win),dealer2(ND,X,L).

%Lager liste med X tomme lister, legger random element fra C inn i tomme lister en etter en.

%dealer2([],X,L).
dealer2(C,X,L):- list(X,[],L1), deal(C,L1,[],L2), rev2(L2,[],L).

list(0,L,L1):- L1 = L, !.
list(X,L,L0):- X1 is X-1, add([[X]],L,L1), list(X1,L1,L0).

deal([],[],L,L2):-  L2=L,!.
deal([],L1,L,L2):-rev(L1,R), append(R,L,L2),!.
deal(C,[],L,L2):- rev(L,R),deal(C,R,[],L2).
deal(C,[L0|L1],F,L4):- random_select(Y,C,R),add(Y,L0,L2), add(L2,F,F1),deal(R,L1,F1,L4).

%list(0,L).
%list(X,L):- X1 is X-1, add([],L,L1), list(X1,L1).

add(X, L, [X|L]).

rev2([],L1,L) :- L = L1.
rev2([H|T],L1,L) :- rev(H,H1), add(H1,L1,L2), rev2(T,L2,L).

rev(L,R):- accRev(L,[],R).
accRev([H|T],A,R):-  accRev(T,[H|A],R).
accRev([],A,A).

%får inn 3 kort fra test og trekker de fra resten av kortene, slik at de resetrende kortene som
skal deles ut er igjen.
%legger også wistaten tilbake som en state(unødvendig?).
winselector(ND,Win):- cards2(OD), test2(X,Y,Z),
subtract(OD,[X,Y,Z],ND),s(X,Y,Z)=Win.


mMaker(X,M):-list(X,L,L0),cDeal(L0,[],M).

cDeal([],M,R):- rev(M,R),!.
cDeal([H|T],M,M1):- cards2(C),append(H,[C],H1), add(H1,M,M2), cDeal(T,M2,M1).
```

Appendix E

query.pl

```
% Author: Vemund
% Date: 08.09.2015


%C = listen med kort spillerene har, Q=spørringen som er en state,A = svaret en får mao
nye kunnskapen
%C er nødt til å være i rett rekkefølge og skal ikke inneholde kortene til spørreren.
%Q blir valg av ett annet predikat som velger hvilken state man skal spørre etter.
%eksempelspørring choice([[1,4,18],[6,19,3],[2,20,21],[17,5,11],[12,13,15]],s(5,7,14),A).
%choice([[[6],1, 15, 14], [[5],20, 6, 19], [[4],2, 4, 5], [[3],18, 9, 16], [[2],7, 17, 10 ], [[1],8, 21,
12]],s(5,6,20),A).

%choice(C,Q,A):-  x -> y;z.
%choice([],S,A):-writeln(S),true,!.
choice([H|T],s(X,Y,Z),[A2,A1],W):-
H=[H1|T1],A2=H1,choice1(T1,s(X,Y,Z),[A2,A1],W),!;once(choice(T,s(X,Y,Z),[A2,A1],W)).
choice([],s(X,Y,Z),[A2,A1],W):- W=([Win,T1,T2]),
writeln(Win),s(X,Y,Z)==Win,A2=[7],A1=(X,Y,Z);A2=[1],A1=(X,Y,Z).

choice1([H|T],s(X,Y,Z),[A2,A1],W):- H==X -> A1=X;H==Y -> A1=Y;H==Z ->
A1=Z;choice1(T,s(X,Y,Z),[A2,A1],W).


win(W,[(X,Y,Z)],U,U2):- write('WE HAVE A WINNNER!!'), writeln(W),W==s(X,Y,Z),U=U2;fail.
```

Appendix F

subtracter.pl

```
% Author: Vemund
% Date: 04.09.2015

%ws1([s(1,7,13),s(1,7,14),s(1,7,15),s(1,7,16),s(1,7,17),s(1,7,18),s(1,7,19),s(1,7,20),s(1,7,21)
,s(2,8,13),s(2,8,14),s(2,8,15),s(2,8,16),s(2,8,17),s(2,8,18),s(2,8,19),s(2,8,20),s(2,8,21),s(3,9,
13),s(3,9,14),s(3,9,15),s(3,9,16),s(3,9,17),s(3,9,18),s(3,9,19),s(3,9,20),s(3,9,21),s(4,10,13),s
(4,10,14),s(4,10,15),s(4,10,16),s(4,10,17),s(4,10,18),s(4,10,19),s(4,10,20),s(4,10,21),s(5,11,
13),s(5,11,14),s(5,11,15),s(5,11,16),s(5,11,17),s(5,11,18),s(5,11,19),s(5,11,20),s(5,11,21),s(
6,12,13),s(6,12,14),s(6,12,15),s(6,12,16),s(6,12,17),s(6,12,18),s(6,12,19),s(6,12,20),s(6,12,2
1)]).
%ws2([s(1,8,13),s(1,8,14),s(1,8,15),s(1,8,16),s(2,7,13),s(2,7,14),s(2,7,15),s(2,7,16)]).

%skal få inn liste med utdelte kort, trekker tra alle states som har kortet i seg.
%eksempelspørring: run(L,[[1],1,5,19]).

%test(L):-dealer([D|R],6,W),writeln(D),run(L,D).

run(L,N):- winstates(WS),run2(L,N,WS),length(L,N1),write('states left:'), writeln(N1).

run2(L1,[H|T],WS):-run1(L1,T,WS).

run1(L,[],WS):- L=WS.
run1(L1,[H|T],WS):-subs(H,WS,WS,L), run1(L1,T,L).

%trekker fra alle states som inneholder kort N
%spørring:
subs(7,[s(1,8,13),s(1,8,14),s(1,8,15),s(1,8,16),s(2,7,13),s(2,7,14),s(2,7,15),s(2,7,16)],[s(1,8,1
3),s(1,8,14),s(1,8,15),s(1,8,16),s(2,7,13),s(2,7,14),s(2,7,15),s(2,7,16)],L).
subs2(_,[],L1,L1).
subs2(N,[H|T],L1,L3):- subs(N,T,L1,L3).
%subs(N,L,L1):- L=[H|T],not(H=s(N,_,_);H=s(_,N,_);H=s(_,_,N))-> writeln(L1).
subs(N,L,L1,L3) :-L=[H|T],(H=s(N,_,_);H=s(_,N,_);H=s(_,_,N)) ->
subtract(L1,[H],L2),subs(N,T,L2,L3);subs2(N,L,L1,L3).
```

Appendix G

unity.pl

```
% Author: Vemund
% Date: 26.11.2015

%deler ut kort og sender spiller 1 videre til trekkfra

counter(X,0,N,R,R2):- R2 is R/N,!.
counter(X,N,N2,R,R2):-N3 is N-1,write('NNNNNNNNNNNNNNNNNNNN: '), writeln(N3),
players(6,U), R3 is R + U,write('RRRRRRRRRRRRRRRRRRRRRRRRRR: '), writeln(R3),
counter(X,N3,N2,R3,R2).
```

```
players(X,U2):-
dealer(L,X,Win),write('deal:'),writeln(L),mMaker(X,M),write('Win:'),writeln(Win),L =
[P|T],P=[H|T2],remover2(H,T2,M,M2,0),W=[Win,M2,[]],winstates(WS),trekkfra(P,WS,T,W,U2
).

%trekker fra utdelte kort(P) og sender videre til spørring
trekkfra(P,WS,T,W,U2):- write('my cards:'), writeln(P),run2(L,P,WS),length(L,N),write('states
left:'), writeln(N), sporring(L,T,P,W,0,U2),write('U: '), writeln(U).

trekkfra2([A1|A2],WS,T,P,W,Q,U,U2):- W=[X,M,Z],  writeln(X),
writeln([A1|A2]),A1==[7],win(X,A2,U,U2),!;write('A:'),
writeln([A1|A2]),run2(L,[A1|A2],WS),W=[X,M,Z],remover2(A1,A2,M,M2,0),W2=[X,M2,Z],sporr
ing2(P,T,L,W2,Q,U,U2).

%trekkfra2(A,WS,T,P):- writeln(A),run1(L,A,WS),length(L,N), writeln(N),sporring(L,T,P).


sporring(L,T,P,W,U,U3):-U2 is U+1,random_select(Q,L,R), write('P: '), writeln(P),write('QM:'),
writeln(Q),choice(T,Q,[A1,A2],W),write('A1:'),
writeln(A1),trekkfra2([A1,A2],L,T,P,W,[[Q,[1],A1]],U2,U3).

%leger P i en ny liste, slik at T blir mindre, når den er tom vil runden være over.
%legger sammen L1 og T2 slik at choice kan brukes. Q er det interesanne her tar med
videre gjennom runden.
%A sin relevans for andre spillere vil jeg lege til senere, andre spillere lærer ingenting atm.
%legg til choice(T3,Q,A)

sporring2(P,T,L,W,Q1,U,U2):- add(P,[],L1), T=[T1|T2], append(T2,L1,T3), selecter(Q,T1,W),
choice(T3,Q,[A1,A2],W),T1=[S|C],Q2=[[Q,S,A1]], append(Q2,Q1,Q3),
sporring2(T1,T2,L1,Q3,L,W,U,U2).

sporring2(T1,[],L1,Q,L,W,U,U2):-append(L1,[T1],L2),L2=[P|T],rev(Q,R),length(L,N1),write('St
atesleft OldL: '), writeln(N1),write('L:'), writeln(L1),write('Q:'),
writeln(R),analyzer(R,W,L,NewL,W1),length(NewL,N),write('states left NewL: '), writeln(N),
sporring(NewL,T,P,W1,U,U2).
%sporring2(T1,[],L1,Q,L,W,U,U2):-append(L1,[T1],L2),L2=[P|T],rev(Q,R),write('Q:'),
writeln(R),write('L:'), writeln(L1), sporring(L,T,P,W,U,U2).
sporring2(T1,T,L1,Q,L,W,U,U2):-append(L1,[T1],L2), T=[T2|T3],
append(T3,L2,T4),selecter(Q1,T2,W), choice(T4,Q1,[A1,A2],W),T2=[S|C], A =[Q1,S,A1],
add(A,Q,Q2),sporring2(T2,T3,L2,Q2,L,W,U,U2).



selecter(Q,T1,[W,X,Y]):-W=s(A,B,C), winstates(WS),run2(L2,[A,B,C],WS),
random_select(Q,L2,L).
%selecter(Q,T1):- Q= s(1,2,3).
```

```
%analyzer(Q,ML,L,L2,OQ,M).

 % analyzer(Q,W,L,L2,W2)
```

Appendix H

modal.pl

```
% Author: Vemund
% Date: 04.02.2016

 %Q: [[s(5,9,21),[1]],[s(6,8,20),[4]],[s(4,12,17),[1]],[s(3,12,14),[1]],[s(5,9,17),[1]]]

 %win:s(6,9,17)

 %deal:[[[1],19,3,16],[[2],21,2,7],[[3],14,13,8],[[4],12,1,6],[[5],4,11,20],[[6],10,18,15]]

 %L: [[[1],[5],[9],[21]], [[2]], [[3]], [[4]], [[5]], [[6]]]

%write('L1: '), writeln(L1),

%spørring:
modal([[s(4,7,16),[1],[5]],[s(5,11,16),[2],[4]],[s(6,12,21),[3],[5]],[s(6,8,14),[4],[1]],[s(1,11,17),[5]
,[3]],[s(4,12,19),[6],[2]]],M,M1,[[[1]], [[2]], [[3]], [[4]], [[5]],
[[6]]],[[[1],[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]],[[2],[1, 2, 3, 4, 5, 7, 8, 9,
10, 12, 13, 14, 15, 17, 18, 19, 20, 21]], [[3], [1, 3, 4, 5, 6, 8, 9, 10, 11, 12, 14, 15, 16, 17, 18,
19, 20, 21]], [[4], [1, 3, 4, 5, 6, 8, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19, 20, 21]], [[5], [1, 2, 3, 4,
5, 7, 8, 9, 10, 12, 13, 14, 15, 17, 18, 19, 20, 21]], [[6], [1, 3, 4, 5, 7, 8, 9, 10, 12, 13, 14, 15,
17, 18, 19, 21]]]).

% modal([],L,L):-!.
% modal(Q,M,L) :- Q= [[s(X,Y,Z),P]|R],  L=[L1|L2],L1=[N|B],write('L1: '), writeln(L1), [P]==L1
-> leggetil(L1,L3,Q),add(L3,L2,L4),write('L4 '),writeln(L4),modal(R,M,L4);
% Q= [[s(X,Y,Z),P]|R],  L=[L1|L2],L1=[N|B],write('NB: '), writeln(B),write('NN: '),
writeln(N),write('NP: '), writeln(P), P==N ->
leggetil2(N,B,L3,[[X],[Y],[Z]]),add(L3,L2,L4),write('NL4 '),writeln(L4),modal(R,M,L4);
% write('haha_L: '),writeln(L),L=[L1|L2],first_last([L1],L2,NL),write('NL: '),
writeln(NL),modal(Q,M,NL).

modal(Q,M1,M2,L,L1):- showcard(Q,M,L),showcard3(Q,M1,M), nocard(Q,M2,L1),
lalala(M1,M2).



showcard3(Q,M1,M):- showcard2(Q,M1,M).

modal2(Q,M1,M,L):-  showcard(Q,M,L),showcard2(Q,M1,M).

showcard([],L,L):-!.
showcard(Q,M,L) :- Q= [[s(X,Y,Z),P1,P]|R],  L=[L1|L2],L1=[N|B], [P]==L1 ->
leggetil(L1,L3,Q),add(L3,L2,L4),showcard(R,M,L4);
 Q= [[s(X,Y,Z),P1,P]|R],  L=[L1|L2],L1=[N|B], P==N ->
leggetil2(N,B,L3,[[X],[Y],[Z]]),add(L3,L2,L4),showcard(R,M,L4);L=[L1|L2],
first_last([L1],L2,NL),showcard(Q,M,NL).

showcard2([],L,L):-!.
showcard2(Q,M,L) :- Q= [[s(X,Y,Z),P1,P]|R],  L=[L1|L2],L1=[N|B], [P1]==L1 ->
leggetil(L1,L3,Q),first_last([L3],L2,L4),showcard2(R,M,L4);
 Q= [[s(X,Y,Z),P1,P]|R],  L=[L1|L2],L1=[N|B], P1==N ->
leggetil2(N,B,L3,[[X],[Y],[Z]]),add(L3,L2,L4),showcard2(R,M,L4);L=[L1|L2],
first_last([L1],L2,NL),showcard2(Q,M,NL).




nocard([],M,M):-!.
nocard(Q,M,L):-Q= [[s(X,Y,Z),P1,P]|R],P==[0],!;Q= [[s(X,Y,Z),P1,P]|R], P==[7],!; Q=
[[s(X,Y,Z),P1,P]|R], L= [L1|LR],L1=[H|T], P1==H ->first_last([L1],LR,L2),
nocard2([X,Y,Z],L2,P,[[]],M1),nocard(R,M,M1);  L= [L1|LR], first_last([L1],LR,L2),
nocard(Q,M,L2).

%nocard2([],L,P,NL,M):- write('L: '),writeln(L),NL=[H|T],write('NL:
'),writeln(T),append(T,L,M),write('M: '),writeln(M).
nocard2([],L,P,NL,M):- NL=[H|T],append(T,L,M).
%nocard2(C,[],P,NL,M):-  write('HEYOOO: '),writeln(NL),NL=[H|T],nocard2(C,T,P,[[]],M).
nocard2([X,Y,Z],LR,P,NL,M):- LR=[N|R], N=[I|C], I==P ->nocard2([],LR,P,NL,M);  LR=[N|R],
N=[I,C], subtract(C,[X,Y,Z],C2),D=[[I,C2]], first_last(D,NL,R2), nocard2([X,Y,Z],R,P,R2,M).

%gå gjennom alle lister i B og lage en ny versjon med X, Y og Z i hver av dem. Mao dersom
det er tre lister fra før vil det nå bli 9 lister.

% leggetil(L,L2,Q):- Q= [[s(X,Y,Z),P]|R], L=[N|B], B==[] -> L2=[N,[X],[Y],[Z]],write('L2: '),
writeln(L2).
 leggetil(L,L2,Q):- Q= [[s(X,Y,Z),P1,P]|R], L=[N|B], B==[] -> L2=[N,[X],[Y],[Z]].

% Q= [[s(X,Y,Z),P]|R], L=[N|B],leggetil2(X,B,[],Bx),write('Bx: '),
writeln(Bx),leggetil2(Y,B,[],By),write('By: '), writeln(By),leggetil2(Z,B,[],Bz),write('Bz: '),
writeln(Bz).



% leggetil2(N1,NR,L,Q):- write('LT_Q: '), writeln(Q),write('LT_NR: '),
writeln(NR),semi_perm(NR,Q,[],L1,NR),write('LT_L1: '), writeln(L1),add(N1,L1,L).
 leggetil2(N1,NR,L,Q):- semi_perm(NR,Q,[],L1,NR),add(N1,L1,L).




 %legger e element inn på sisteplass i listen
 first_last(L,L2,L3) :- rev(L2,NL2),NL2=[H|T], add(H,L,Ll) , first_last2(Ll,T,L3).
 first_last2(L,[],L).
 first_last2(L,L2,L3) :- L2=[H|T], add(H,L,Ll) , first_last2(Ll,T,L3).
 %setter sammen nye kombinasjoner av nåværende liste L, og de nye kortene som skal inn
N, til en ny liste FL, L og BL er den samme BL str for backuplist.
 %spørring: semi_perm([[1],[2],[3]],[[4],[5],[6]],[],FL,[[1],[2],[3]]).

 semi_perm(L,[],L2,L2,BL).
 semi_perm([],N,L2,FL,BL):- N =[N1|NR], semi_perm(BL,NR,L2,FL,BL).
 semi_perm(L,N,L2,FL,BL) :-L=[H|T], N=[N1|NR], first_last(N1,H,H2), add(H2,L2,L3),
semi_perm(T,N,L3,FL,BL).



 analyzer(Q,W,L,L2,W2):- W=[Win,ML,E], showcard(Q,M,[[[1]], [[2]], [[3]], [[4]], [[5]], [[6]]]),
nocard(Q,M2,ML),add(M,E,E2), anSC(E2,M2,M2,[],N),
remover(N,M2,NM2,L,L2),W2=[Win,NM2,E2],!.

 anSC(M,[],M2,NewM,N):- anSC(M,M2,M2,NewM,N).
 anSC([],_,M2,M,R):- rev(M,R),!.
 anSC([[]|T],_,M2,M,N):- anSC(T,M2,M2,M,N).
 anSC([M|Rm],[M2|Rm2],M4,NewM,N):- M=[H1|T1],H1=[H|T], M2=[H2,T2], H==H2 ->
assemble(T,T2,[],L1),add([H|L1],NewM,NewM2),
anSC([T1|Rm],Rm2,M4,NewM2,N);M=[H1|T1], first_last([H1],T1,NM),
anSC([NM|Rm],[M2|Rm2],M4,NewM,N).


 remover([],M2,M2,L,L).
 remover([H|T],M2,NM2,L,L2) :-  length(H,Num), Num==2 ->
H=[H1,T1],remover2(H1,T1,M2,M3,0),add(H1,T1,T3),run2(L3,T3,L),
remover(T,M3,NM2,L3,L2); remover(T,M2,NM2,L,L2).

 remover2(_,_,M,M,5):- !.
 remover2(H1,T1,[M|MR],M4,N):-N1 is N+1,  M=[H,T], H1=\=H -> subtract(T,T1,T3),
first_last([[H,T3]],MR,M3),remover2(H1,T1,M3,M4,N1);first_last([M],MR,MR2),remover2(H1,
T1,MR2,M4,N).

 assemble([],T2,L,L).
 assemble([H|T],T2,L,L1):- length(H,N),intersection(H,T2,H2),length(H2,N2), N==N2 ->
add(H2,L,M), assemble(T,T2,M,L1);assemble(T,T2,L,L1).


  %anSC2(Q,M1,M2,NewM1,N2).
 M==[],anSC([Rm],[M2|Rm2],M4,NewM,N);.


 anSC2([Q1|QR],[],_,NewM1,N).
 anSC2([Q1|QR],[M1|Rm],[M2|Rm2],NewM1,N):- Q1= [s(X,Y,Z),P1,P2],  M1=[H|T],
M2=[H2,T2], H==P2, H2==P1 -> assemble(T,T2,[],L1), add([H|L1],NewM1,NewM2),
anSC2(QR,Rm,Rm2,NewM2,N);
first_last(M1,Rm,NM),anSC2([Q1|QR],NM,[M2|Rm2],NewM1,N).


 showcard2(Q,M1,[[[1]], [[2]], [[3]], [[4]], [[5]], [[6]]]),
```