Imperial College London

General Card Game Playing

Author:
Mark Law

Supervisor:
Graham Deane

June 20, 2013

Abstract

Determining good heuristics for an AI player in a given card game is usually fairly easy, provided there is no betting involved. However, building a system which is able to find these heuristics from the game rules alone is far more interesting. Over the course of this project we identified a general subset of card games, found a simple way of representing them and designed and implemented a method for analysing this representation to lead to the creation of an AI player.

We used of Answer Set Programming to implement a technique called hyper play[1] which enabled us to approximate the values of particular moves by considering possible models of the games.

We developed a complete system to allow input of games in the form of a flowchart along with fully functional playable versions of the games.

Acknowledgements

Firstly I'd like to thank my supervisor, Graham Deane, for introducing me to Answer Set Programming. He was always available when I needed advice and I have thoroughly enjoyed learning about ASP from him. My second marker Dr Krysia Broda has also been a great source of advice for me this year.

I would also like to acknowledge Dr Marek Sergot and again Dr Krysia Broda, as their courses on Knowledge Representation and Automated Reasoning helped to further my understanding of the concepts in section 2.5.3. Some of the definitions in this section are influenced by the definitions in these courses.

I'd also like to thank everyone who was kind enough to proof read this report for me... I apologise for the length!

Contents

1 Introduction

1.1 Project Aims

In the last decade there has been a large amount of research into General Game Playing and even an annual competition to produce the best General Game Player[2]. The interest in this field is largely due to the belief that, just as game playing has its applications to the real world, General Game Playing will have many more applications to real world AI's.

Most current General Game Players focus only on *perfect information* games (games where each player has a complete knowledge of the current game state). One example of such a game is Chess. While this is interesting its applications are rather limited as in the real world an AI will rarely have perfect information.

For this reason *imperfect information* games are of huge interest as their applications are much more widespread. One example of an imperfect information game would be the simple card game *trumps* where, at the beginning of the game, each player has no information about another player's hand.

Our aim for this project was to produce an application where a user can input the rules to a general subset of card games. This application should then generate the game and also, more interestingly, reason about the best strategy/tactics in order to generate an AI player based solely on the game rules. The game rules must specify only what is a legal *move* and how the game is scored; they are not to contain any information about tactics.

We chose to focus on a subset of card games rather than all general games. This is because the goal was not only to produce the AI but to build a playable version of the game, complete with a Graphical User Interface. While it is possible to build a fairly general card game GUI, it be would very difficult to build a general game GUI which represents every different type of game clearly. However, we believe that the techniques we have used for the AI part of the project should extend to general games.

It was not realistic within the course of this project to develop a solution which covers all card games (some card games e.g. poker are currently an area of research in game playing[3]). Instead we chose to focus on carefully defined subsets of card games e.g. trick based card games.

1.2 Approach

We decided early in the project that the main application should be web based as it allows for multi-player games without much difficulty and also makes the application easily accessible to most users.

Most current techniques for General Game Playing involve finding *heuristics* for evaluating the value of the game state after making a possible next move. A popular method for finding these heuristics is to use machine learning[4]. However, we decided to avoid this as it can often be very difficult to understand the reasoning of the player's decisions at game time.

The problem can be divided into two stages: *pre-game* and *game-time*. The pre-game stage will occur only once, when the rules are input by the game creator. The game-time stage takes place every time a decision is to be made about which move to take.

The goal of the pre-game stage is to identify (pseudo) heuristics. The idea is that at *game-time* these are used to evaluate each of the possible game states.

As the games we consider are *imperfect information* games there are many possibilities for the current game state and it would be very inefficient to consider them all. We have used a technique called *hyper play*[1]. The idea here is that we randomly sample the set of possible game states in order to approximate the expected value of making a particular move.

As for representing the games we had two good options. Use a readily available *Game Description Language* (or design our own), or instead represent the games as flow charts. We felt that flow charts were much easier for the average user to input and also represents the state based structure of our games very well. One reason for not wanting to use an already available Game Description Language was that they are designed to represent general games. By designing it ourselves it was easy to add keywords like playCard, shuffle, deal etc which simplifies the implementation of a card game for the user.

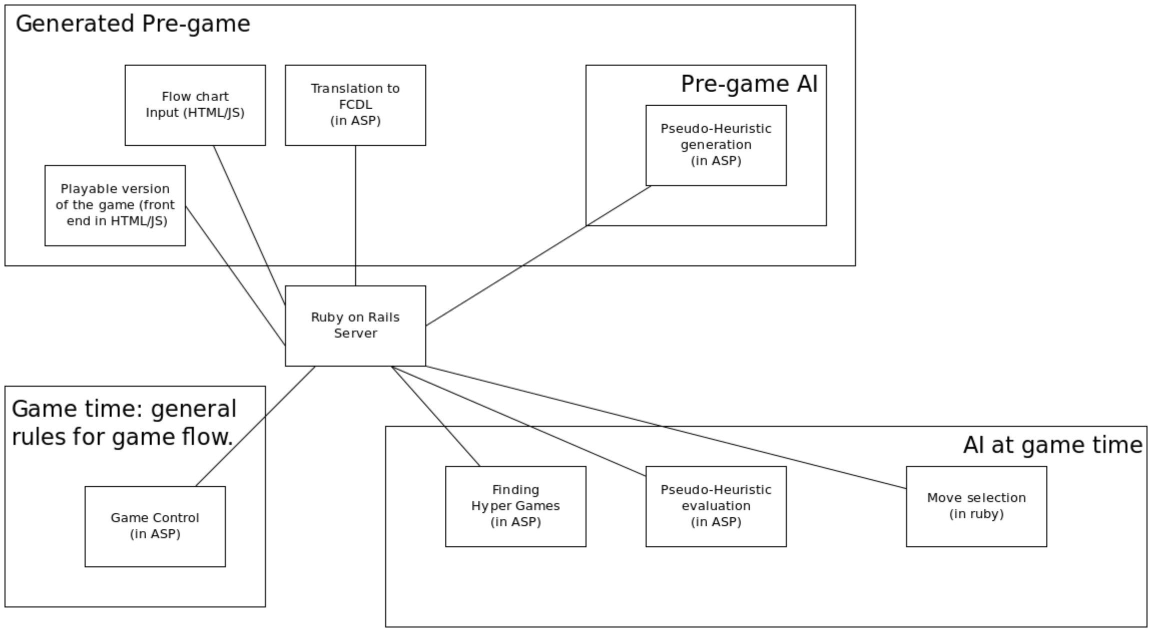

Generated Pre-game
Flow chart Input (HTML/JS)
Translation to FCDL (in ASP)
Pre-game AI
Pseudo-Heuristic generation (in ASP)
Playable version of the game (front end in HTML/JS)
Ruby on Rails Server
Game time: general rules for game flow.
Game Control (in ASP)
AI at game time
Finding Hyper Games (in ASP)
Pseudo-Heuristic evaluation (in ASP)
Move selection (in ruby)

Figure 1.1: An overview of the major components of the system.

1.3 Accomplishments

The major accomplishments in this project were:

1. A language for representing the structure and rules of general card games which can be easily translated into a flow chart but is also easy to analyse when performing (pseudo) heuristic generation.
2. A method for (pseudo) heuristic generation for a subset of general card games.
3. A working implementation of the heuristic generation using Answer Set Programming for all games considered.
4. A working implementation of the hyper play technique again utilising ASP for trick-based games.
5. A working Ruby on Rails server which can be used both to create games and also to play games either against another player (online), or against an AI player which has been generated by the system.

1.4 Report Structure

Chapter 2: This section describes the preliminaries necessary to thoroughly understand the rest of the report. Much of the Answer Set Programming section can be skipped if the reader is only interested in what has been achieved by this project; however, it is necessary for a full understanding of the reasoning behind the decisions made in section 4.

Chapter 3: In this section we introduce much of the theory behind how our solution works. It is very high level and covers with examples the ideas that led to the final solution.

Chapter 4: Here we cover the details of exactly how we translated the ideas in section 3 into a working solution in ASP. We go into great depth on how hyper play is achieved in ASP and how we have generated our pseudo-heuristics and how at game-time we evaluate these pseudo-heuristics. We also describe how we have combined the Ruby side of the solution with ASP. Ruby is used to maintain the game state and control the ASP side of the project. The details of the web application are also contained at the end of this section but can be skipped if the reader is only interested in the AI side of the project.

Chapter 5: The evaluation contains the results of runs of the solution on our games. It also compares our solution with related work.

Chapter 6: The conclusion analyses what we have actually achieved and explores what the next steps for this project could be.

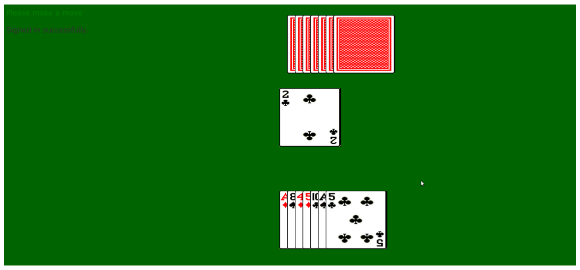
2 Background

2.1 Game Theory

Definition 2.1. A game is said to be a game of *perfect information* if every player has enough information to compute all possible games from the current game state (all combinations of legal moves leading to the end of the game).

(A game without perfect information is said to have *imperfect information*)

Example 2.2. The game of Chess is of perfect information whereas the game of trumps is not as no player can see which cards are in another player's hand and therefore cannot compute which moves will be legal by the next player.


Please make a move

Signed in successfully.

Figure 2.1: A screen shot from a card game (being played with our final system). The opponent's hand is hidden from us.

Remark. Some literature calls perfect information *complete information.*

Definition 2.3. A game is said to be *deterministic* if there is no element of chance involved in the game.

(A game is *non-deterministic* if there is an element of chance)

Example 2.4. Chess is a deterministic game whereas backgammon is non-deterministic.

2.2 General Game Playing with Perfect Information

Definition 2.5. A Game Description Language is a language which can represent the rules of a general game.

Table 1: Some game classifications.

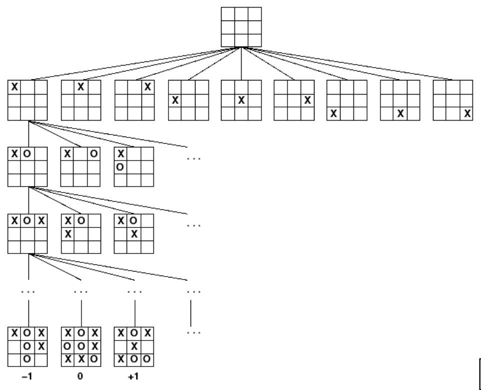
	Deterministic	Non-deterministic
Perfect Information	Chess	Backgammon, Monopoly
Imperfect Information	Battleship	Klondike (Solitaire), Trumps, Poker, Bridge, (Most other card games)

The most commonly used is GDL (outlined in [2]). General Game Players submitted to the tournament described in the article are expected to play games of this form. GDL only covers games of perfect information.

2.2.1 Heuristic Evaluation

Definition 2.6. The *game tree* of a game is a directed graph in which each of the nodes are possible game states and each of the edges is a legal move from its parent state whose child is the resultant state after the move.

Example 2.7. A partial game tree showing the first few moves in a game of noughts and crosses.



[5]

In games like noughts and crosses where the entire game tree is relatively small (and can be reduced further by not considering rotations of games as separate states) we can easily calculate things like the likelihood of a player winning/losing/drawing from each state. Therefore it is easy to see that we can also decide what the optimal moves in each state are.

However, in more complicated games with a larger game tree, it becomes too expensive to conduct an exhaustive search. An alternative is to compute a value for non-terminal states. This value is called a *heuristic*. An example heuristic assigns a value approximately proportional to the probability of the player winning from that state. Using this heuristic it is possible for the player to approximate the optimal decision by selecting the state with the highest value (chance of winning).

When writing an AI for a given game these heuristics are hard coded but clearly, as one heuristic will not fit all games, for General Game Playing they must be calculated at run time for each game. This process is called *heuristic evaluation.*

Many approaches use machine learning[4][6][7] to find a heuristic. We chose to avoid this, mainly because we believe that it is often unclear why a heuristic values one game state more than another. Another reason is that it requires a training phase where many games must be played and we preferred a solution based purely on reasoning about the game rules rather than test data.

Most heuristic generation approaches, such as the one described in [7], are usually very similar (at least at their core) to the one described in [4]. The approach is first to identify key features of the game. These are not necessarily full evaluations of the game state but are functions from the game state to a number.

Definition 2.8. A *feature* of a game G is a function from any possible game state of G to the real numbers that has some relevance to assessing the game state.

Example 2.9. In a two player version of the simple card game trumps (see example 2.14) a good feature would be the difference between the scores of the two players.

Note: this is not enough to evaluate the game state on its own (even though a player is 1 trick ahead he/she could have such a bad hand remaining that he/she has no realistic hope of taking another trick in the game) but is very useful when used in conjunction with other features to evaluate the game state.

They then identify a set of candidate expressions made up of these features which should lead to a heuristic. Their method for finding these candidate expressions is simply to analyse the game description. Although this may seem naive at first they argue that if a game is succinctly described then it must be defined in terms of its most salient features.

From there it is easy to see how a heuristic can be fully defined (giving weights to each of the features).

As our aims were slightly less general (being restricted to card games) we have a head start when it comes to identifying features. This is not to say that the features will be the same every time, but they are likely to be based around similar things each time. For example, most games will have several features based on the player's hand. Therefore we adopted the authors approach of analysing the game definition but we searched for paths through the main part of the game. We then find the conditions on the cards that could legally be played to follow each path along with the value to our player of taking the path.

2.2.2 Applications and Limitations of Perfect Information Games

Game Playing has been viewed as important to AI as the techniques which arise from it are applicable in other areas. For example, the techniques employed to play a game of Chess successfully. However, these usually have very specific use cases meaning that whenever something new is needed the algorithm must be tailored to the job in hand.

For this reason General Game Playing is seen as important. If we can find an algorithm which adapts to the task at hand then surely this can be applied in other areas and as it is much more general, one solution should cover a much wider collection of tasks.

As most of the current GGP solutions are based on GDL rather than GDL II (described in section 2.3.1), in their current form, they are often not suitable for problems with imperfect information.

It is envisaged that GGP will be useful in finance for example in trading stocks. Trading stocks could be translated into a game where the object is to make as much money as possible. However, in most financial applications there is some degree of missing information. Therefore if we want General Game Playing to truly be general enough then we must tackle the problem of imperfect information games.

2.3 General Game Playing with Imperfect Information

Any card game where players have *hands* which are unseen by the other players are imperfect information games. In fact most card games involve some degree of imperfect information. Therefore this project requires General Game Playing with imperfect information.

2.3.1 GDL II

In [8] GDL has been extended to GDL II which considers games of imperfect information. GDL II has two new keywords, *sees* and *random.* Sees is used to show what information each player has and random is used to generate the next game state when this is not deterministic (e.g. dealing a card).

2.3.2 Hyper-play

The paper on *hyper play* [1] puts forward the idea of converting an imperfect information game to a perfect information game. The way it does this is to sample the state space of the possible imperfect information games finding a collection of models of the game which do not contradict what has gone on so far.

Definition 2.10. A *hyper game* is a complete set of values for the unknown variables in the imperfect information game up to the current game state. Such that together with the game rules:

1. they form a perfect information game.
2. they do not contradict what has happened so far in the game, for example: they do not imply that a move which has already taken place was illegal.

The method then plays the next round in each of these as if it were a perfect information game and calculates the expected payoff (how much the player expects to gain by making a move) for each of the legal moves the player has available.

Once the expected payoff has been calculated (relative to each perfect information game) the idea is to then use this to approximate the expected payoff of the imperfect information game.

Take HG to be the set of hyper games. The authors approximate each hyper game as being proportional to the size of that games decision tree (they do not consider the possibility of modelling the probability that a good player would have made such a decision). Therefore if we take C_{hg} to be the product of the number of choices at each node in game $hg \in HG$ then their approximation for the probability of a particular game is:

$P(hg) = \frac{1/C_{hg}}{\sum_{g \in HG}(1/C_g)}$

Definition 2.11. The expected payoff of a move m in a perfect information game g is written as $ExpectedPayoff_g(m)$.

They then approximate the expected payoff of move m in the imperfect information game as:

$ExpectedPayoff(m) = \sum_{hg \in HG}(ExpectedPayoff_{hg}(m) \times P(hg))$.

However, the paper suggests that often a large number of hyper games are needed at each stage. This could have meant that the games were slow to play which would have been unacceptable for the application, however, we found (see section 5.1) that for our card games we could get an acceptable performance from around 20 hyper games.

We used Answer Set Programming, described in section 2.5, to find the sets of hyper games. ASP was ideal for this as each Answer Set corresponded to one hyper-game and it can compute any number of random Answer Sets. It could be designed to test for consistency of the various game models.

2.3.3 Applications

As stated in section 2.2.2 most of the proposed applications for General Game Playing will require the ability to cope with imperfect information.

2.4 Card Games

As highlighted in the introduction, covering all types of general card game is beyond the scope of this project. We therefore decided on an initial subset of card games to focus on. All games considered here are two player games.

Basic Card Game Terminology

The *rank* of a card can be 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King or Ace.

The *suit* of a card can be clubs, diamonds, hearts or spades.

A *pack* of cards has one of each combination of the possible ranks and suits (We will not use the jokers in any of the games described in this project).

A player's *hand* is the cards which he/she "owns". Unless stated otherwise only he/she can see them.

When the pack is *shuffled* the cards in the pack are put into a random order.

When cards are *dealt* each player is given (usually the same number of) cards for their hand.

If a suit is said to be *trumps* then it will beat a card of any other suit (regardless of rank).

2.4.1 Trick Based Games

One of the simplest types of card games are *trick based games*.

Definition 2.12. A *trick* is played as follows:

1. The *trick leader* (the player who plays first in the trick) plays a card from his hand, sometimes with restrictions on what they can play. Every player can see this card.
2. The players then take turns (usually in clockwise order) to play a card, sometimes with restrictions on what they can play, until each player has played exactly one card in the current trick.
3. Based on the cards which have been played by each player the trick winner is decided according to the rules of the game. The player's scores may go up or down based on what has been played.

Definition 2.13. A *trick based* card game is one that takes the following form:

1. The pack is shuffled.
2. The same number of cards are dealt to each player.
3. The initial trick leader is decided (usually at random).
4. If some other termination condition has not been met then while the players still have cards in their hand, tricks (defined below) are played. The winner of one trick is the trick leader of the next.
5. The winner of the game is the one with the highest score (scores are determined by the tricks) at the end of the game.

Example 2.14. The two player game of *trumps* (in this simplified version we will only play one round of the game).

The initial trick leader is chosen at random.

A suit is also chosen at random and this is said to be "trumps".

The cards are shuffled and 7 cards are dealt to both players.

In each trick the trick leader can play any card. If the other players have cards of the same suit then they must play one of these. If not they can play any card. Turns are taken in a clockwise order. If no trumps are played then the winning card is the highest card of the same suit as the trick leader. If a trump (a card which has Trumps as its suit) has been played then the winning card is the highest trump. The score of the player who played the winning card is increased by 1. The trick winner is the next trick leader.

Example 2.15. In the game *kings and jacks* the objective is not to take any tricks where kings or jacks have been played.

2.4.2 Shedding Games

In order to show that our method will extend further we decided to apply it to a completely different style of card game.

Definition 2.16. A *shedding game* is one in which players play cards in turn with the objective of running out of cards first.

This doesn't sound all that different. However, there are some major differences. Depending on which cards have been played it is possible to force another player to pick up cards. It is also possible to make the other player miss a turn. Similar to trick based games there are usually restrictions on which cards can be played. Another big difference is that players can choose to (or have to) pass (not lay a card) on their turn.

These games have no fixed length, also there is no measurable score which updates every turn. In fact the best measure of a score is usually how many cards a player has remaining in their hand. However, we wanted our method to work this out on its own. We do not tell the computer that it should be aiming to lose all its cards! For more information on shedding games see section 3.5.

2.5 Answer Set Programming

2.5.1 History

Over the last 15 years much work has been done in the area of Answer Set Programming. This started with the work of Gelfond and Lifschitz in defining the Stable Model Semantics for Logic Programs[9].

Since then work has been focused on developing sophisticated solvers for Answer Set Programs[10] and some simple extensions to the language have led to ASP being very useful for optimisation/planning problems.

2.5.2 Logic programming

Definition 2.17. A *term* can be any of the following:

a *constant* (e.g. six_of_spades, 2 or bob)

a *variable* (e.g. X, Y, Z)

if f is an *n-ary function symbol* and t_1 to t_n are terms then $f(t_1, ..., t_n)$ is a term.

Remark. For the rest of this project we will adopt the widely used convention that function symbols and constants must begin with a lowercase letter and variables must begin with an uppercase letter.

Example 2.18. p, q, bob, X and $card(6, spades)$ are all terms, $\neg p$ is not.

Definition 2.19. A *ground term* is one with no variables.

Definition 2.20. Let R be an *n-ary relation symbol and* $t_1,\ ...\ , t_n$ be terms.

Then $R(t_1,\ ...\ , t_n)$ is an *atomic formula* (or *atom* for short).

$\top$ and $\bot$ are also atomic formulas.

Example 2.21. $p,\ q,\ p(x,\ y)$ and $bought(bob,\ newspaper)$ are all atoms.

Definition 2.22. A *literal* is an atom or a negated atom.

Example 2.23. $p,\ \neg q$ and $\bot$ are all literals.

Definition 2.24. The binary operators we will be using are:

or : $\vee$,

and : $\wedge$,

implies : $\rightarrow$,

if and only if: $\leftrightarrow$

Example 2.25. $A \ \wedge \ B \rightarrow C \ \vee \ D$ means that if A and B are both true it implies that either C is true or D is true (or both).

Definition 2.26. We will also be using the unary negation operator: $\neg$.

This is often referred to as *classical negation.*

Example 2.27. $\neg A$ holds when A is false.

Definition 2.28. A formula of the form $A_1 \vee ... \vee A_n$ is called a *disjunction.* The A_i's are called the *disjuncts.*

Definition 2.29. A *clause* is a disjunction of finitely many literals.

Example 2.30. $p, \ \perp, \ p \vee q,$ and $p \vee q \vee \neg r$ are all clauses.

However, $\neg(p \vee q)$ and $p \wedge q$ are not.

Definition 2.31. A *horn clause* is a clause with at most one positive literal.

Example 2.32. p is a horn clause. So is$\neg p \vee \neg q \vee r$ and also $\neg r$.

However, $\neg p \vee q \vee r$ is not.

Definition 2.33. A *logic program* is a set of horn clauses.

Horn clauses are used in logic programming because they can be interpreted procedurally.

For example the horn clause $p \vee \neg q \vee \neg r$ is equivalent to $q \wedge r \rightarrow p$.

This is often represented in logic programs by $p \ :- \ q, \ r.$ where p is the *head* of the rule and $q, \ r$ is the *body.*

Definition 2.34. A *definite clause* is a horn clause with exactly one positive literal.

Example 2.35. $\neg p \vee \neg q \vee r$ is a definite clause. It is logically equivalent to $p \wedge q \rightarrow r$.

Definition 2.36. A definite clause with no negative literals is called a *fact*.

Definition 2.37. A *definite logic program* is a logic program made up of only definite clauses.

Definition 2.38. *(Negation as failure)* not p means that p has not been shown to hold.

not p (where p is a literal) is allowed to appear in the body of a rule in a logic program.

2.5.3 The Stable Model Semantics

Definition 2.39. The *Herbrand universe* of (the language of) a logic program P is the set of all terms made from constants and function symbols used in P.

Example 2.40. Let P be the logic program:

$p(f(a)) :- q(b), r.$
Then the Herbrand universe of (the language of) P is the set:

$\{a, f(a), f(f(a)), \ldots , b, f(b), f(f(b)), \ldots \}$

Definition 2.41. The *Herbrand base* of (the language of) a logic program P is the set of all ground atoms using terms from its Herbrand universe.

Example 2.42. Let P be the logic program:

$p(f(a)) :- q(b).$
Then the Herbrand base of (the language of) P is the set
$\{p(a), p(f(a)), p(f(f(a))), \ldots , p(b), p(f(b)), p(f(f(b))), \ldots ,$
$q(a), q(f(a)), q(f(f(a))), \ldots , q(b), q(f(b)), q(f(f(b))), \ldots ,$
$r \}$

Definition 2.43. A *Herbrand interpretation* of (the language of) a logic program P assigns a truth value $\bot$ or $\top$ to each of the atoms in the Herbrand base of P.

Remark. We usually represent a Herbrand interpretation as the set of atoms of the Herbrand base which it assigns to $\top$. Any atom not in the set is assigned to $\bot$.

Example 2.44. Let P be the logic program:

p :- not q.
q :- not p.

The Herbrand base of this program is $\{p, q\}$

So there are four Herbrand interpretations:
$\{\}$, $\{p\}$, $\{q\}$ and $\{p, q\}$

Definition 2.45. A *Herbrand model* M of a logic program P is a Herbrand interpretation of (the language of) P such that for every clause C in P:

If, treating all instances of negation by failure as classical negation, the body of C is made true by the assignments of M then the head of C must be in M.

Example 2.46. Let P be the logic program:

p :- not q.
q :- not p.

As before there are four Herbrand interpretations:

1. $\{\}$: This cannot be a Herbrand model as q is not in so "*not q*" is made true by $\{\}$, hence by the first rule p should be in $\{\}$ which clearly is not the case!
2. $\{p\}$: This is a Herbrand model! For the first clause the body is again true but this time our set contains the head of the rule (p). For the second clause the body is false as p is true.
3. $\{q\}$: This is also a Herbrand model. We can see this by symmetry of p and q.
4. $\{p, q\}$: Again, this is a Herbrand model. It makes the body of both clauses false.

Definition 2.47. A set Herbrand model M of a program P is *minimal* if there is no strict subset M^* of M such that M^* is a Herbrand model of P.

Example 2.48. The minimal Herbrand models of the program above are $\{p\}$ and $\{q\}$.

Remark. Every definite logic program has a unique minimal Herbrand model[9].

Definition 2.49. The *grounding* of a clause C of a logic program P is the set of all clauses found by replacing the variables of C with terms from the Herbrand universe of (the language of) P.

Example 2.50. Let P be the logic program:

$p(X) : - q(X).$
$q(a).$
$r(c).$

Then the Herbrand base of (the language of) P is the set:
$\{a, c\}$

So the grounding of the first clause is the set:
$\{\ p(a) :- q(a)., p(b) :- q(b).\ \}$

Definition 2.51. The *grounding* of a logic program P is the logic program containing the *grounding* of each clause in P.

Example 2.52. Let P be the logic program:

$p(X) :- q(X).$
$q(a).$
$r(c).$

Then the grounding of P is the logic program:

$p(a) :- q(a).$
$p(c) :- q(c).$
$q(a).$
$r(c).$

Remark. For many logic programs this grounding is very large but the modern Answer Set Solvers are able to give a much smaller equivalent logic program. It achieves this at ground time by removing any clause C such that the body of C evaluates to false. For example in the previous example the 2nd clause would not appear in the grounding.

Definition 2.53. For a ground logic program P and a set of atoms X the *reduct*, written P^X, is constructed in two steps:

1. remove any clause from P whose body contains the condition *not* q where $q \in X$
2. remove all negation by failure conditions from bodies of the remaining clauses

Remark. What remains is a definite logic program with a unique minimal Herbrand model.

Example 2.54. Let P be the logic program:

p :- not q.
q :- not p.

and let X be the set $\{p\}$.

The second clause is removed by the first step as $p \in X$. However, as $q \notin X$ the first clause is not removed.
So after the first step we are left with the logic program:

p :- not q.
After applying the second step this leaves:

p.
So $P^{\{p\}} = \{p\}$.
This is a definite logic program with the unique minimal Herbrand model $\{p\}$. Similarly $P^{\{q\}} = \{q\}$.

Definition 2.55. A set of atoms X is a *stable model* of a logic program P iff the following equation is true:

$X = M(grounding(P)^X)$

where $M(Q)$ is the unique minimal Herbrand model of the definite logic program Q.

Example 2.56. Let P be the logic program:

p :- not q.
q :- not p.

By the previous example clearly both p and q are stable models.
In fact these are the only stable models of P.

Example 2.57. Consider the logic program P:

$p :- \ not\ p.$

We claim that this program has no stable models.

Proof. Assume there is a set of atoms X such that X is a stable model of P

Case 1: $p \in X$

Then the reduct $grounding(P)^X$ must be empty as the only clause in $grounding(P)$ is $p :- \textit{not } p$. which has *not* p in it's body and $p \in X$.
Clearly the minimal Herbrand model of an empty program must be empty.
So $X = M(grounding(P)^X) = \{\}$.
So X is empty.
Contradiction as X contains p.

Case 2: $p \notin X$

Then the reduct $grounding(P)^X$ is p.
Clearly any Herbrand model must assign p to be true and so the least Herbrand model certainly does.
Hence $p \in M(grounding(P)^X)$
Hence as $X = M(grounding(P)^X)$, X contains p.
Again this is a contradiction.

We reached a contradiction in both cases and so clearly our original assumption must have been false.

The program P has no stable models!

□

Remark. Answer Set Programming also allows for *extended logic programs* which contain classical negation in addition to negation by failure. The definitions above can be extended to cover these however, depending on the interpretation of the result, these can be thought of as giving a set of ground atoms which are known to be true/false and all other ground atoms in the Herbrand base of the program are unknown. In this case we cannot think of the set as a Herbrand interpretation/model as it is no longer an assignment from the Herbrand base of the program to true/false. For this reason we refer to the results as *Answer Sets* rather than stable models, hence the name *Answer Set Programming.*

2.5.4 Language Extensions

As previously mentioned, in ASP rules take the form *head* $:-$ *body*.

Some special rules are:

- facts: rules with no body.

- constraints: rules with no head.
- choice rules: rules with an aggregate (see below) as the head.

Example 2.58. The rule

$p.$

is a fact. It holds when p is true.

Definition 2.59. An *integrity constraint* is a rule with no head. It means that the body of the rule should not be true in an answer set.

Example 2.60. The rule

$: -\ p_1,\ \dots\ ,\ p_n.$

is a constraint.

Its meaning is $\neg(p_1 \wedge\ \dots\ \wedge p_n)$.

Remark. Constraints can be used as a way of eliminating unwanted Answer Sets.

Definition 2.61. A *counting aggregate atom* is of the form $x\{p_1,\ \dots\ ,\ p_n\}y$ for $x,\ y,\ n \in \mathbb{N}$ (x and y are both optional. If missing they are replaced with 0 and n respectively). The meaning is that between x and y of the elements of $\{p_1,\ \dots\ ,\ p_n\}$ are true.

Example 2.62. $1\{p,\ q\}1$ is a counting aggregate atom. It holds when exactly 1 of p and q is true.

$1\{p,\ q\}2$ on the other hand holds when at least 1 of p and q is true.

Definition 2.63. A *choice rule* is a rule with a counting aggregate atom as its head.

Example 2.64. $1\{p,\ q\}2\ :-\ r$

is a choice rule. The meaning is that if r is in an Answer Set then between 1 and 2 of p and q must also be in the answer set.

Definition 2.65. A *summing aggregate atom* is of the form $x[p_1 = a_1,\ \dots\ ,\ p_n = a_n]y$ for $x,\ y,\ n \in \mathbb{N}$. The meaning is that the sum of a_i's whose corresponding p_i is true is between x and y.

The solver we am using for this project, clingo [10], allows for optimisation for example adding the rule

$\#minimize[p(X) = X]$

will find the answer set A which minimises $\sum_{p(X) \in A} X$.

2.5.5 Reasons for Choosing Answer Set Programming over Prolog

If we represent an imperfect information game using logic programming there are going to be some properties which are unknown to a particular player. For example in card games the other player's hand is usually hidden. Representing these unknowns in Prolog is difficult, because we would like to be able to say things like "it is possible that the other player has the queen of hearts". It would have likely involved keeping various lists of possible hands.

However, by using Answer Set Programming we can let each of the Answer Sets correspond to one possible game model. This is a huge improvement on trying to keep track of every game state at once in Prolog. Usually there are hundreds of these so using the techniques described in [1] we only approximate the state space.

The choice rules provide a good way of generating the various models. We have used them to say that, for example, "player(1) has 7 cards from the pack of 52". We have also used the integrity constraints to say things like "no two distinct players have the same card at the same time".

There are parts of the project which could have been implemented in Prolog just as well (but probably no better). However, having chosen ASP for one part of the project it made sense to use it for the whole project.

2.6 Ruby on Rails

We decided from an early stage in the project that the final application should be web based. This was to make it easy for the end user to build, play and share games from any computer and even some mobile devices.

The main reason for choosing Ruby on Rails over other options like PHP was that it has several features which make it easy to build web applications quickly.

These include (but are not limited to):

- Code generation - When adding to the application Rails can generate much of the “boilerplate” code.
- Convention over Configuration - There is no need to specify parts of the application which follow convention.
- Testing - When generating models and controllers Rails can automatically generate some of the tests for them.
- Easy to read - Ruby code is easy to understand.

I also had some experience in building Ruby on Rails applications which we felt would help me get started quickly.

3 Design Approach and Theory

One of the goals we had when first thinking about this project was that the games should be as easy to input as possible. The hope was that an average computer user should be able to create their own game. For this reason there had to be careful thought about the best way for someone without a programming background to design a card game in a way that a computer can easily interpret.

After some thought we rejected designing a simple card game programming language. This was mainly because to be easy enough for an average user to understand, the language wouldn't be powerful enough to easily reflect the structure of the card games.

We therefore decided that the best way would be a flow chart. This is because it is easy enough for an average user to understand, but also has an underlying structure which is very easily interpreted by a computer.

3.1 Overview of the System

Once we have the flow chart for a card game we can already do a lot of the reasoning about how to play the game. As mentioned in section 2.3.2, by utilising hyper play we can model the game as a perfect information game.

Our approach is split into two distinct phases: *pre-game* and *game-time*.

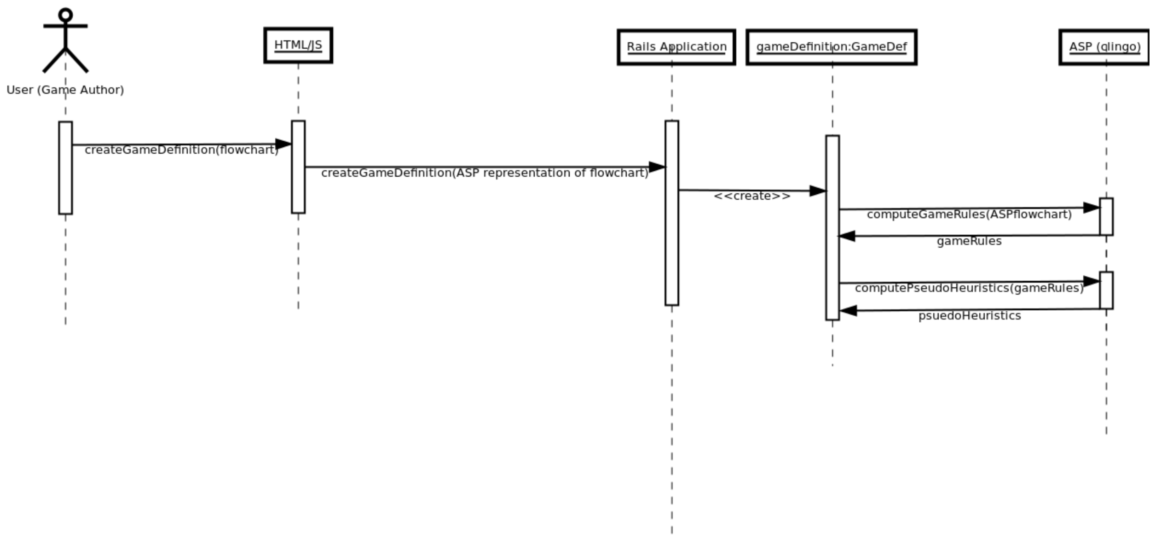

User (Game Author)
HTML/JS
Rails Application
gameDefinition:GameDef
ASP (clingo)
createGameDefinition(flowchart)
createGameDefinition(ASP representation of flowchart)
<<create>>
computeGameRules(ASPflowchart)
gameRules
computePseudoHeuristics(gameRules)
psuedoHeuristics

Figure 3.1: Pre-game pseudo-heuristic generation.

Most of the more time consuming reasoning about how to play the game is done in the *pre-game* phase (See Figure 3.1). We build a pseudo-heuristic (for reasons explained later in this chapter it shouldn't be thought of as a complete heuristic, but can be used to build one). The method for generating this pseudo-heuristic is given at a high level in this chapter and details of the implementation are given in section 4.

At *game-time* (Figure 3.2) this means that the decision making can happen relatively quickly. We use the pseudo-heuristics to give each of the legal options a value. We then pick the move corresponding to the highest value.

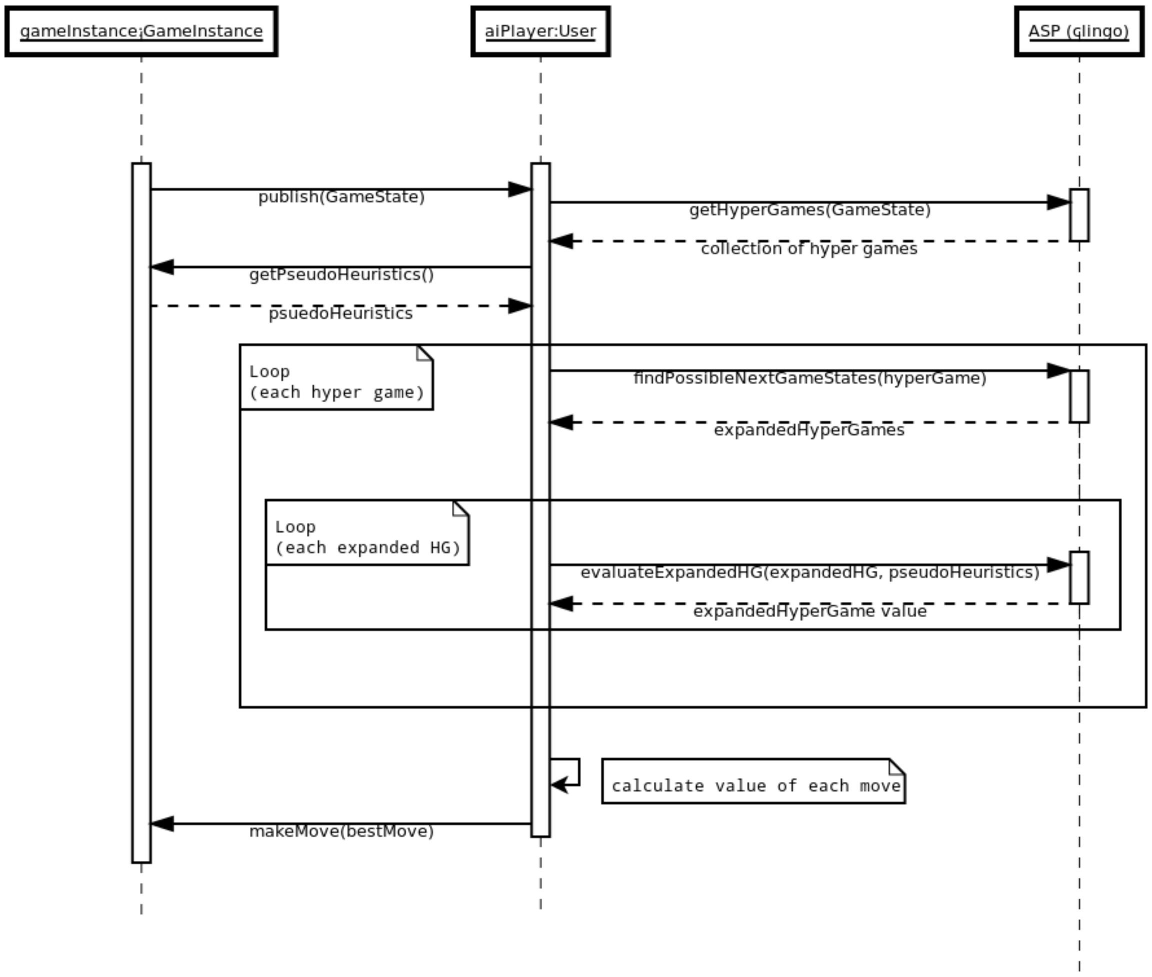


Figure 3.2: Game-time move selection.

3.2 Card Games Represented as a Flow Chart

In our flow chart we use three basic elements (nodes).

Definition 3.1. A *terminal* can be of two types: Start and End. They mark the beginning and end of the game. Exactly one of each exists per game.

They are represented by an ellipse.

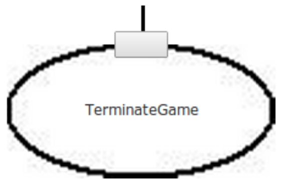


Figure 3.3: A terminal element in one of our flow charts.

Definition 3.2. A *statement* is used to assign variables, publish information or to indicate that an action should take place (e.g. "player 1 should play a heart", "7 cards should be dealt to each player" etc)

They are represented by a rectangle.

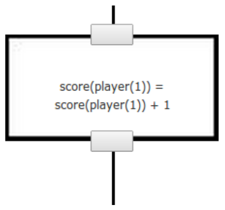


Figure 3.4: A statement element in one of our flow charts.

Definition 3.3. A *choice* is used like an if statement. If the condition holds then the path directly below the choice node should be followed. Otherwise the path to the right should be followed. (the condition could be "player 1 has a spade" etc)

They are represented by a rhombus.

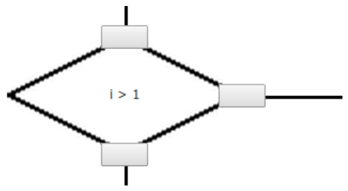


Figure 3.5: A choice element in one of our flow charts.

Within the flow charts the statements allowed are mostly straightforward.

We cover the translation from the flow chart to a *flow chart description language (FCDL)* in section 4. An *FCDL game* is a game based on one of these flow charts.

Remark. The publish statement is used to convey information on the screen of a player e.g. “Hearts were chosen as trumps” or “Player 2 has won the game”.

Definition 3.4. A *trick based game* is one that can be represented as in Figure 3.6. In fact each of the rectangles would be made up of another flow chart determining exactly how each part of the game is played.

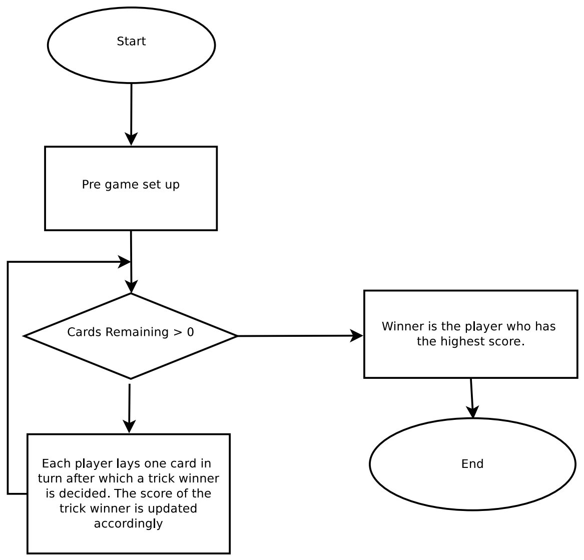

Start
Pre game set up
Cards Remaining > 0
Winner is the player who has the highest score.
Each player lays one card in turn after which a trick winner is decided. The score of the trick winner is updated accordingly
End

Figure 3.6: A “high level” flow chart describing a general *trick based game*.

Example 3.5. For the game of trumps:

- The pre game setup consists of shuffling, dealing and choosing (randomly) which suit is to be trumps and which player will play first (lead).
- The trick is fairly standard:
  - The first player (*trick-leader*) can play any card.
  - After that each player must follow suit if they can.
  - Otherwise they may play any card.
  - After everyone has played the *trick-winner* is decided:
    - If no trumps have been played then the trick-winner is the player who played a card, of the same suit as the trick-leader, which has the highest rank.

        * If a trump has been played then the trick-winner is the player who played the highest ranked trump.

    - The trick-winner is awarded one point and the played cards are collected and discarded.

- The winner is the player which the most points (the one who took the most tricks).

Example 3.6. The *trick* for the game of Trumps can be represented as a flow chart. This is shown in Figure 3.7.

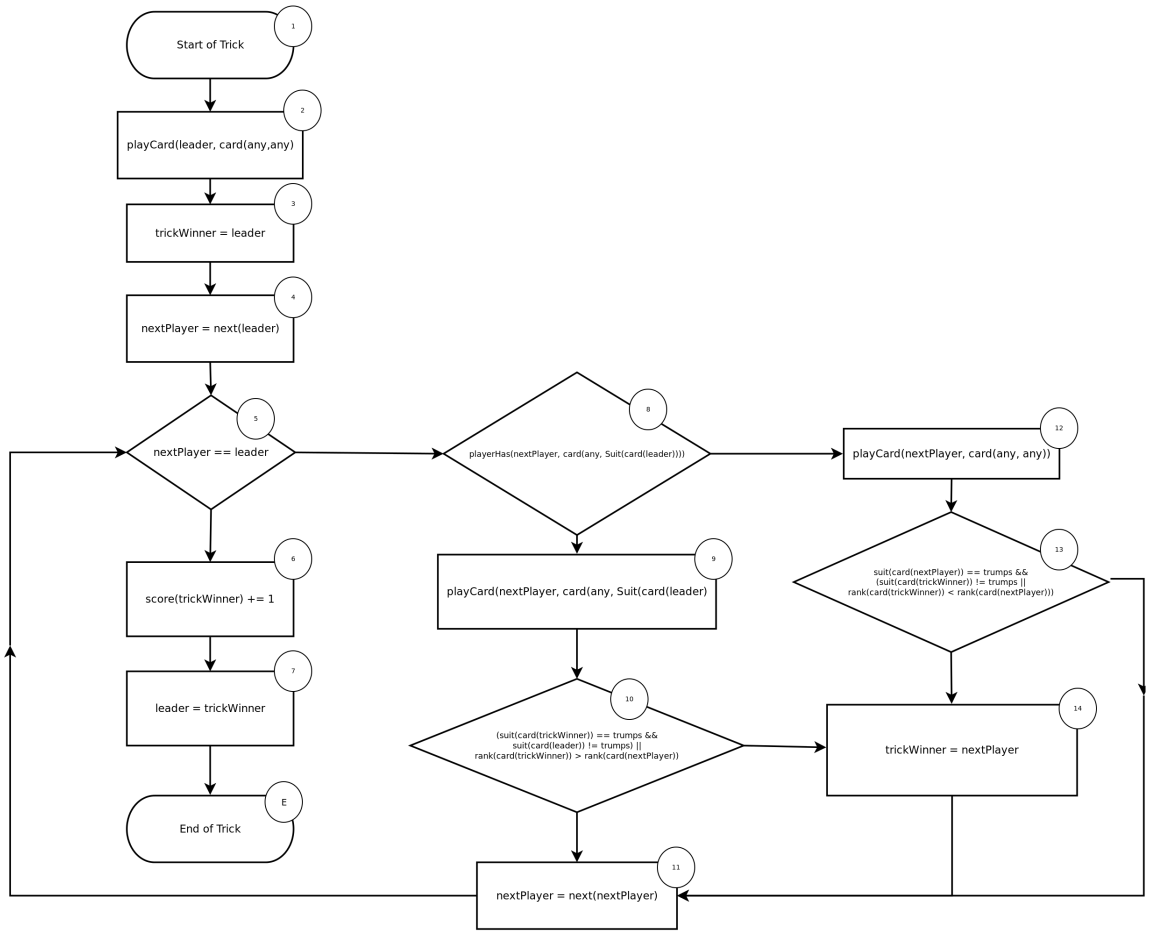

Start of Trick
1
playCard(leader, card(any,any)
2
trickWinner = leader
3
nextPlayer = next(leader)
4
nextPlayer == leader
5
score(trickWinner) += 1
6
leader = trickWinner
7
End of Trick
E
playerHas(nextPlayer, card(any, Suit(card(leader))))
8
playCard(nextPlayer, card(any, Suit(card(leader)
9
(suit(card(trickWinner)) == trumps && suit(card(leader)) != trumps) || rank(card(trickWinner)) > rank(card(nextPlayer))
10
nextPlayer = next(nextPlayer)
11
playCard(nextPlayer, card(any, any))
12
suit(card(nextPlayer)) == trumps && (suit(card(trickWinner)) != trumps || rank(card(trickWinner)) < rank(card(nextPlayer)))
13
trickWinner = nextPlayer
14

Figure 3.7: The flow chart for the game of *trumps*.

3.3 Pre-Game Pseudo-Heuristic Generation for Trick Based Games

In order to play a trick based game well a player must determine the value of their hand compared with the expected value of their opponent's hand.

As we are using hyper-play we just need a way of determining the value of a player's hand given a model for the opponent's hand.

Here we propose a simple method for doing so.

1. Given our cards and the opponent's cards, calculate all possible tricks which could be legal in the *next* trick and calculate the change in score for each player.
2. Calculate the possible tricks which might be legal later (a trick may not be legal given a player's current hand, but later in the game it may be).

This can be used to calculate a rough estimate of the expected tricks a player will take in the remaining game. This is our heuristic.

Calculating this at game-time would be very time consuming (especially when the method is extended to a more general game type).

Therefore we have a simple algorithm which generates much of the information in the pre-game phase. This information is computed solely from the game rules and so will apply to any instance of that game. This is run *once per game definition.*

We do not refer to the result of the algorithm as a full heuristic as strictly it is not. However, when combined with the method described in section 3.4 it does give a full heuristic. Therefore we refer to the result as a set of *pseudo-heuristics.*

Definition 3.7. An *instance* of a path through the trick of a trick based FCDL game is the path together with an assignment to any variables which represent players.

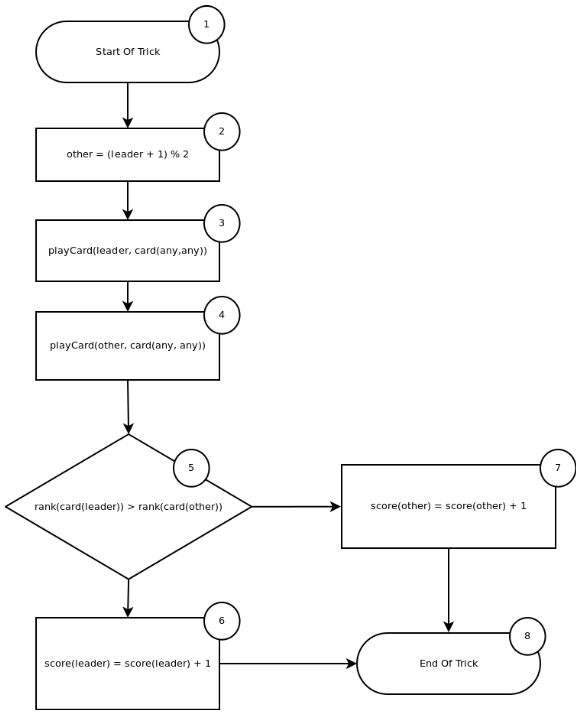
Example 3.8. In the trick of the game of trumps each path has exactly two instances. One where $leader$ is $player(0)$ (and $nextPlayer$ is $player(1)$) and the other where $leader$ is $player(1)$ ($nextPlayer$ is $player(0)$).

Definition 3.9. A *pseudo-heuristic* for a trick based FCDL game is an integer W and a set of conditions C. Each pseudo-heuristic maps to an instance of a path through the trick where W is equal to the change in score to the AI player and C is the set of conditions needed to travel along the path.

```
Given the rules of the game:

  1) expand the body of the trick to
     one with no sub loops.

  2) calculate all paths through this
     expanded loop body. Paths may have multiple
     instances (if x is told to play a card
     then there is an instance for if the AI
     player is x and an instance for if the other player is x).

  For each instance of each path:

    3) calculate the changes in the variables
       made along each path and the conditions
       on the initial values of variables for
       each of the paths to be taken.

    4) The path instance is given
       the weighting $changeInAIPlayersScore - changeInOtherPlayersScore$
```

Figure 3.8: A simple algorithm to compute the pseudo-heuristics of a trick based game.

Example 3.10. Consider the two player game with a simple "trick" where both players lay a card and then the player who laid the card with the higher rank is awarded a point. The flow chart is shown below.


1
Start Of Trick
2
other = (leader + 1) % 2
3
playCard(leader, card(any,any))
4
playCard(other, card(any, any))
5
rank(card(leader)) > rank(card(other))
7
score(other) = score(other) + 1
6
score(leader) = score(leader) + 1
8
End Of Trick

If we apply our algorithm:

Step 1: The trick doesn't have any sub loops, therefore this step doesn't change anything.

Step 2: There are clearly two paths through the trick:

(a) {1, 2, 3, 4, 5, 6, 8} (The path taken if *leader* won the trick)

(b) {1, 2, 3, 4, 5, 7, 8} (The path taken if *other* won the trick)

Step 3: Both paths change the values of the cards played by the players.

(a) This path increases the score of *leader* by 1.

(b) This path increases the score of *other* by 1.

Step 4: (a) {1, 2, 3, 4, 5, 6, 8} This path has two instances, one where *leader* is the AI player, and one where *other* is the AI player (If it wasn't for statement 2 there could potentially be four instances).

i. In the instance where the AI player is *leader* the weighting of this path is 1.

ii. In the other instance the weighting is -1.

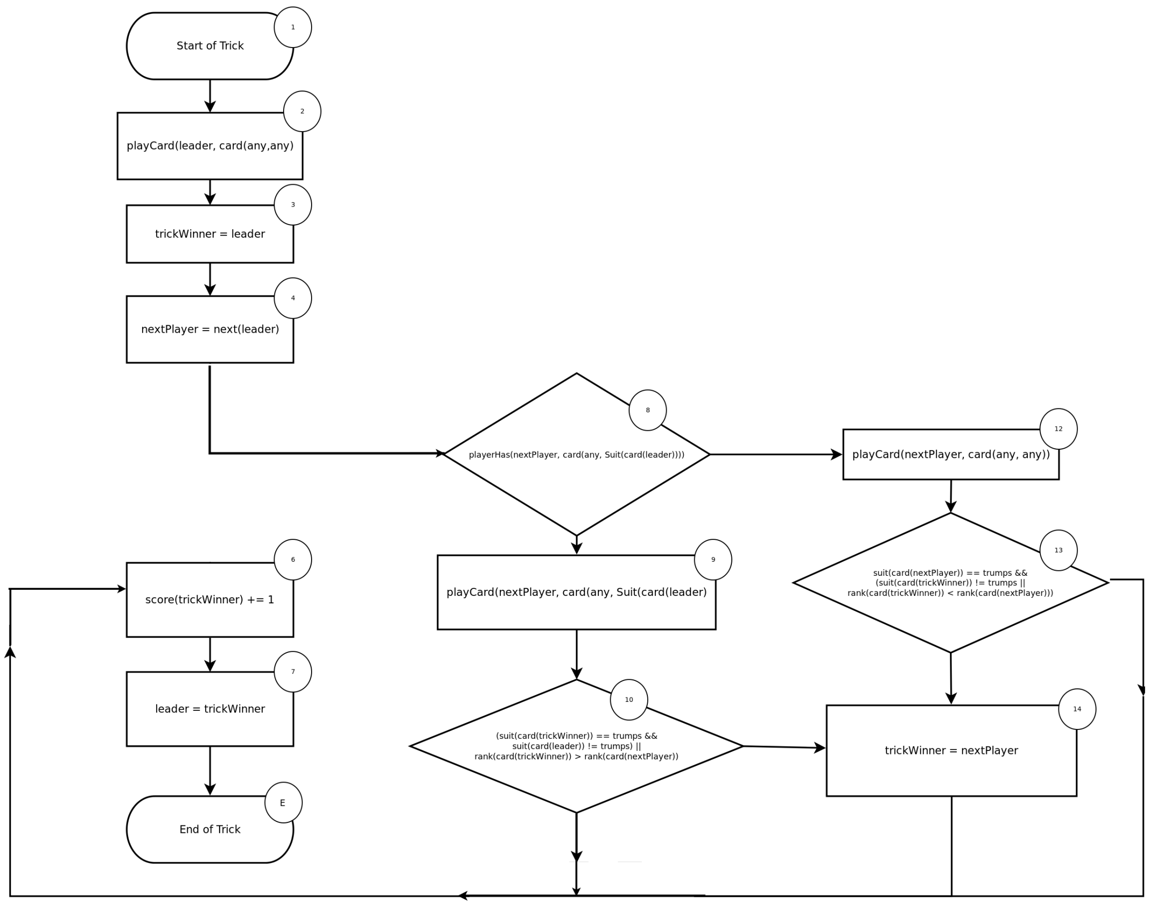
(b) {1, 2, 3, 4, 5, 7, 8} Again this path has two instances,

i. In the instance where the AI player is *leader* the weighting of this path is -1.

ii. In the other instance the weighting is 1.

Example 3.11. For the two player game of trumps:

Step 1: As we know the number of players the main loop of the trick can be flattened. In fact as there are two players the program doesn't appear to change too much.



Start of Trick
1
playCard(leader, card(any,any)
2
trickWinner = leader
3
nextPlayer = next(leader)
4
playerHas(nextPlayer, card(any, Suit(card(leader))))
8
playCard(nextPlayer, card(any, any))
12
score(trickWinner) += 1
6
playCard(nextPlayer, card(any, Suit(card(leader)
9
suit(card(nextPlayer)) == trumps && (suit(card(trickWinner)) != trumps || rank(card(trickWinner)) < rank(card(nextPlayer)))
13
leader = trickWinner
7
(suit(card(trickWinner)) == trumps && suit(card(leader)) != trumps) || rank(card(trickWinner)) > rank(card(nextPlayer))
10
trickWinner = nextPlayer
14
End of Trick
E

Figure 3.9: The trumps trick has now been "flattened".

Step 2: The paths through this loop-less program are:

(a) {1, 2, 3, 4, 8, 9, 10, 6, 7, E}
(b) {1, 2, 3, 4, 8, 9, 10, 14, 6, 7, E}
(c) {1, 2, 3, 4, 8, 12, 13, 14, 6, 7, E}
(d) {1, 2, 3, 4, 8, 12, 13, 6, 7, E}

Step 3: (a) The conditions for this path are:

  i. *nextPlayer* has a card of the same suit as the card played by the *leader*.
  ii. either the trick has already been *trumped* by another player (This actually can't be the case in the two player version of the game so we will ignore this condition) or the rank of the card played by the current *trickWinner* is higher than the rank of the card played by *nextPlayer*.

Or in FCDL (for the two player game):

  i. playerHas(nextPlayer, card(any, suit(middle(leader)))).
  ii. greaterThan(rankValue(middle(trickWinner)), rankValue(middle(nextPlayer))).

  Note we also have two extra conditions on the cards which were played:
  iii. cardInstanceOf(middle(leader), card(any, any)).
  iv. cardInstanceOf(middle(nextPlayer), card(any, suit(middle(leader)))).

This path has two instances:

  i. if *leader* is the AI player then the weighting is 1. The full pseudo-heuristic (with weighting 1) in this case is:
    A. playerHas(OtherPlayer, card(any, suit(middle(AIPlayer)))).
    B. greaterThan(rankValue(middle(AIPlayer)), rankValue(middle(OtherPlayer))).
    C. cardInstanceOf(middle(AIPlayer), card(any, any)).
    D. cardInstanceOf(middle(OtherPlayer), card(any, suit(middle(AIPlayer)))).
    E. equal(player(leader), AIPlayer).
  ii. if *leader* is the other player then the weighting is -1. The full pseudo-heuristic (with weighting -1) in this case is:
    A. playerHas(AIPlayer, card(any, suit(middle(OtherPlayer)))).

B. greaterThan(rankValue(middle(OtherPlayer)), rankValue(middle(AIPlayer))).

C. cardInstanceOf(middle(OtherPlayer), card(any, any)).

D. cardInstanceOf(middle(AIPlayer), card(any, suit(middle(OtherPlayer)))).

E. equal(player(leader), OtherPlayer).

(b) The conditions for this path are:

i. *nextPlayer* has a card of the same suit as the card played by the *leader*.

ii. The trick has not already been *trumped* by another player (can't be the case in a two player game so we will ignore this condition).

iii. the rank of the card played by the current *trickWinner* is not higher than the rank of the card played by *nextPlayer*.

Or in FCDL (for the two player game):

i. playerHas(nextPlayer, card(any, suit(middle(leader)))).

ii. neg(greaterThan(rankValue(middle(trickWinner)),rankValue(middle(nextPlayer)))).

iii. cardInstanceOf(middle(leader), card(any, any)).

iv. cardInstanceOf(middle(nextPlayer), card(any, suit(middle(leader)))).

This path has two instances:

i. if *leader* is the AI player then the weighting is -1. The full pseudo-heuristic (with weighting -1) in this case is:

A. playerHas(OtherPlayer, card(any, suit(middle(AIPlayer)))).

B. neg(greaterThan(rankValue(middle(AIPlayer)),rankValue(middle(OtherPlayer)))).

C. cardInstanceOf(middle(AIPlayer), card(any, any)).

D. cardInstanceOf(middle(OtherPlayer), card(any, suit(middle(AIPlayer)))).

E. equal(player(leader), AIPlayer).

ii. if *leader* is the other player then the weighting is 1. The full pseudo-heuristic (with weighting 1) in this case is:

A. playerHas(AIPlayer, card(any, suit(middle(OtherPlayer)))).

B. neg(greaterThan(rankValue(middle(OtherPlayer)),rankValue(middle(AIPlayer)))).

C. cardInstanceOf(middle(OtherPlayer), card(any, any)).

D. cardInstanceOf(middle(AIPlayer), card(any, suit(middle(OtherPlayer)))).

E. equal(player(leader), OtherPlayer).

(c) The conditions for this path are:

i. *nextPlayer* has no card of the suit played by *leader*.
ii. *nextPlayer* plays a card of suit *trumps*.
iii. either the current *trickWinner* played a card with a suit other than trumps or a card with rank lower than that played by *nextPlayer*.

Or in FCDL (for the two player game):

i. neg(playerHas(nextPlayer, card(any, suit(middle(leader))))).
ii. equal(suit(middle(nextPlayer)), trumps)
iii. operation(or, neg(equal(suit(middle(trickWinner)), trumps)), lessThan(rankValue(middle(trickWinner)), rankValue(middle(nextPlayer)))).
iv. cardInstanceOf(middle(leader), card(any, any)).
v. cardInstanceOf(middle(nextPlayer), card(any, any)).

This path has two instances:

i. if *leader* is the AI player then the weighting is -1. The full pseudo-heuristic (with weighting -1) in this case is:
   A. neg(playerHas(OtherPlayer, card(any, suit(middle(AIPlayer))))).
   B. equal(suit(middle(OtherPlayer)), trumps)
   C. operation(or, neg(equal(suit(middle(AIPlayer)), trumps)), lessThan(rankValue(middle(AIPlayer)), rankValue(middle(OtherPlayer)))).
   D. cardInstanceOf(middle(AIPlayer), card(any, any)).
   E. cardInstanceOf(middle(OtherPlayer), card(any, any)).
   F. equal(player(leader), AIPlayer).
ii. if *leader* is the other player then the weighting is 1. The full pseudo-heuristic (with weighting 1) in this case is:
   A. neg(playerHas(AIPlayer, card(any, suit(middle(OtherPlayer))))).
   B. equal(suit(middle(AIPlayer)), trumps)
   C. operation(or, neg(equal(suit(middle(OtherPlayer)), trumps)), lessThan(rankValue(middle(OtherPlayer)), rankValue(middle(AIPlayer)))).

D. cardInstanceOf(middle(OtherPlayer), card(any, any)).
E. cardInstanceOf(middle(AIPlayer), card(any, any)).
F. equal(player(leader), OtherPlayer).

(d) The conditions for this path are:

i. *nextPlayer* has no card of the suit played by *leader*.
ii. either *nextPlayer* did not play a trump, the current *trickWinner* played a trump which is of higher rank than the card played by *nextPlayer*.

Or in FCDL (for the two player game):

i. neg(playerHas(nextPlayer, card(any, suit(middle(leader))))).
ii. operation(or, neg(equal(suit(middle(nextPlayer)), trumps)),
operation(and, equal(suit(middle(trickWinner)),trumps),
greaterThan(rankValue(middle(trickWinner)),
rankValue(middle(nextPlayer))))).
iii. cardInstanceOf(middle(leader), card(any, any)).
iv. cardInstanceOf(middle(nextPlayer), card(any, any)).

This path has two instances:

i. if *leader* is the AI player then the weighting is 1. The full pseudo-heuristic (with weighting 1) in this case is:
A. neg(playerHas(OtherPlayer, card(any, suit(middle(AIPlayer))))).
B. operation(or, neg(equal(suit(middle(OtherPlayer)), trumps)),
operation(and, equal(suit(middle(AIPlayer)),trumps),
greaterThan(rankValue(middle(AIPlayer)),
rankValue(middle(OtherPlayer))))).
C. cardInstanceOf(middle(AIPlayer), card(any, any)).
D. cardInstanceOf(middle(OtherPlayer), card(any, any)).
E. equal(player(leader), AIPlayer).
ii. if *leader* is the other player then the weighting is -1. The full pseudo-heuristic (with weighting -1) in this case is:
A. neg(playerHas(AIPlayer, card(any, suit(middle(OtherPlayer))))).
B. operation(or, neg(equal(suit(middle(AIPlayer)), trumps)),
operation(and, equal(suit(middle(OtherPlayer)),trumps),
greaterThan(rankValue(middle(OtherPlayer)),
rankValue(middle(AIPlayer))))).

C. cardInstanceOf(middle(OtherPlayer), card(any, any)).
D. cardInstanceOf(middle(AIPlayer), card(any, any)).
E. equal(player(leader), OtherPlayer).

As mentioned before these paths do not give a full heuristic, but they can be used to find one. The basic idea is that we find all the instances of these paths (with our cards and the opponent's cards).

Methods for converting this to a full heuristic are given in the next section.

For example one basic approach would be to take the average weighting of those pseudo-heuristic instances for which all the conditions are true.

Remark. Here each of the heuristic components have the same weighting (if we ignore the sign). This is because if the trick is played the value to the player is either +1 (if the player wins the trick) or -1 (if the player loses the trick).

3.4 Move Selection

The previous section talked about *pseudo-heuristics*. Although, as described in this section, they can be used to create a full heuristic they are not a full heuristic or even a *feature* (see definition 2.8).

So far we have seen how to calculate our pseudo-heuristics pre-game and later we will see how to generate many hyper games. We now need to combine all the information we have in order to give a full heuristic. This will enable us to decide on the best move in any situation.

3.4.1 Reasons for Using the Minimax Algorithm

Given the rules G to the game.

Let $G*$ be the set of facts known to the AI player about the game so far and let H_{G*} be a set of randomly generated hyper games for $G*$. and our set of pseudo-heuristics PH.

For each hyper game $h \in H_{G*}$ and legal move m we must calculate the value to our player of being in the state immediately after they have made the move m.

However, calculating the value of being in this state may not be clear.

Example 3.12. In the game of *trumps*, say our hyper game h contains the following information:

trumps: hearts

my_hand: {ten of hearts}

other_hand: {jack of hearts, two of spades}

Our player has just played the queen of hearts.

If we were to examine the possible instances of the heuristics at this precise point in the game we would find that *others* jack of hearts would beat our player's ten of hearts. However, *other* has to follow suit in the current trick and so they are forced to play their jack of hearts now. This means that their jack of hearts cannot possibly beat our player in the next trick as they won't have it. In fact, the rest of the tricks are ours.

This simple example shows that it is not a good idea to evaluate the game state half way through a trick. Therefore we chose to always evaluate the game state at the end of a trick.

The way we do this is to use the minimax theorem to calculate the optimal strategies for each player assuming that they both want to be in the best position they can be at the end of the trick.

The Minimax algorithm with alternate moves applies clearly to two player trick based games. We only use it however, up to the end of the current trick. This is because we wanted the main focus of the work to be on the heuristic evaluation and wanted to show that the reasoning behind the heuristic evaluation really was good enough to play the games. If we were to continue further then, once the heuristic evaluation had been demonstrated as good enough, we would utilise minimax more.

In a two player trick based game there are two possibilities. One where our AI player is the trick leader. In this case we assume that the our opponent will make a move which will minimize the game state value (for our AI player).

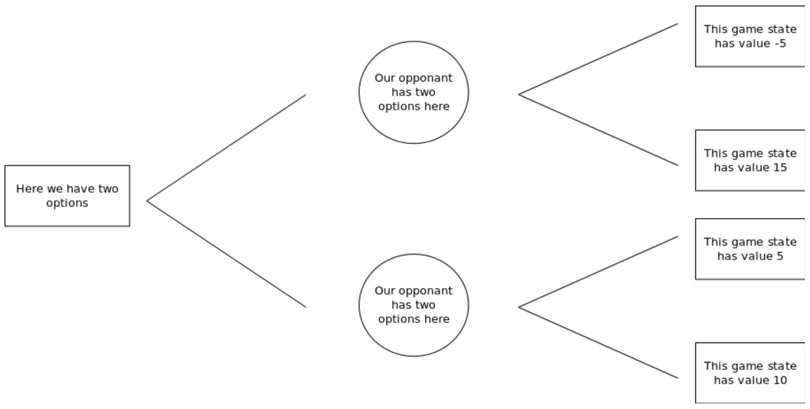

Here we have two options
Our opponant has two options here
Our opponant has two options here
This game state has value -5
This game state has value 15
This game state has value 5
This game state has value 10

Figure 3.10: The tree for a simple trick in which both players have two options.

Example 3.13. In Figure 3.10 we can see that the AI player has two options. If we make the assumption that the other player will make the choice which minimises the value to our player of the next game state then the tree "folds back" to give the smaller tree in Figure 3.11.

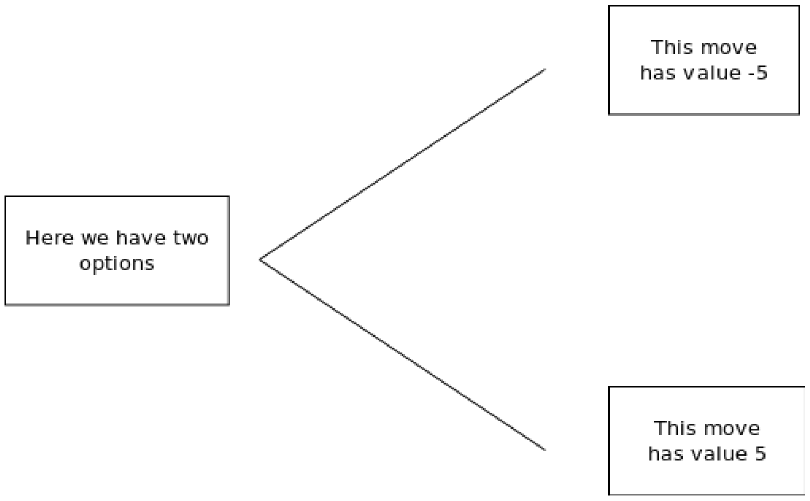


Figure 3.11: The tree "folded back" tree for a simple trick in which both players have two options.

We can clearly see here that our AI player should take the 2nd option, even though there was a possible game state following from the 1st option with a higher value.

In the case where our AI player is not the trick leader then by the time our algorithm is required to make a decision we already know which move the other player has made. Therefore in this case for each legal move we could make we have one unique value.

Example 3.14. Consider the game tree (see Figure 3.12) where it is our AI player's turn and the other player has already had their turn in the trick.

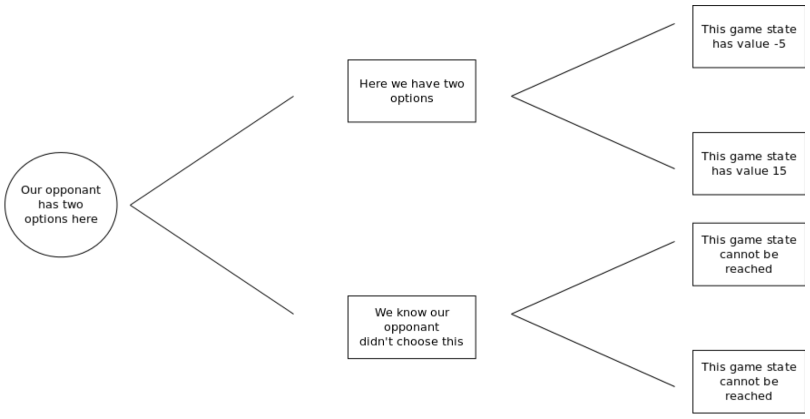

Our opponant has two options here
Here we have two options
We know our opponant didn't choose this
This game state has value -5
This game state has value 15
This game state cannot be reached
This game state cannot be reached

Figure 3.12: The tree for a simple trick in which both players have two options but the other player has already made their choice.

Notice that what the other player could have chosen becomes irrelevant now. We only have to consider the future game states which are still possible. The other player has no influence over this and so we get a unique value for each possible move that our AI player could make (see Figure 3.13).

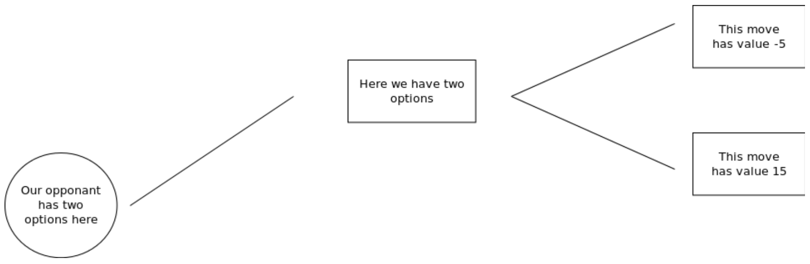


Figure 3.13: The tree for a simple trick in which both players have two options but the other player has already made their choice (irrelevant parts stripped away).

Notice that in both cases the value we assign to each move can be found by taking the minimal value over the future game states our AI player could be in after making the move. For this reason, we do not have to think much about minimax (or the two separate cases) when we are implementing, we simply have to minimise.

3.4.2 Hyper Game Evaluation

Now we have a hyper game which has been extended to a possible game state at the start of the next trick, we need to give this a value.

Clearly the current scores of each player are big factors in calculating this value; however, the more interesting part is to find a good estimate of the expected scores at the end of the game.

To do this we must estimate how many tricks we expect to take in the remainder of the game.

Definition 3.15. An *instance* of a pseudo-heuristic PH substitutes any instance of $middle(AIPlayer)$ in a condition of PH with a card in the AI player's hand, and all instances of $middle(OtherPlayer)$ with a card from the other player's hand.

Definition 3.16. An instance of a pseudo-heuristic PH is *active* at a step S in a hyper game H if $\forall c \in conditions(PH)$ are true given the values of variables at step S in HG.

Example 3.17. This is an instance of pseudo-heuristic 1 from the trumps example.

(a) playerHas(OtherPlayer, card(any, suit(card(6,hearts)))).

(b) greaterThan(rankValue(card(6,hearts)), rankValue(card(4,hearts))).

(c) cardInstanceOf(card(6,hearts), card(any, any)).

(d) cardInstanceOf(card(4,hearts), card(any, suit(card(6,hearts)))).

(e) equal(player(leader), AIPlayer).

Note: if AIPlayer == player(leader) then this pseudo heuristic is active as all the conditions are true. (condition (a) must be true as OtherPlayer has the 4 of hearts).

One possible approach as mentioned in the previous section would be simply to average the weightings of the active pseudo-heuristics instances at the start of the next trick and multiply by the number of tricks remaining. However, this can be misleading as some cards may have move active pseudo-heuristics than others (but they can each only be played once). For this reason we decided to average the pseudo-heuristics *per card in the AI player's hand.* This gives a much more accurate estimate of the expected tricks.

So the value of the game state is:

$$value_{game\_state}(GameState) = \sum_{card \in AIHand} value_{card}(Card)$$

However, it does still make one (possibly false) assumption. The current values of the conditions will be true for the rest of the game.

Example 3.18. Notice each of the pseudo-heuristics calculated for the game of trumps has a condition about the value of player(leader). This means that only half of them can have active instances at any one time.

This could cause a player not to see the value of a card which is of high value to them when they are following (eg a low trump) if they are currently the leader.

The final function we decided on attempts to overcome this by considering first (with a higher weighting) those heuristics which are active now. It then finds those heuristics which might be active in a few steps time and calculates how many steps this might take (weighting the results accordingly).

Definition 3.19. A pseudo-heuristic is *semi-active* if when we remove conditions involving a negative playerHas predicate or the conditions about which player is which it becomes active. Its *count* is defined as the number of things which need to change for it to become active (If a player has 2 hearts and one of the conditions is that they have no hearts then the count would be 2).

Remark. An active pseudo-heuristic is also semi-active with count 0.

So the final $value_{card}$ function is defined as:

$$value_{card}(Card) = \frac{\sum_{h \in SAH(Card)} \frac{weighting(h)}{(1+count(h))^2}}{|SAH(Card)|}$$

where $SAH(Card)$ is the set of semi-active pseudo-heuristics for $Card$.

3.4.3 Given All of This Information What is the Best Move?

We now have a collection of hyper games H_G of G each of which has its own collection of game states which are possible at the start of the next trick depending on the move that each player makes. Each of these game states has a value given by the last section.

For each $h \in H_G$

We know that currently it is our AI player's move. So we can group the collection of possible game states according to the move m we made.

We decided to use the minimax with alternate moves algorithm. Hence, as our player does not have another move before the next trick, we know that if we play m we can expect the value of the game state to be the minimal value of a game state where our player played m.

Therefore by minimax/maximin we should play the move m^* which maximises this minimal value

3.5 Shedding Games

We now have a method which works for trick based card games. However, the idea of this project is to show that this method is actually applicable to much more general card games. Obviously in the time frame of this project it was not possible to create something which works for all card games. Instead we show that our algorithm extends (without much effort) to cover *both* trick based games *and* a new very different type of card game.

For this game type we chose *shedding games*

Definition 3.20. A shedding game is one which takes the following form:

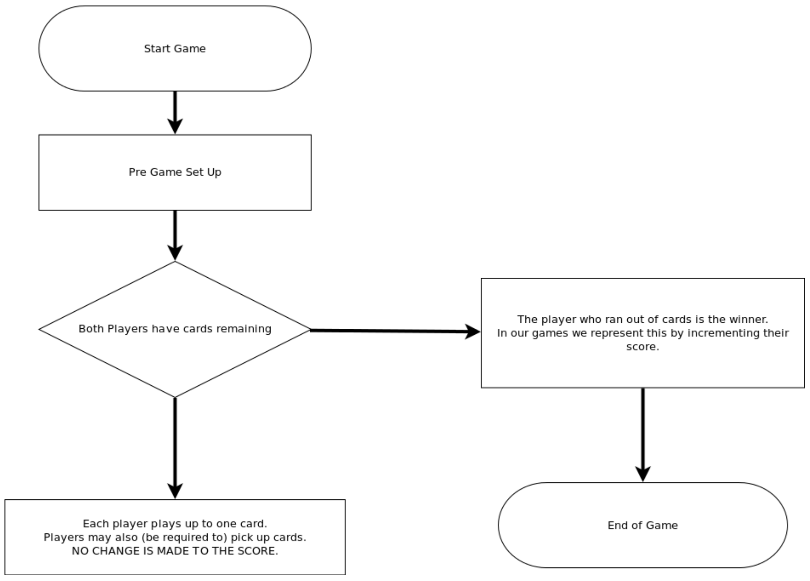

Start Game
Pre Game Set Up
Both Players have cards remaining
The player who ran out of cards is the winner.
In our games we represent this by incrementing their score.
Each player plays up to one card.
Players may also (be required to) pick up cards.
NO CHANGE IS MADE TO THE SCORE.
End of Game

Figure 3.14: The high level flow chart for a shedding game.

Example 3.21. Flow chart for the game of a simplified version of the game of *crazy eights*

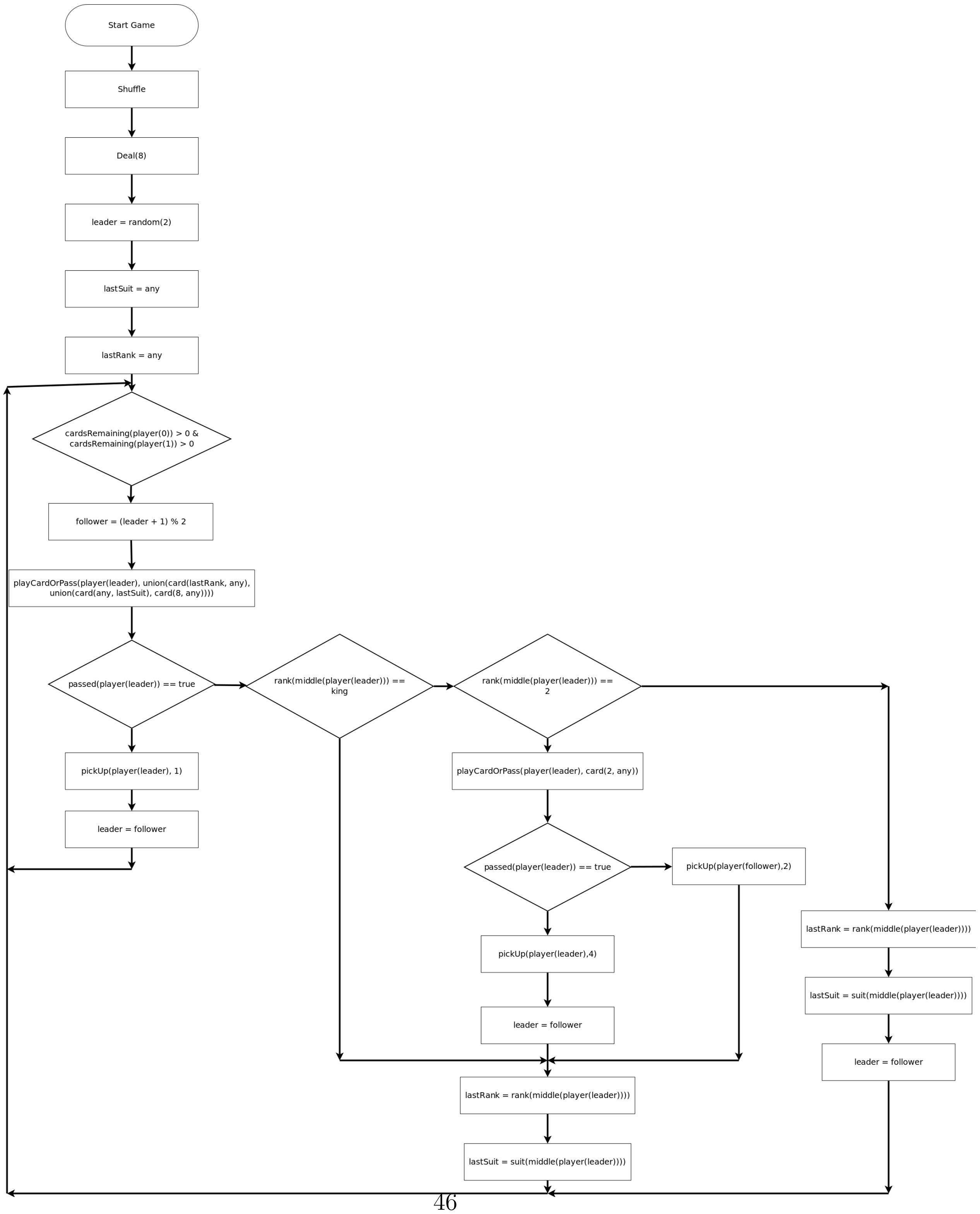

Start Game
Shuffle
Deal(8)
leader = random(2)
lastSuit = any
lastRank = any
cardsRemaining(player(0)) > 0 & cardsRemaining(player(1)) > 0
follower = (leader + 1) % 2
playCardOrPass(player(leader), union(card(lastRank, any), union(card(any, lastSuit), card(8, any))))
passed(player(leader)) == true
rank(middle(player(leader))) == king
rank(middle(player(leader))) == 2
pickUp(player(leader), 1)
leader = follower
playCardOrPass(player(leader), card(2, any))
passed(player(leader)) == true
pickUp(player(follower),2)
pickUp(player(leader),4)
leader = follower
lastRank = rank(middle(player(leader))))
lastSuit = suit(middle(player(leader))))
lastRank = rank(middle(player(leader))))
lastSuit = suit(middle(player(leader))))
leader = follower

Figure 3.15: The flow chart for crazy eights.

The immediately obvious differences with this new game type are

1. The games are not of fixed length!
2. Players can be forced to pick up more cards!
3. Players do not play a fixed number of cards inside the body of the main loop.
4. There is not a change to the score in the main loop.

Due to the way we defined our original algorithm, by considering paths through the main loop rather than paths through the whole game the first difference actually doesn't change our approach.

Players being forced to pick up cards doesn't change things either.

That players do not play a fixed number of cards inside the main loop again isn't actually a problem, due to the way we designed the original algorithm. The constraint we do have however is that players must play a maximum of one card inside this main loop. This constraint could be removed later but as it considerably simplified the implementation we chose to leave it in.

The fact that there is not a change to the score in the main loop is more challenging. It means that we need to first find properties outside of the loop which we wish to maximise (or minimise). These should be the properties which any change to the score depends on. We should then find the main loop in the same way as before.

3.5.1 A More General Algorithm for Pseudo-Heuristic Generation

This section attempts to generalise the trick based pseudo-heuristic generation algorithm given in Figure 3.8.

Firstly, we would not like our algorithm to consider the two types of games separately but rather, we would like to find a class of games which contains both.

Definition 3.22. We call an FCDL game *repetitive* if there is a loop in the game which contains all player moves.

Definition 3.23. For an FCDL game G, we call the smallest loop of G which contains all player moves and contains all changes to a particular property which should be maximised/minimised *the main loop* of G.

Remark. For a trick based game this main loop is the *trick* and the property which should be maximised is the *score*.

Definition 3.24. We call part of an FCDL game *strictly finite* if there is enough information to transform this part of the flow chart into an equivalent flow chart with no loops (pre-game).

Definition 3.25. We call a repetitive FCDL game *mainly finite* if its main loop is strictly finite.

The algorithm defined in this chapter will find a collection of pseudo-heuristics for all mainly finite games in which either all changes to the player's scores are contained inside the main loop or all changes to the player's scores occur after the main loop (Any changes to the player's scores before the main loop would be unfair as neither has played a card yet).

Remark 3.26. Note for all games which we have discussed so far the main loop of the game has been the largest loop in the flow chart. We have only tested the implementation of the algorithm on games of this kind; however, we believe that the principles of the algorithm should extend to all mainly finite games. This would be the next step in any future work on this project.

Example 3.27. Trumps is mainly finite. This can be seen by considering Figure 3.9.

Remark. Shedding games are mainly finite (The property which should be minimised in their main loop is the number of cards remaining in a player's hand).

Algorithm:

```
1) First we need to find the main loop of the game.

2) expand the body of the main loop to
   one with no sub loops.

3) calculate all paths through this
   expanded loop body. Paths may have multiple
   instances (if x is told to play a card
   then there is an instance for if the AI
   player is x and an instance for if the other player is x).

4) Next we find any changes to the scores of the players
   which occur outside of the main loop.
   - If they are desirable (ie an increase to our score) we list the conditions
       which led to this
     as a maximise condition.
   - If they are undesirable we list their conditions as a minimise condition.

5) calculate the changes in the variables
   made along each path and the conditions
   on the initial values of variables for
   each of the paths to be taken.

6) Each instance of each path is given
   the weighting totalChangeInMaximiseProperties - totalChangeInMinimiseProperties.
```

This is simply an extension of the algorithm for trick based games. It has been tested with several trick based games and shedding games and is known to give a good result (see later). We believe that it will also give a good result for all mainly finite games subject to the condition specified by remark 3.26.

4 Implementation

4.1 System Architecture

The main benefit to the project of using Answer Set Programming was that it was well suited to finding the sets of Hyper Games. It was also a good tool for calculating the next game state and for finding the pseudo-heuristics. However, there are many parts of the project for which ASP is not appropriate. Clearly in order to build a User Interface we needed to use something else. As we wanted the games to be web based we decided to use the Ruby on Rails framework.

We implemented a Ruby on Rails application which was responsible for the various user interfaces for game design and game play, managing the database where the game definitions were stored and most importantly running ASP and interpreting the results.

There are some parts of our logic programs which are fixed (general rules which apply to all card games that we have considered). Other rules and facts are dynamic. They depend on the flow chart of the game and on the current game state. Whenever ASP is "called" on our server Ruby has had to prepare the dynamic part of the program and run ASP on the full logic program. Ruby then interprets the results from ASP.

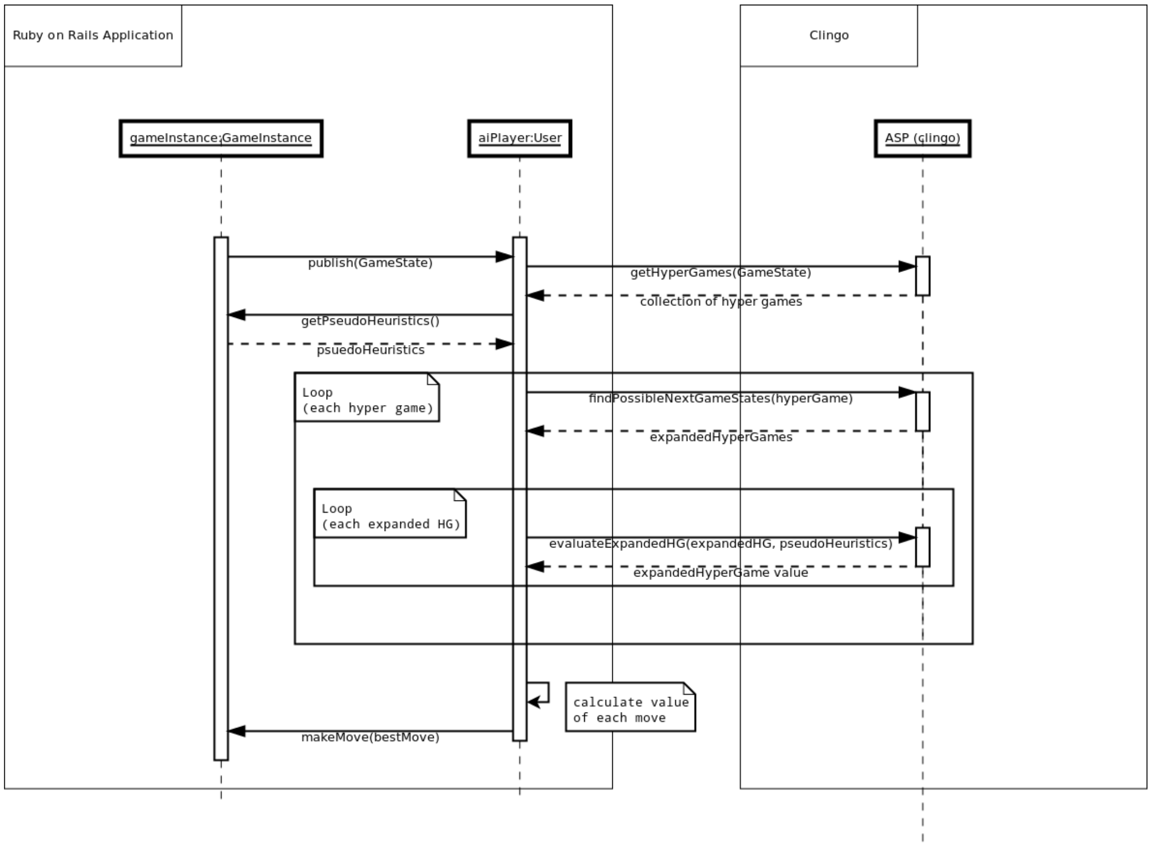


Figure 4.1: An example of the interactions between the Ruby on Rails application and ASP (clingo).

Faye

When one player makes a move the other player needs to be notified. Rails has no way of sending messages to a user unless a request has been made. To avoid the player having to poll the server we chose to use a Faye server. Faye is a publish-subscribe messaging system based on the Bayeux protocol[11].

When a player joins a game (or returns to a game) the idea is that they subscribe to the channel:

"/game_instance/" + GameInstanceId + "/" + UserId

This channel is unique to the game and user, this is because different users are allowed to see different information about the game state.

Whenever the Rails application needs to notify a user of an update it publishes a message to this channel on the Faye server. The user's browser then receives this message and runs following piece of JavaScript.

```
PrivatePub.subscribe("/game_instance/<%= @game_instance.id %>/<%= current_user.id
    %>", function(data){
  if(data.msg.variables){
    this.game_data = data.msg;
    processData(data.msg);
  }
  else {
    // message is to be published to user
    // print message to screen
  }
});
```

4.2 Choice of Game Description Language

Our requirements for a game description language were that it should be

- a machine readable representation
- concise
- suitable for interpretation in ASP
- easily translated to and from the flow charts described in section 3.2
- human readable (if possible) as this makes debugging far easier

As stated in section 2.2 the most common language for describing games is GDL. This fulfils four out of five of the requirements but it is not always easily represented as a flow chart. Therefore if we were to use this language we would need an intermediate representation. For this reason we chose not to define a new game description language which was enough to (together with some general rules) play a game using ASP but also retained enough of the structure of the flow chart to be able to reconstruct it.

4.2.1 FCDL (Flow Chart Description Language)

The flow chart is converted into ASP using the following rules.

Every element and connection of the flow chart is given a unique index when it is created.

Each element and connection becomes a predicate (connections have 5 arguments, all others have 4). The first argument of each is their index.

For connections the following 4 arguments just define which elements they connect and where on the elements it should be connected.

The available connections on an element are numbered beginning at the top clockwise from 0 to 3.

Example 4.1. connection(4, 1, 1, 2, 0) means that element 1 has a connection to element 2. It should leave element 1 on the right and join element 2 from above.

On an element the 2nd and 3rd arguments are its x and y coordinates respectively and its final argument is its caption (for a choice element the caption is its condition).

Example 4.2. terminal(1, 0, 0, "Start Game") means that a start terminal is at (0, 0).

After the game is defined some of the information about the flow chart becomes irrelevant. For example at game time we have no need for the information that the start terminal is located at (100, 200). Also for convenience it is far easier to absorb the connections into the node of the flow chart which they leave.

Definition 4.3. A *state* in an FCDL game is one of the nodes of the flow chart. In FCDL is is represented as the predicate $state(Index, StateDescription)$ where $Index$ is as before and $StateDescription$ can be any instance of the primitive states described in the following definitions.

Definition 4.4. A *start terminal* is represented as $startTerminal(Index)$ where $Index$ is the index of the (unique) element in the flow chart which it has a connection to.

Remark. Clearly a game can only have one start terminal

Definition 4.5. A *statement* is represented as $statement(Index, Statement)$ where $Statement$ is the (translated) caption of the element of the flow chart (e.g. "assign(x, 0)"). $Index$ is the index of the unique element which the statement has an outgoing connection to.

Remark. The translation of these statements is sometimes necessary to avoid syntax errors in ASP and is covered in chapter 5.

Definition 4.6. A *choice* is represented as $choice(Index1, Index2, Condition)$ where $Condition$ is the (translated) condition in the caption of the choice node in the flow chart. $Index1$ is the node which is connected from the bottom part of the choice node (The path which should be taken when $Condition$ holds) and $Index2$ is the node which is connected from the right hand side of the choice node (This path should be taken which $Condition$ does not hold).

Definition 4.7. An *end terminal* is represented by $endTerminal$ (It has no outgoing connections or important information which needs to be represented)

This conversion is done with a simple ASP program immediately after the flow chart has been entered into the system by the user.

4.3 Game State Representation

Definition 4.8. We say a game G is *on* the FCDL state $State$ if at step $StepNumber$ $stepState(StepNumber, State)$ is true.

Remark. Each step can only be on one state. The first step of every game is on the unique start terminal of the game.

Definition 4.9. We say the game G is *currently on* FCDL state $State$ if the current step (largest step for which is on a state) is on that $State$.

The flow chart node which the game is currently on is obviously only one part of the game state which needs to be represented. It does not, for example, capture which cards are currently in a particular player's hand or which card a player has just played.

Definition 4.10. The games are run in *steps*. Each step has *one* state and along with some other step facts represents the game state at a particular point in time.

Remark. The steps are indexed with a *step number*. These start from 1.

Consecutive steps must have states which are consecutive in the flow chart (There must be a connection going from the first step to the second).

Definition 4.11. A *step fact* is a fact which is made to be true by the game play which has taken place so far. These are represented as $stepFact(StepNumber, Fact)$

Remark. Step facts are persistent. Unless stated otherwise by the current step they stay the same.

Definition 4.12. The most common step fact is a *variable*. These are represented as $variable(VariableName, Value)$.

Remark. Unlike in logic programming the variable names here must begin with a lowercase letter. This is because although they are variables in the flow chart game, in the ASP program they are actually ground terms (usually constants).

Another common step fact is $playerHasCard$ which has arity 2. The first argument is the player, the second is the card expressed $card(Rank, Suit)$.

4.4 Game Control

There are many rules in ASP which control the game flow. When the ASP program is called with the rules and a previous game state it continues the game until a point where it needs input from the user or from the Ruby application (eg for a random number) or until it has computed a fixed number of steps more in the game. We stop it after a fixed number of steps to reduce the size of the grounding but the rest of this chapter will be written ignoring this fact (It doesn't change the result, just the speed).

```
% Every game starts at state 0. The UI for
% flow chart creation forces state 0 to be the unique
% start terminal for the game.
stepState(0, State) :-
  state(0, State).


% If the previous state was the start terminal
% then there is no choice about where to go next.
stepState(Step, State) :-
  validStep(Step),
  stepState(Step - 1, startTerminal(B)),
  state(B, State).


% Similarly a statement has only one outgoing
% connection, therefore the next state is easy
% to find.
stepState(Step, State) :-
  validStep(Step),
  stepState(Step - 1, statement(B, \_)),
  state(B, State).


% For a choice state things are slightly more complicated.
% If the condition held in the previous state then we follow
% one path.
stepState(Step, State) :-
  validStep(Step),
  state(B, State),
  stepState(Step - 1, choice(B, C, Condition)),
  conditionHolds(Step - 2, Condition).

% If the condition did not hold (using negation by failure) in the previous
    state
% then we take the other path.
stepState(Step, State) :-
  validStep(Step),
  state(C, State),
  stepState(Step - 1, choice(B, C, Condition)),
  not conditionHolds(Step - 2, Condition).


% If a step is not complete then we may not have enough information to correctly
% identify the next step. A step can be incomplete for several reasons
% eg a player is required to play a card but hasnt yet.
validStep(Step + 1) :-
  stepComplete(Step).
```

Figure 4.2: These are probably the most important (and simplest) of the rules in the ASP program.

Remark. For efficiency the full implementation has extra conditions on each rule in order to make the grounding of the logic program smaller. However, as they do not change the answer sets of the program, we omitted them here for simplicity.

The *stepState* predicate stores the (unique) flow chart state which the game is "on" at that step. One interesting thing to note is that we can use negation by failure to determine when the conditions are false. This is because, although the games are imperfect information, the program controlling the game has perfect information.

Step facts are only changed by statements. Variables are mainly changed by assignment, but some of the "built in" variables are changed by other statements. For example the value of "cardsRemaining(player(0))" is changed when cards are dealt or played.

The *stepComplete* predicate is used to prevent the grounding of the logic program becoming infinite. If it wasn't there then when the program was ground $stepState(Step, \_)$ would appear in the grounding for all natural numbers $Step$.

4.4.1 Expression Evaluation

When variables are assigned, it is not always to a simple expression. For example they may be assigned to the values of other variables or even some of the arguments of other variables.

Example 4.13. We may wish to create a variable *leadSuit* which stores the suit of the card played by the trick leader. This is useful when we want to check that the following players follow suit.

This would be written as $assign(leadSuit, suit(middle(leader)))$. $middle(Player)$ is a built in variable for all game players which stores the value of the last card played by the given player (unless the stack of cards has been collected).

In this situation *leader* is a variable we set earlier in the game to store the value of the trick leader.

In order to evaluate this expression we need to first evaluate *leader*, then $middle(leader)$ and finally $suit(middle(leader))$

In order to compute these evaluations we have the following rules in ASP:

```
% If Step requires an assignment to be made then we must
% evaluate the right hand side of the assignment.
% The head of this rule can be read as "evaluate Var2 at Step -1"
eval(Step - 1, Var2) :-
  stepState(Step, statement(_, assign(Var1, Var2))).

% We must also evaluate the name of the variable.
% For example if we had score(leader) = 1, and we knew that
% leader = player(0), then we should assign score(player(0)) to
% 1 rather than score(leader) as leader may change later.
evalVarName(Step - 1, Var1) :-
  stepState(Step, statement(_, assign(Var1, Var2))).

% If we know that Var is a variable with the value
% Val at the step in question then we can give this evaluation.
% The head of this rule can be read as "At Step, Var evaluates to Val"
eval(Step, Var, Val) :-
  eval(Step, Var),
  stepFact(Step, variable(Var, Val)).

% Once we have computed the above we can make the
% assignment.
stepFact(Step, variable(Expr1Val, Expr2Val)) :-
  stepState(Step, statement(_, assign(Expr1, Expr2))),
  evalVarName(Step - 1, Expr1, Expr1Val),
  eval(StepNumber - 1, Expr2, Expr2Val).
```

Figure 4.3: Some of the rules involved in evaluating expressions. There are also many rules for recursive evaluations (if we want to evaluate $score(leader)$ then we must first evaluate $leader$). Again these rules are slightly different in order to optimise the grounding.

A simple example to demonstrate why we need to evaluate the variables in the previous step is given below.

Example 4.14. Lets say we are in the game state $stepState(1, assign(x, operation(add, x, 1)))$ (to avoid syntax errors in ASP this is how we represent arithmetical expressions). Say we have the fact $stepFact(1, variable(x, 0))$.

If we evaluate $operation(add, x, 1)$ in step 1 this would give 1. But this creates the fact $stepFact(1, variable(x, 1))$. Clearly this will cause a loop! Besides which, it doesn't make sense for a variables to have two values in the same step! Hence we follow the convention that the value of variables are updated in the step *immediately after* the expression has taken place.

Example 4.15. If we have the statement $playCard(currentPlayer, card(any, any)$ and we know that the value of $currentPlayer$ is $player(0)$ then in the step after the player has laid the card we need to update the value of $middle(player(0))$ rather than $middle(currentPlayer)$ because $currentPlayer$ may change and it may not be the case that the new player it represents has the same card in the middle.

Therefore when computing an assignment we need to evaluate the variable name rather than the value of the variable which is being set.

4.4.2 How Ruby Maintains the Game State and Calls ASP

Each game instance has many steps. As mentioned previously these do not directly correspond to the steps in the FCDL game, rather they are each a collection of FCDL steps. The idea is that ASP processes the information known and progresses as far as it can through the game before it needs further input. This input could be for example that it needs to know which card a player chooses to play.

When Answer Set Programs are run using clingo they are allowed to contain lua functions. These can be as simple as basic arithmetical functions which have not been built into the language or functions for manipulating strings. Where possible we have used lua rather than Ruby. This is because it is much faster to have the lua execute during the grounding of the program than have the entire program solved, Ruby analyse the result and pass the program back to ASP to solve a second time.

However, at times it is necessary for Ruby to perform calculations. A good example of this is with random number generation. For many card games it is necessary to use random number generation, for example in trumps when deciding who will play first.

As described in section _ ASP will continue until the game has terminated or until the current step is not complete.

Example 4.16. In the case of random number generation the Answer Set will contain a fact:

$rubyRanEval(Step, random(Value))$.

Ruby is then expected to calculate a random integer X which is between 0 and $Value - 1$.

Ruby will then run the ASP program again together with the fact:

$eval(Step, random(Value), X)$.

Example 4.17. When the state requires a message to be published to the users ASP does not require any additional information but it must return to Ruby anyway in order for Ruby to process and publish the message.

If the current state requires a message to be published ASP does not mark the step as complete.

The Answer Set will contain the fact $currentFact(publish(Message))$ where $Message$ is a string.

The strings are allowed to contain variables which should be replaced by Ruby with their current value. These are specified by enclosing them with the notation $\#\{Variable\}$. To avoid the spaces being removed (as described earlier) they are replaced with $\sim$.

For example if the Answer Set contains the facts:

$currentFact(variable(trumps, hearts)).$
$currentFact(publish("Trumps \sim were \sim selected \sim as \sim \#\{trumps\}.")).$
$currentStep(CurrentStep).$

Ruby should publish the message "Trumps were selected as hearts."

Ruby does not then need to give any more information to ASP but to avoid a loop it must run the program with the additional fact $stepComplete(CurrentStep)$.

Other examples include shuffling the cards or when a player is required to make a move.

4.5 Pseudo-Heuristic Generation

Here we implement the algorithm given at the end of chapter 3.

```
1) First we need to find the main loop of the game.

2) expand the body of the main loop to
   one with no sub loops.

3) calculate all paths through this
   expanded loop body. Paths may have multiple
   instances (if x is told to play a card
   then there is an instance for if the AI
   player is x and an instance for if the other player is x).

4) Next we find any changes to the scores of the players
   which occur outside of the main loop.
   - If they are desirable (ie an increase to our score) we list the conditions
       which led to this
     as a maximise condition.
   - If they are undesirable we list their conditions as a minimise condition.

5) calculate the changes in the variables
   made along each path and the conditions
   on the initial values of variables for
   each of the paths to be taken.

6) Each instance of each path is given
   the weighting totalChangeInMaximiseProperties - totalChangeInMinimiseProperties.
```

Figure 4.4: The high level algorithm for computing the pseudo-heuristics.

The actual implementation takes a different structure to that described in the algorithm.

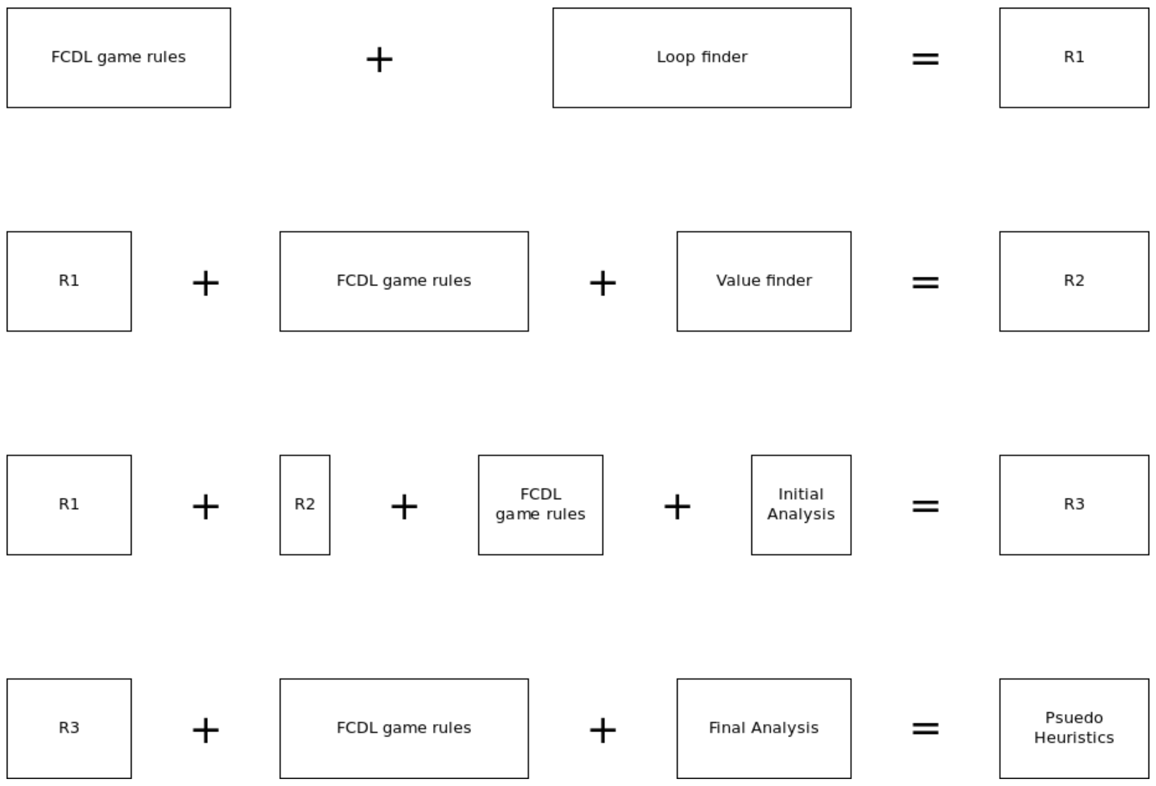


Figure 4.5: A diagram showing the stages of our pseudo-heuristic generation implementation.

Definition 4.18. A *sub-program* S of an FCDL game is a subset of the elements of the flow chart such that there is no partition $P = (S1, S2)$ of S in which no element of $S1$ is connected to an element in $S2$.

Loop finder: This breaks the flow chart into a set of loopless sub-programs. It also returns how these sub-programs fit back together to form the main flow chart.

Value finder: This calculates all of the possible paths through each of these loopless sub-programs. It also calculates the changes in variable values caused by the different paths and the conditions (on the values of the variables at the beginning of the path) required for the path to be taken.

Initial Analysis: Here we expand the body of the main loop to be an equivalent loopless program. We return each of the paths through the main loop. (This is why we needed the flow chart to be mainly finite). We also determine which properties we should be maximising/minimising.

Final Analysis: Here we again compute the change in variables along the paths of the main loop (These are different to what we calculated in the Value Finder step as the main loop is now being considered as a whole rather than as a series of sub-programs which are executed an unknown number of times). We can calculate the conditions needed to take each path along with the change in variables. We then use the change in variables which we are maximising/minimising to give the path a value. These paths are then returned with their conditions and weighting as pseudo-heuristics.

How these different steps fit together can be seen in Figure 4.5.

These problems are easier to implement in iclingo (incremental clingo). We used it to prevent the grounding of the logic program becoming too large and therefore slowing down the overall computation (this was a problem because we were analysing many possible game states at once each of which had a large possible game tree). The idea is that the logic program is solved multiple times with the answer set of the previous iteration becoming facts in the current iteration. Programs to be executed with iclingo contain a base, cumulative and a volatile section. The base is a standard ASP program which is solved once. The cumulative/volatile sections are the parts which are solved many times. They have a cumulant (to avoid confusion in our programs this is ALWAYS d) which is incremented every time the program is solved.

4.5.1 Loop Finder

```
program(Start, End, 1) :-
  state(Start, startTerminal(_)),
  state(End, endTerminal).
```

Figure 4.6: The first sub-program we consider is the entire flow chart.

The base of our incremental program contains the above rule along with the definition of a *nextState* predicate which represents all paths of length 1 in the flow chart.

```
% If two states are connected by a single path from the first
% to the second and the first one is in a sub-program P, then the second
% is accessible from the first within P.
accessible(program(Start, End, d), S1, S2) :-
  program(Start, End, d),
  S2 != Start,
  S1 != End,
  accessible(program(Start, End, d), Start, S1),
  nextState(S1, S2).

% The next state is accessible from the start of a program.
accessible(program(Start, End, d), Start, Next) :-
  nextState(Start, Next),
  program(Start, End, d).


% accessibility is transitive
accessible(program(Start, End, d), S1, S2) :-
  program(Start, End, d),
  accessible(program(Start, End, d), S1, I),
  accessible(program(Start, End, d), I, S2).
```

Figure 4.7: This determines which states are accessible within a sub-program.

```
% Loops are determined by accessibility
programContainsLoop(P) :-
  program(Start, End, d),
  P = program(Start, End, d),
  accessible(P, Start, S1),
  accessible(P, S1, S2),
  accessible(P, S2, S1),
  accessible(P, S2, End).

programLoopStart(program(Start, End, d), StartOfLoop) :-
  programContainsLoop(program(Start, End, d)),
  program(Start, End, d),
  accessible(program(Start, End, d), Start, StartOfLoop),
  accessible(program(Start, End, d), LoopElement, StartOfLoop),
  accessible(program(Start, End, d), LoopElement, End),
  accessible(program(Start, End, d), StartOfLoop, LoopElement),
  nextState(PrecedingState, StartOfLoop),
  not accessible(program(Start, End, d), LoopElement, PrecedingState).

programLoopEnd(program(Start, End, d), Prev) :-
  programContainsLoop(program(Start, End, d)),
  program(Start, End, d),
  programLoopStart(program(Start, End, d), StartOfLoop),
  nextState(Prev, EndOfLoop),
  StartOfLoop == Prev,
  not accessible(program(Start, End, d), EndOfLoop, StartOfLoop),
  accessible(program(Start, End, d), Prev, StartOfLoop).
```

Figure 4.8: These find the first loop in a sub program.

Once we have found the first loop in a program, we split the program into three sub programs. The program before the loop, the program after, and the body of the loop. We then check these for further loops.

Then the predicates returned by loop finder are *loopContainer*, *evaluate*, *programLoopless*, *programLoopStart* and *programLoopEnd*.

evaluate is an arity 1 predicate. Its argument is any loopless program. Loop container shows how any program which contains a loop can be broken up into the three programs mentioned above.

```
loopContainer(program(Start, End, d - 1), program(Start, LoopStart, d), program(
    LoopNext, LoopEnd, d), program(AfterLoop, End)) :-
  programLoopStart(program(Start, End, d - 1), LoopStart),
  programLoopEnd(program(Start, End, d - 1), LoopEnd),
  program(LoopNext, LoopEnd, d),
  program(AfterLoop, End, d),
  nextState(LoopStart, LoopNext),
  nextState(LoopEnd, AfterLoop).
```

Figure 4.9: The rule for loopContainer.

Remark. Before storing the results of loopFinder we strip out any records of the cumulant from each fact. So $program(S, E, 1)$ becomes $program(S, E)$.

4.5.2 Value Finder

Given the set of loopless programs we need to find the paths through them. We identify the paths by a string. The string is made up of a's and b's. Each letter corresponds to a choice node in the path. An a means that the condition at that choice node must be true on that path. A b, similarly, means that the condition was false.

```
nextState(program(S, E), Path, S1, Path, S2, d) :-
  S1 != E,
  currentState(program(S, E), Path, S1, d - 1),
  state(S1, statement(S2, _)).

nextState(program(S, E), Path, S1, @concat(Path, "a"), S2, d) :-
  S1 != E,
  currentState(program(S, E), Path, S1, d - 1),
  state(S1, choice(S2, _, _)).

nextState(program(S, E), Path, S1, @concat(Path, "b"), S2, d) :-
  S1 != E,
  currentState(program(S, E), Path, S1, d - 1),
  state(S1, choice(_, S2, _)).

nextState(program(S, E), Path, S1, Path, S2, d) :-
  S1 != E,
  currentState(program(S, E), Path, S1, d - 1),
  state(S1, startTerminal(S2)).
```

Figure 4.10: These are the rules used to build up the paths.

```
% An assignment statement creates an initial/updated assignment
assignment(Program, Path, State, EvalVar, EvalVal, d) :-
  eval(Val, EvalVal, path(Program, Path), State, d),
  evalVar(Var, EvalVar, path(Program, Path), State, d),
  nextState(Program, Path, S1, Path, State, d),
  state(S1, statement(State, assign(Var, Val))).

% Assignments are persistent by default
assignment(Program, Path2, State, Var, Val, d) :-
  not assignmentOverwritten(Program, Path, State, Var, d),
  nextState(Program, Path, S1, Path2, State, d),
  assignment(Program, Path, S1, Var, Val, d - 1).

% This indicates that the assignment should not persist
% at this step.
assignmentOverwritten(Program, Path2, State, Var, d) :-
  assignment(Program, Path2, State, Var, Val2, d),
  nextState(Program, Path, S1, Path2, State, d),
  assignment(Program, Path, S1, Var, Val1, d - 1),
  Val1 != Val2.
```

Figure 4.11: These are the rules for assignment. They are based on similar eval predicates to those defined for game control.

We then use the information about the assignments and paths to return $finalAssignment$ (any assignment which holds at the end of a path) and $finalCondition$ (these are the conditions with the variables replaced by their assignments at the step of the condition).

4.5.3 Initial Analysis

The basic idea here is that we identify the main loop, then transform the body of the main loop into an equivalent loop-less sub program.

```
% finalPOI is the main loop.
$ The current step is "important" if it
% is analysing the main loop.
importantCalcStep(d) :-
  finalPOI(Program, _),
  analyseProgram(Program, d).

% If the main loop contains a loop then this needs to be expanded.
% It does this using the fullLoopPath predicate
% any paths through P1, P2 and P3 could be put together to form a
% path through the main loop body.
importantPath(FullProgram, @concat(@concat(Path1, Path2), Path3), I, d) :-
  importantCalcStep(d),
  analyseProgram(P2, D),
  loopUpperBound(I, D),
  path(P1, Path1),
  fullLoopPath(Path2, P2, _),
  path(P3, Path3),
  loopContainer(FullProgram, P1, P2, P3),
  analyseProgram(FullProgram, d).

%If the main loop doesn't contain a loop
% then things are MUCH simpler. We just return
% the paths through the loop-less sub-program
importantPath(Program, Path, 0, d) :-
  importantCalcStep(d),
  not loopContainingStep(d),
  path(Program, Path),
  analyseProgram(Program, d).
```

Figure 4.12: Some of the rules needed to find the paths through the main loop of an FCDL game.

Now we have the paths through the main loop. This program also calculates the important properties which should be maximised/minimised.

```
importantProperty(max, score(me), d).
importantProperty(min, score(other), d).
```

Figure 4.13: We always know that we are trying maximise our score and minimise the score of our opponent.

```
% If we are trying to maximise one property and we know that
% it will be incremented if two things are equal, then we try to
% minimise the difference between these two properties.
%
% Note we have made the assumption here that Arg1 > Arg2
% This is fine for trick-based/shedding games, but in general
% we should build in the concept of absolute value.
importantProperty(min, EvalProperty, d + 1) :-
  evalExpr(operation(subtract, Arg1, Arg2), EvalProperty, d),
  not minProperty(**(MAX), **(Var), d),
  not knownToBeNEQ(Var, MAX, d),
  not loopStep(d),
  finalCondition(Program, Path, equal(Arg1, Arg2)),
  finalAssignment(Program, Path, **(Var), operation(add, **(Var), X)),
  not loopContainingStep(d),
  X > 0,
  analyseProgram(Program, d),
  importantProperty(max, **(MAX), d).
```

Figure 4.14: This is one of many rules which finds the important properties for maximisation/minimisation.

As stated in the comment in Figure 4.5.3 we made an assumption about the conditions we were evaluating. This condition always holds for the structure of trick based games and shedding games (In trick based games the only important property is the score). However, if we had the time we would take the absolute value after subtracting. Currently we could not do this as we have not yet built the concept of absolute value into FCDL.

4.5.4 Final Analysis

The final analysis program returns the pseudo-heuristics as several different facts. Each pseudo-heuristic has a unique id specified by the assignments of players, the path and the start/end of the sub program.

```
heuristicConjunctInitial(Condition, Weight, id(S, E, Players, Path, d)) :-
  valueOfPath(program(S, E, Players), Path, Weight, d),
  finalCondition(program(S, E, Players), Path, Condition, d).
```

Figure 4.15: This rule builds the heuristic conjuncts.

The initial conjuncts are discarded if they are easy to be proven true pre-game, eg $1 < 2$. They are broken into two if they are conjunctions.

The assignments and conditions are built up in the same way as with value finder, only this time they are built from the information of the whole path through the main loop rather than just the loopless sub-programs.

The value of the path is calculated as follows.

```
subValueOfPath(Program, Path, MaxProperty, @number(Value), d) :-
  @number(Value) != "nan",
  finalImportantProperty(max, MaxProperty),
  finalAssignment(Program, Path, MaxProperty, Value, d).

subValueOfPath(Program, Path, MinProperty, @negative(Value), d) :-
  @negative(Value) != "nan",
  finalImportantProperty(min, MinProperty),
  finalAssignment(Program, Path, MinProperty, Value, d).


valueOfPath(Program, Path, Value, d) :-
  subValueOfPath(Program, Path, _, d),
  Value = #sum[ subValueOfPath(Program, Path, Property, V, d) = V :
      finalImportantProperty(Property)].
```

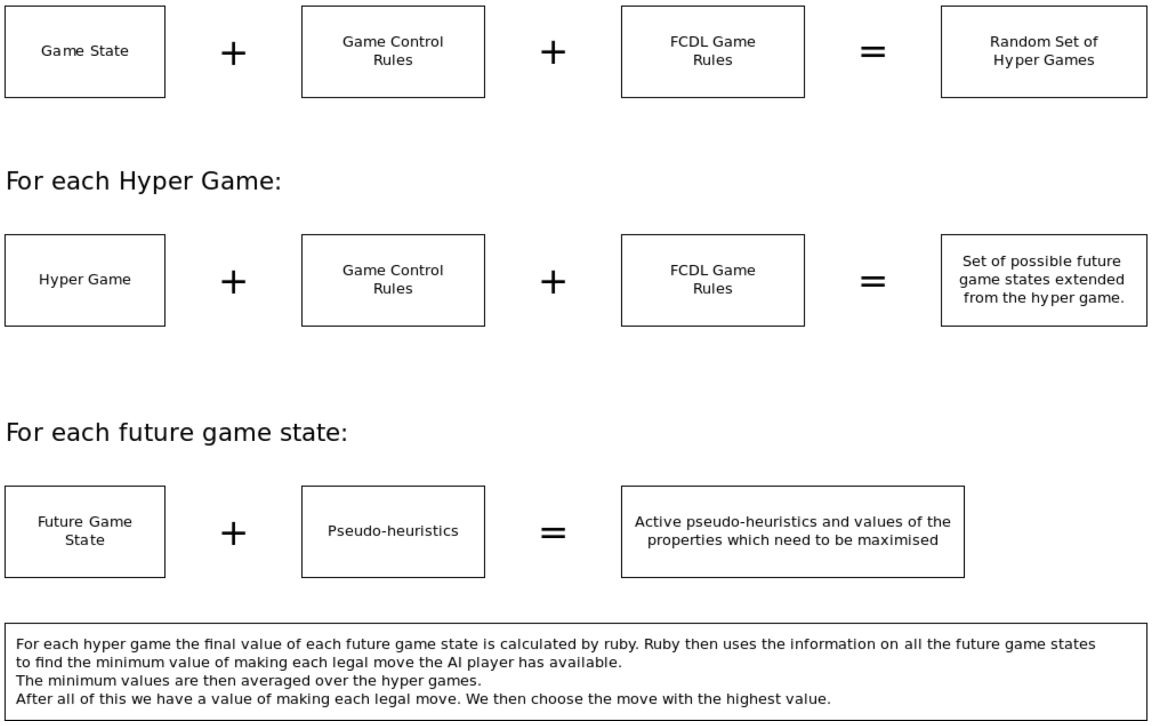
Figure 4.16: The value of the path is made up of subValues which are caused by changes to the maximal/minimal properties found in the last section.

We use the lua function @number because otherwise things which aren't numbers might appear in the grounding. This would break the sum (as clingo can't sum things which aren't integers).

4.6 Hyper Play in ASP

When finding the hyper games we have imperfect information about the game state. Some of the step facts are given to the players, for example who laid a particular card when, how many cards each player has remaining and the value of anything declared as a variable. However, some of the step facts such as *playerHasCard* are hidden from the player unless they are the player the predicate is about.

However, as we know how many cards a player has left it is easy to represent this partial information in ASP.


Game State
+
Game Control Rules
+
FCDL Game Rules
=
Random Set of Hyper Games
For each Hyper Game:
Hyper Game
+
Game Control Rules
+
FCDL Game Rules
=
Set of possible future game states extended from the hyper game.
For each future game state:
Future Game State
+
Pseudo-heuristics
=
Active pseudo-heuristics and values of the properties which need to be maximised

For each hyper game the final value of each future game state is calculated by ruby. Ruby then uses the information on all the future game states to find the minimum value of making each legal move the AI player has available.
The minimum values are then averaged over the hyper games.
After all of this we have a value of making each legal move. We then choose the move with the highest value.

Figure 4.17: A diagram showing how the different parts of out game time AI implementation fit together.

```
% This rule specifies that if N cards are dealt to a player, then the player
% should have N cards in their hand in that step. (We assume cards are dealt once
% per game). There are similar rules for when a player picks up cards in a shedding
    game.
N { stepFact(Step, playerHasCard(P, card(R, S))) : card(R, S) } N :-
  stepAction(Step, deal(N)),
  player(P).
```

Figure 4.18: This rule “generates” many answer sets in which the player has been dealt different cards.

This one rule is enough to generate many models of the game. However, not all of them will be possible given the information we have. It could for example give a model where both $player(0)$ and $player(1)$ have the six of clubs.

This is handled by a simple constraint:

```
:- stepFact(Step, playerHasCard(Player1, Card)),
   stepFact(Step, playerHasCard(Player2, Card)),
   Player1 != Player2.
```

Figure 4.19: No two different players should have the same card in their hand at the same step.

The following rules ensure the persistence of a player's hand:

```
% Unless cards have been dealt or picked up at Step the player cannot have
% a card in their hand which they did not have before
stepFact(Step - 1, playerHasCard(Player, Card)) :-
  Step > 0,
  stepFact(Step, playerHasCard(Player, Card)),
  not stepAddsCards(Step).

stepAddsCards(Step) :-
  stepAction(Step, deal(CardsDealt)).

% If at Step a player does not play a particular card which was
% in their hand at (Step - 1) then they should still have that card
% at Step.
stepFact(Step, playerHasCard(Player, Card)) :-
  stepFact(Step - 1, playerHasCard(Player, Card)),
  not stepFact(Step, playerLaidCard(Player, Card)).

% If a player laid a card at Step, then it must have been in their
% hand at (Step - 1)
stepFact(Step - 1, playerHasCard(Player, Card)) :-
  stepFact(Step, playerLaidCard(Player, Card)).
```

Figure 4.20: The rules to ensure persistence of a player's hand.

This now produces a hyper game which is consistent with the cards a player has played and which doesn't allow players to share cards or have the wrong number of cards. However, we still haven't checked that the correct path was followed through the flow chart. As the values of the game variables are known to us at every step and we know the path the game took, all we need to do is check the values of the conditions at the choice nodes (there is only a maximum of one path from any other node type).

We use the same rules as those used in game control to evaluate the conditions given the known information/ information from the hyper game.

We then use the following two constraints to eliminate answer sets where the correct path has not been followed (these are impossible hyper games).

```
:- not stepState(Step + 1, State),
   state(B, State),
   stepState(Step, choice(B, C, Condition)),
   conditionHolds(Step - 1, Condition).

:- not stepState(Step + 1, State),
   state(C, State),
   stepState(Step, choice(B, C, Condition)),
   not conditionHolds(Step - 1, Condition).
```

Figure 4.21: These two constraints ensure that the wrong path could not have been followed through the flow chart given the data we have about the game.

Any answer set of this program (combined with all the rules from game control) is a possible hyper game! This is the part of the project where the decision to use ASP really payed off. If we wanted we could run this and get ALL possible hyper-games. There is a one-to-one mapping between the answer sets of the program and the possible hyper-games. However, in practice there are far too many answer sets to consider all of them. Using clingo it is possible (although details of this were not easy to find) to ask for a specific number of randomly selected answer sets. This is perfect for our needs.

In order to do this we run the command:

```
clingo -n {Number of Answer Sets} {Filenames} --restart-on-model --rand-prob=yes --
    reduce-on-restart --rand-watches=yes --rand-freq=1 --seed={random seed}
```

4.7 Calculating the Heuristic Value of a Hyper Game

Now that we have seen how to calculate a hyper game, as we already have the pseudo-heuristics, all that remains is to show how we implemented the calculation of a value of a hyper game.

We covered this at a high level in chapter 3. There were two steps:

1. Find all possible game states at the start of the next trick given that we started at the hyper game given.

2. Calculate the number of pseudo-heuristics which are active / semi-active for each card.

Remark. The calculation of the final value is actually done by Ruby using the equation given in section 3.4.2, but is based solely on the information found from the answer sets. We could not do the final calculation in ASP as clingo only supports integer arithmetic.

4.7.1 Finding the Possible Game States

As we intend to group the game-states by the move the AI player has made and we already know all legal moves for the AI player it is easier to include this information as a fact in the logic program (for each move).

```
stopStep(Step) :-
  Step > Min,
  minStep(Min),
  stopState(State),
  stepState(Step, statement(State, _)).
```

Figure 4.22: The above clause is one of the rules which define the *stopStep* predicate. This is to determine when the program should stop. *StopState* is given as a fact in the program and its argument is the state where the heuristic is to be evaluated (the start of the main loop). There are similar rules for the other flow chart elements.

```
% A player plays EXACTLY one card if the step action requires them to.
1 { stepFact(Step, playerLaidCard(Player, card(Rank, Suit))) : playerHasCard(Player,
     card(Rank, Suit)) } 1 :-
   stepPredictAction(Step, playCard(Player, CardTemplate)).

stepPredictAction(Step, playCard(Player, CardTemplate)) :-
  evalStep(Step),
  eval(Step - 1, CardTemplateVar, CardTemplate),
  eval(Step - 1, PlayerVar, Player),
  stepState(Step, statement(_, playCard(PlayerVar, CardTemplateVar))).
```

Figure 4.23: These rules are here to “generate” the answer sets where the player makes each legal move.

```
:- stepFact(Step, playerLaidCard(Player, Card)),
   not stepFact(Step - 1, playerHasCard(Player, Card)),
   playerHasCard(Player, Card),
   stepPredictAction(Step, playCard(Player, CardTemplate)).

:- stepFact(Step, playerLaidCard(Player, Card)),
   stepFact(Step - 1, playerHasCard(Player, Card)),
   playerHasCard(Player, Card),
   not cardInstanceOf(Card, CardTemplate),
   stepPredictAction(Step, playCard(Player, CardTemplate)).
```

Figure 4.24: The above constraints are to remove those answer sets where a player played a card which was not a legal move at the time they played it.

The rules described are the main rules in generating the possible game states. When combined with a few more rules and with the main logic program for game control the answer sets of the program are each a possible game state given the original hyper game.

4.7.2 Calculating Active Pseudo-Heuristics

Each instance of each pseudo-heuristic is converted to a rule (with an ID in its head).

```
heur(961, card(10,hearts), 20) :-
  conditionHolds(equal(rank(card(king,hearts)),king)),
  conditionHolds(equal(rank(card(10,hearts)),king)),
  conditionHolds(lessThan(rankValue(card(10,hearts)),rankValue(card(king,hearts))))
      ,
  conditionHolds(playerHas(player(0),card(any,suit(card(10,hearts))))),
  conditionHolds(cardInstanceOf(card(10,hearts),card(any,any))),
  conditionHolds(cardInstanceOf(card(king,hearts),card(any,suit(card(10,hearts)))))
      .

calc(equal(rank(card(king,hearts)),king)).
calc(equal(rank(card(10,hearts)),king)).
calc(lessThan(rankValue(card(10,hearts)),rankValue(card(king,hearts)))).
calc(playerHas(player(0),card(any,suit(card(10,hearts))))).
calc(cardInstanceOf(card(10,hearts),card(any,any))).
calc(cardInstanceOf(card(king,hearts),card(any,suit(card(10,hearts))))).
```

Figure 4.25: An ASP representation of the rule for determining whether a pseudo heuristic is *active*. The calc facts are there so the program evaluates those particular conditions. This example is taken from the game of *Kings and Jacks*.

Similarly the rules for determining the count of a semi-active heuristic are computed.

```
heur(7, card(8,clubs), 20, Count) :-
  conditionHolds(equal(rank(card(ace,clubs)),king)),
  conditionHolds(equal(rank(card(8,clubs)),king)),
  conditionHolds(equal(follower,0)),
  conditionHolds(cardInstanceOf(card(8,clubs),card(any,any))),
  conditionHolds(cardInstanceOf(card(ace,clubs),card(any,any))),
  evalCount(player(1),card(any,suit(card(ace,clubs))),Count).

evalCount(player(1),card(any,suit(card(ace,clubs)))).
```

Figure 4.26: An example of the rules required to compute whether a semi-active heuristic is true and if so what the count is.

```
value(Card, X, 0) :- X = [ heur(ID, Card, W) = W ], myCard(Card).
value(Card, X, Count) :- X = [ heur(ID, Card, W, Count) = W ], heurCounter(Card,
    Count).


activeHeuristics(Card, X, 0) :- X = [ heur(ID, Card, W) ], myCard(Card).
activeHeuristics(Card, X, Count) :- X = [ heur(ID, Card, W, Count) ], heurCounter(
    Card, Count).

heurCounter(Card, Count) :- heur(_, Card, _, Count).


% This rule is used to evaluate the count for a semi-active heuristic
evalCount(Player, CardTemplate, Count) :-
  evalCount(Player, CardTemplate),
  Count = #count{ cardInstanceOf(Card, CardTemplate) : playerHasCard(Player, Card)
       }.
```

Figure 4.27: The main rules used to compute the information necessary to give a game state a value.

4.8 Optimisation of the Logic Programs

A constant struggle throughout the project was to keep the time taken to run the logic programs at an acceptable speed given that the games are meant to be played against human users.

We won't go into too much detail in this section of how this was actually achieved for each rule. However, we thought it was worth noting some of the techniques we used.

Firstly we “turned off” all evaluation for any step which had already been computed by a previous run of ASP. This was achieved by having a predicate which indicated whether a step should be evaluated further and including it as a condition for any evaluation rule.

We also had a problem when the ground program contained rules for steps a long way in the future. As the game state of those steps was not yet determined there were many possibilities for which rules may be applicable and therefore the grounding became very big (this slows down the computation). For this reason we introduced a *maxStep* predicate. If the computation reaches this step then ASP should not go any further before “returning” to Ruby. Ruby then increases this *maxStep* and calls ASP again. While it may seem counter-intuitive to increase speed by solving two logic programs instead of one, the grounding of the program is reduced so much that these two logic programs can be solved much faster than one. We could have used iclingo for this; however, by the time we realised that this might be necessary it was too much of a change to make.

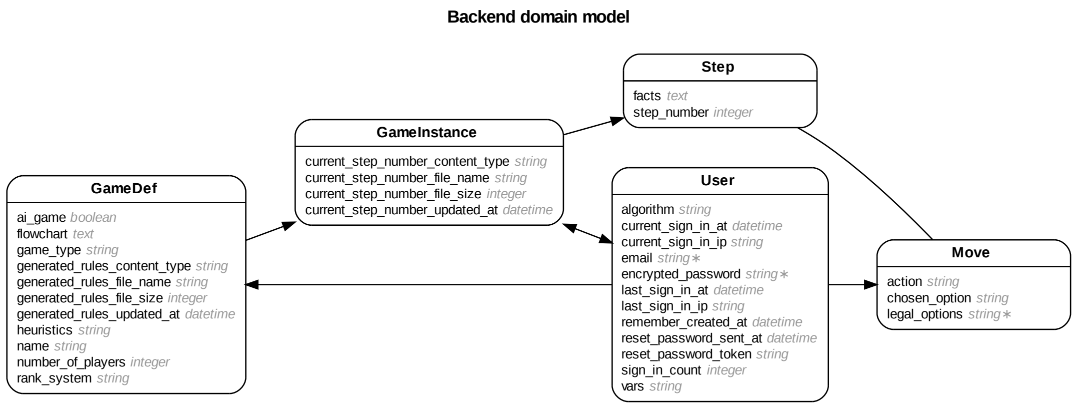
We also tried to remove any rules from the grounding where we knew the body could not possibly be true but the grounder had not picked it up. We did this by adding further conditions to the rules.

4.9 Ruby on Rails Application Overview

Rails applications are based on the model view controller pattern. The models are the data of the application and the rules with which to manipulate the data. The views are the user interface (web pages). The controllers are responsible for procession requests from the views computing a response based on the data stored in the models and passing this back to the views. [12]

4.9.1 Models

The Rails application uses a sqlite3 database, the standard for rails applications, the data model of which is shown in the diagram below.


Backend domain model
Step
facts text
step_number integer
GameInstance
current_step_number_content_type string
current_step_number_file_name string
current_step_number_file_size integer
current_step_number_updated_at datetime
GameDef
ai_game boolean
flowchart text
game_type string
generated_rules_content_type string
generated_rules_file_name string
generated_rules_file_size integer
generated_rules_updated_at datetime
heuristics string
name string
number_of_players integer
rank_system string
User
algorithm string
current_sign_in_at datetime
current_sign_in_ip string
email string ∗
encrypted_password string ∗
last_sign_in_at datetime
last_sign_in_ip string
remember_created_at datetime
reset_password_sent_at datetime
reset_password_token string
sign_in_count integer
vars string
Move
action string
chosen_option string
legal_options string ∗

The most important models to consider are game definitions named *GameDefs*, game instances and users.

Most of the user model was generated by a "gem" (Ruby plug-in) called *devise*. Devise also builds in the standard sign up/authentication controllers and views. A user can have many game definitions (although these are public, so other players can play the game).

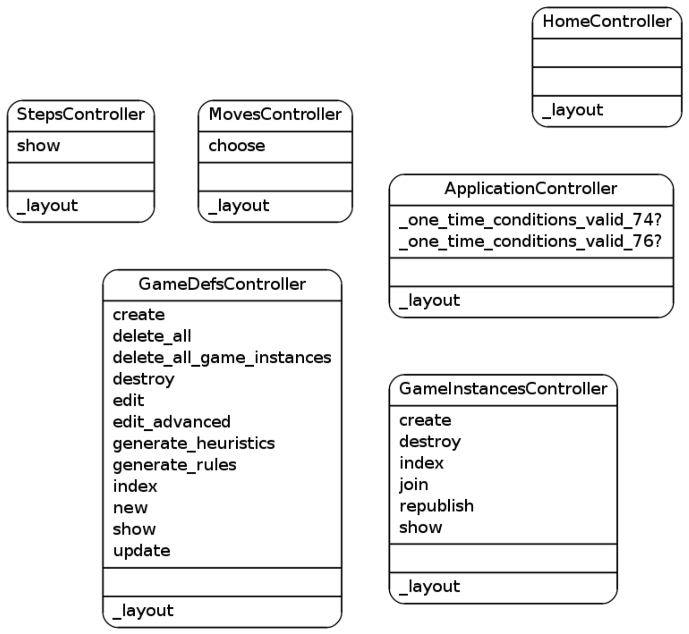
A game definition has many instances. These are the individual games. Clearly a game instance must have many users (these are the players). Equally a user must have many game instances (these are the games which they are playing or have played in the past).

The other two models listed are parts of the game instances. A game instance has many steps. These do not directly correspond to the individual game steps and are fully described in the next section.

A step might have a single move. This move belongs to a user (who can have many moves). The user it belongs to is the one who is required to make a move.

For simplicity AI players are not shown in the diagram, they inherit from the user class but are stored in the database in exactly the same way. This is why the user model has a field called “algorithm”.

4.9.2 Controllers



Most of these controllers are fairly standard and were generated by the rails framework.

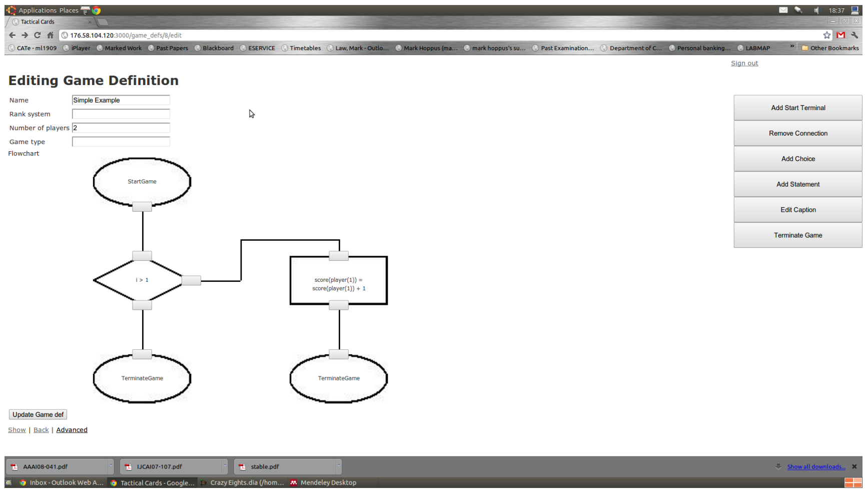
For a game instance the additional controllers are “join” and “republish”. “republish” is there to allow users to leave a game and come back. They do not want to join the game as they are already a part of it, they just need all the information about the game resending. This means that users can leave a game and come back later (with it in exactly the same state).

A move has the “choose” controller (this can only be used by the correct user).

For a game definition we added quite a few controllers, many of which were for debugging purposes. “delete_all” deletes all game definitions created by the current user. “delete_all_game_instances” on the other hand deletes all game instances of the current game definition. “edit_advanced” brings up the ASP facts generated by JavaScript allowing the user to manually edit the flow chart. “generate_heuristics” and “generate_rules” are done once upon creation of the flow chart but (as the later is quite computationally intensive) the user must call them manually if they change the definition of the flow chart.

4.9.3 Views

The front end of our application was written in HTML and JavaScript. The most important two pages are those responsible for flow chart editing (see Figure 4.28) and game play (see Figure 4.29).


Editing Game Definition

Name: Simple Example
Rank system:
Number of players: 2
Game type:
Flowchart

Figure 4.28: A screen shot showing the edit screen for a simple flow chart (this is not a real game).

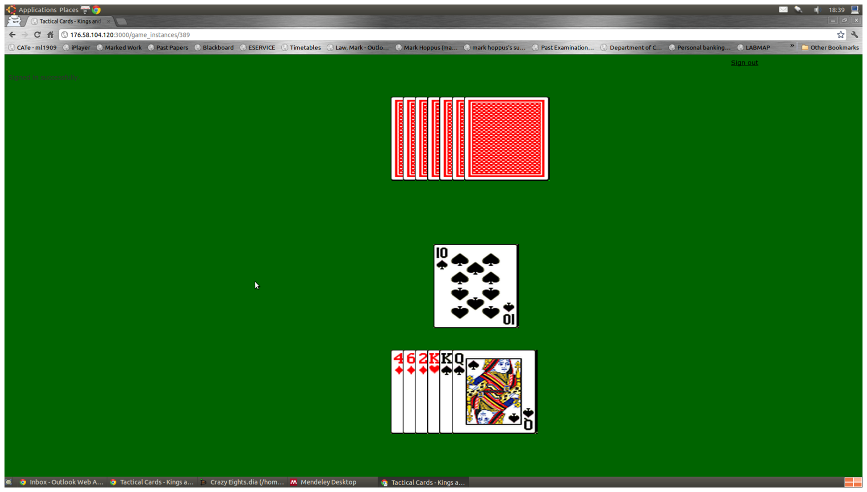


Figure 4.29: A screen shot from a card game (Kings and Jacks) being played on our system.

JSON

When the user reloads the page, after rejoining the channel, they make an asynchronous request for the last game state to be republished to them. This enables players to leave the game at any point and reload it later.

We chose to use JSON to represent the data being broadcast. This is because it is well supported by rails - we can easily add Ruby objects to the message which are converted to JSON.

We send two different types of JSON object. The simplest is the one sent when the current FCDL state requires a message to be published.

This message takes the form:

```
{"messages":["hearts~has~been~chosen~as~trumps."]}
```

And consists of a list of *all* messages which have been published in the game so far. This is so that if an observer comes along late they will still have a record of all messages broadcast so far.

The more common (and slightly more complex) JSON object we send is used to describe the current game state.

Example 4.19. This is a JSON object which is sent during the game of Trumps.

```
{
  "variables":{
    "score(player(0))":"1",
    "cardsRemaining(player(1))":"6",
    "trickWinner":"player(0)",
    "leader":"0",
    "numberOfCards":"7",
    "numberOfPlayers":"2",
    "score(player(1))":"0",
    "currentPackIndex":"14",
    "trumps":"diamonds",
    "currentPlayer":"player(1)",
    "passed(player(1))":"false",
    "middle(player(1))":"empty",
    "passed(player(0))":"false",
    "middle(player(0))":"card(3,diamonds)",
    "suit(3)":"spades",
    "suit(2)":"hearts",
    "suit(1)":"diamonds",
    "suit(0)":"clubs",
    "cardsRemaining(player(0))":"5",
    "leadSuit":"diamonds","i":"1","me":1
  },
  "game_history":[
    "stepState(0,startTerminal(1))",
    "stepState(1,statement(3,\"Shuffle\"))",
    "stepState(2,statement(5,\"Deal(7)\"))",
    "stepAction(2,deal(7))",
    "stepState(3,statement(7,assign(leader,random(numberOfPlayers))))",
    "stepState(4,statement(70,assign(trumps,suit(random(4)))))",
    "stepState(5,statement(9,publish(\"#{trumps}~has~been~chosen~as~trumps.\")))",
    "stepAction(8,playCard(player(leader),card(any,any)))",
    // We have cut parts of the game history to reduce the size for the example.
    "stepAction(30,playCard(currentPlayer,card(any,leadSuit)))",
    "stepState(24,statement(17,assign(trickWinner,player(leader))))",
    "stepState(25,statement(19,assign(leadSuit,suit(middle(player(leader))))))",
    "stepState(26,statement(21,assign(i,1)))","stepState(27,choice(23,49,lessThan(
        i,numberOfPlayers)))",
    "stepState(28,statement(25,assign(currentPlayer,player(operation(modulo,
        operation(add,leader,i),numberOfPlayers)))))",
    "stepState(29,choice(27,36,playerHas(currentPlayer,card(any,leadSuit))))",
    "stepState(30,statement(29,\"playCard(currentPlayer,card(any,leadSuit))\"))"],
    "step_number":30,
  "state":"statement(29,\"playCard(currentPlayer,card(any,leadSuit))\")",
  "hand":[
    {"rank":"9","suit":"spades"},
    {"rank":"8","suit":"diamonds"},
    {"rank":"10","suit":"clubs"},
    {"rank":"5","suit":"spades"},
    {"rank":"queen","suit":"diamonds"},
    {"rank":"ace","suit":"clubs"}],
  "move":{
    "action":"playCard(any,diamonds))",
    "chosen_option":null,
    "created_at":"2013-06-11T11:30:45Z",
      "id":962,
      "legal_options":" Players[1].playCard(8,diamonds) Players[1].playCard(queen,
         diamonds)",
      "step_id":2658,
      "updated_at":"2013-06-11T11:30:45Z",
      "user_id":2
   }
 }
```

This is all the information needed to completely represent the game state. We also decided to include the game history (the list of game states which have been taken so far). This was mainly to be helpful for debugging the FCDL games.

If the object contains a move, then the player is required to choose one of the *legal options* listed in the move. It sends this decision using the update controller for that move (asynchronously).

The variables are not a complete list. Any that give away the hidden information of the game (the other player's hand etc) are not included. $middle(player(X))$ represents the card last played by X (this will be empty if the cards have been collected since).

5 Evaluation

5.1 Results

The first path of the results of our system is the output from the pseudo-heuristic algorithm. However, this output is very large and so we decided that it fits better in an appendix (see section 8). Our implementation of the algorithm works for each of the 3 tested games. The pseudo-heuristics are exactly as expected.

We also show in section 7 some examples of our player making "sensible" decisions.

For trick based games we have implemented the AI player at game time. The results of our player playing against other AI players (a random and a greedy player) are given here.

We experimented with how our AI player performed using different numbers of hyper games. We found that when playing a random player it made little difference for the game of trumps how many hyper games we used. This is reflected in Figure 5.1. What we can see from this graph is that, although the mean number of games won remains largely the same, the standard deviation seems to decrease slightly as we up the number of hyper games. By 20 hyper games it is fairly settled on around 75%.

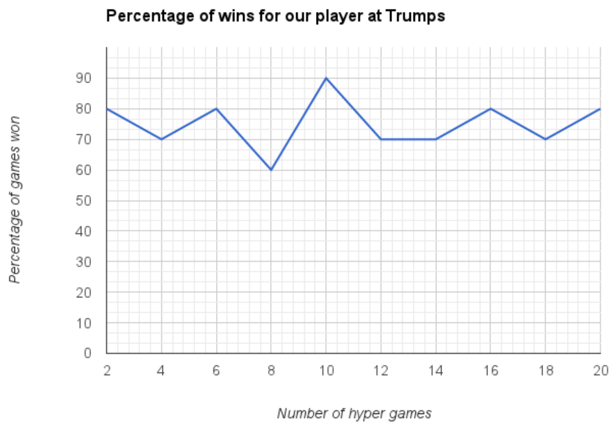


Figure 5.1: The graph of our player playing against a random player in the game of Trumps.

For the game of Kings and Jacks, in the case of a draw, we recorded half a win to both players.

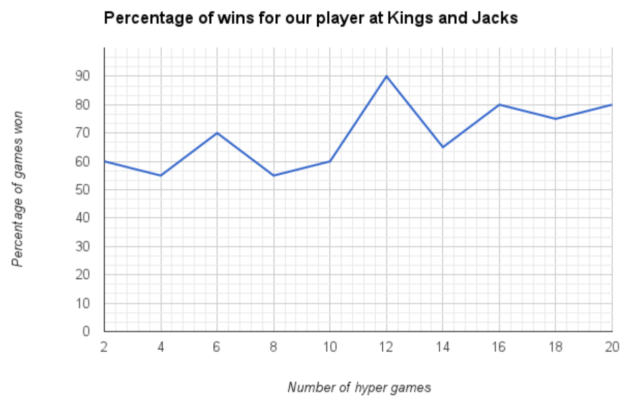


Figure 5.2: The graph of our player playing against a random player in the game of Kings and Jacks.

Figure 5.2 shows a far more erratic picture, with our AI not too much better than a random player, for a low number of hyper games. We believe that this is because making a false assumption (for example, the other player doesn't have the Jack of hearts) can be far more costly a mistake than some of the false assumptions in the game of trumps. Hence as the number of hyper games is increased, and the player has a better picture of the possibilities, we see that the performance settles down to around 80%.

We also tested our player against a "greedy" AI player. This is a player who only focuses on their score at the end of the current trick. We implemented the greedy player simply by "switching off" the pseudo-heuristic evaluation part of our player. This means that the greedy player still uses hyper play, so the comparison really is to see whether our pseudo-heuristics are benificial. We found that over a sample of 40 games for Kings and Jacks our player won 61.25% (again a draw counts as half a win) and for trumps our player won 70% of the games.

This shows that there is some substance to the idea of our approach. We believe that, had we better modelled the other player's decisions, we could expect even better results.

Unfortunately for shedding games we have not yet managed to implement the hyper play part of the AI. At the time of writing we have a bug in the part of the program which generates game models which include a player picking up cards. We do not believe that there is very much left to do once this issue has been resolved. The generation of the game for users to play and the pseudo-heuristic generation all works. The pseudo-heuristics for a simplified version of the game of Crazy Eights can be found in section 8.

5.2 Related Work

As highlighted in the background section, most of the work we could find on General Game Playing was focused on perfect information games[2][7][13]. There is some research on General Game Players with imperfect information; however, these focus on all games which are definable in GDL II. In short due to the exact specification of the problem we set ourselves, there is no research focused on exactly the same set of games. For this reason it would be unfair to do a direct comparison of performance of the Game Players. However, we can discuss how our work differs from, or is similar to, other work.

We were clearly hugely influenced by the work of the authors of the Hyper Play method[1]. In their paper on Hyper Play they do not specify how they found their set of Hyper Games. We chose to implement this side of the project using Answer Set Programming. This was well suited to the task and we were able to use clingo to find a random selection of the Hyper Games.

In our pseudo-heuristic generation, part of our algorithm finds properties which our algorithm would like to maximise/minimise as the game progresses. In our examples these were the score or the number of cards remaining. These are just like the features described in [4]. As we knew the games we were playing were repetitive we were able to consider paths through the main loop of the game. In games with no such repetition (or games made of a series of repetitive steps) we would need to adapt this technique. A similar technique would likely still be applicable but we would have to plan which of our cards (or other assets) were most valuable in which parts of the game.

Many techniques for heuristic generation that we found made use of machine learning [4][6][7]. Our technique instead reasoned about the paths through the game. As mentioned previously this technique may not directly apply to General Game Playing; however, we believe it could be adapted. Our results show that there is some substance to the approach when applied to card games. It would be very interesting to see if the adapted approach for general games written in GDL-II[8] had the same success, and if this success was comparable to the machine learning approaches of finding a heuristic.

6 Conclusion

As summarised in the introduction, our major accomplishments in this project were:

1. A language for representing the structure and rules of general card games which can be easily translated into a flow chart but is also easy to analyse when performing (pseudo) heuristic generation.

2. A method for (pseudo) heuristic generation for a subset of general card games.

3. A working implementation of the heuristic generation using Answer Set Programming.

4. A working implementation of the hyper play technique again utilising ASP.

5. A working Ruby on Rails server which can be used both to create games and also to play games either against another player (online), or against an AI player which has been generated by the system.

We have shown that the idea of using paths through the main loop of a repetitive card game is a viable approach to generating (pseudo) heuristics. We have implemented and tested this for a subset of card games and believe that this result should also apply to general games if the implementation were adapted sufficiently.

We have also shown that card games can be well represented by a flow chart. We gave a method for converting this flow chart into a language which is ready for the analysis necessary to perform the pseudo-heuristic generation.

In focusing on card games we had an advantage over other General Game Players. We were able to define some rules which are only applicable to card games. For example that the cards in a player's hand persist by default. However, this did not provide an advantage to the AI side of the project. If the game were written in GDL the rules we specified as part of our game control "library" would have had to be specified in the rules of the game. The main advantage to our project was actually that it simplified the flow charts which needed to be input by the user.

Representing General Games by a flow chart may not be of interest to AI researchers as they are likely to be good programmers and realise that it is easier for *them* to implement the games in a logic based programming language like GDL. However, if research on General Game Playing is successful then it is conceivable that users far less experienced in computing may have to use the tools which use the techniques.

In its current form the UI of this project is probably not user friendly enough to be classified as easy to use. However, we believe that we have shown that flow charts are a possible way for a less experienced user to implement the game (or other task) that they want the AI to play (or perform).

6.1 Future Work

Clearly the first bit of future work for this project would be to find the bug in the hyper play implementation for shedding games. We do not believe that this would take very long.

If this project were to be continued we see two distinct paths which could be followed. The first would be to work towards the goal of releasing the application as a product. Our preferred path would be to continue the research on the AI side of the project.

Before continuing with extra features we would like to test what we have already implemented on many more games. So far our implementation has only been tested on a handful of games and, while we are confident that the algorithm works, our implementation may have bugs which could be uncovered and fixed by further testing.

6.1.1 Future Work as a Product

If the project were being released as a product there are several issues which would need to be addressed:

1. The flow chart user interface needs work to make it more user friendly.
2. The flow charts are not tested for "syntax errors" before they are translated to FCDL.
3. There is currently no way of debugging the games.
4. Some games are slow at game time.

Improving the UI would not take all that much work, but it was not the main focus of the project, hence it being neglected. Similarly testing the flow charts for "syntax errors" is quite easy. The system would need to check for the obvious errors (illegal/missing connections) and for actual syntax errors in the captions of the elements of the flow chart.

We have some ideas of how debugging the games could work. We could allow a game creator to start a game in debug mode. This would allow them to see all of the cards (ie a "face up" version of the game). They would be able to switch between views of being each player (and could also make moves for them). A big help in debugging the games would be to know the value of each variable in any state. This would all be very easy to implement.

Some games being slow at game time is in part due to the hardware we have available, if we were releasing this as a product we would have far greater resources. However, there are further ways of optimising the logic programs and given a week or two we are sure that we could have all games running at an acceptable speed (even with our current hardware).

6.1.2 Future Work as a Research Project

If we were continuing research on the AI side of the project there are many interesting areas which we could explore. Our personal preferences would be to focus on:

1. Better approximating the probability of each hyper-game.
2. Extending the types of card games which can be played (trying 4 player games, or collection games)
3. Applying what we have learnt about General Card Game Playing to General Game Playing with Imperfect Information.

When approximating the probability of each hyper game we currently use the approach put forward by the authors of the hyper play technique[1]. This is to assume that the other player is equally likely to choose each possible move. In the previous section we showed that our result can in some cases be improved considerably by better modelling the other player's decisions. It would be very interesting to investigate how we could efficiently implement this modelling in general.

Every game we have seen in this report has been a two player, constant sum game (and each player can be in either position at a particular trick). We relied heavily on this when generating the pseudo heuristics (all of our pseudo-heuristics have a corresponding symmetric pseudo-heuristic). Many card games require four players and therefore it would be interesting to see if our methods would extend to four player card games. Another possible extension is to try a different style of card game (eg games where a player must collect a set of cards with a particular pattern), however we don't think that this would add too much as it is only another category of game.

Our technique for heuristic generation relies on first detecting the main loop of the game and then working out the value of the paths through it. We see no reason why this cannot be applied to all repetitive games. It would be interesting to try this technique against other General Game Players. Using ASP to find the set of games needed for hyper play should also work in general. Given another few years on this project our aim would be to make a General Game Player using the techniques of this project that would (hopefully) be capable of competing with other General Game Players with the possibility of entering in the annual tournament[2]!

7 Appendix A: Examples of how our Algorithm Makes Decisions at Game Time

In this section we show some examples of where our algorithm makes what we consider to be a sensible decision. We chose the game of Kings and Jacks for this.

RANDOM_AI is the other player (It will choose randomly between it's legal options) The purpose of these examples is not to show that we can beat a random player, it is to show the reasoning behind the decision making process. We will not show the active heuristic instances as there are far too many, we will simply comment on the value found for each move.

HYPER-HEUR is our AI player.

Example 7.1. The following is an example from the game of Kings and Jacks.

The aim is not to take any trick containing a King or a Jack.

```
(RANDOM_AI): choosing option  8 of hearts
```

We can see that the other player has played the 8 of hearts, our only legal options here are to play the 9 or the Queen of hearts.

A simple "greedy" AI, only focused on maximising their score and minimising the other player's score would see no difference in the two options. Either way the trick is worth 0 to both players.

However, our player *does* see a difference. This is due to the fact that in 1 of the models below the other player has the Jack of hearts. This means there is the potential that in a future trick our player could take Jack of hearts with the Queen of hearts. This would lose our player 10 points. The 9 of hearts could not take this same future trick. Therefore our player takes the opportunity to play the Queen of hearts now, when they know it is safe to.

```
potential model: ["card(7,clubs)", "card(7,diamonds)", "card(5,hearts)"]
potential model: ["card(2,clubs)", "card(king,hearts)", "card(3,spades)"]
potential model: ["card(jack,diamonds)", "card(queen,diamonds)", "card(4,spades)"]
potential model: ["card(7,diamonds)", "card(10,diamonds)", "card(2,spades)"]
potential model: ["card(4,diamonds)", "card(6,hearts)", "card(7,spades)"]
potential model: ["card(ace,hearts)", "card(2,hearts)", "card(4,spades)"]
potential model: ["card(king,clubs)", "card(4,clubs)", "card(6,spades)"]
potential model: ["card(9,clubs)", "card(3,diamonds)", "card(4,spades)"]
potential model: ["card(king,diamonds)", "card(jack,hearts)", "card(5,hearts)"]
potential model: ["card(3,diamonds)", "card(4,diamonds)", "card(10,diamonds)"]
potential model: ["card(7,diamonds)", "card(8,diamonds)", "card(2,spades)"]
potential model: ["card(ace,clubs)", "card(10,diamonds)", "card(ace,hearts)"]
potential model: ["card(6,clubs)", "card(7,clubs)", "card(8,diamonds)"]
potential model: ["card(4,clubs)", "card(9,clubs)", "card(2,spades)"]
potential model: ["card(10,diamonds)", "card(king,hearts)", "card(6,spades)"]
potential model: ["card(3,clubs)", "card(2,hearts)", "card(4,hearts)"]
potential model: ["card(4,clubs)", "card(5,hearts)", "card(jack,spades)"]
potential model: ["card(jack,diamonds)", "card(3,diamonds)", "card(jack,spades)"]
```

Figure 7.1: The hands of our opponent in hyper games found by our system which are consistent with the game so far.

The values of playing each card are shown below. These are the total values of the game state after making the moves.

```
card(9,hearts):  -0.08040474133988484
card(queen,hearts):  0.44247107565357924
```

Example 7.2. The following example (again in Kings and Jacks) show that the player "recognises" that there is no difference between some moves.

```
  (RANDOM_AI): choosing option  6 of clubs
```

The other player has played the 6 of clubs. Our player could play the 7 or the 8. As these cards are consecutive there are no cards which can go "between" our two cards. Therefore any trick which we would take with the 8, we would also take with the 7 (and similarly the other way around).

```
potential model: ["card(8,spades)", "card(8,hearts)", "card(7,hearts)", "card(3,
    hearts)"]
potential model: ["card(8,spades)", "card(10,diamonds)", "card(3,clubs)", "card(
    queen,clubs)"]
potential model: ["card(9,spades)", "card(king,hearts)", "card(10,diamonds)", "
    card(ace,clubs)"]
potential model: ["card(7,spades)", "card(2,spades)", "card(10,clubs)", "card(jack
    ,clubs)"]
potential model: ["card(queen,spades)", "card(8,diamonds)", "card(5,diamonds)", "
    card(3,clubs)"]
potential model: ["card(8,spades)", "card(king,spades)", "card(2,hearts)", "card(
    jack,hearts)"]
potential model: ["card(10,spades)", "card(9,spades)", "card(2,spades)", "card(10,
    diamonds)"]
potential model: ["card(queen,spades)", "card(10,diamonds)", "card(4,clubs)", "
    card(2,clubs)"]
potential model: ["card(2,diamonds)", "card(9,clubs)", "card(5,clubs)", "card(jack
    ,clubs)"]
potential model: ["card(6,spades)", "card(5,spades)", "card(2,hearts)", "card(5,
    diamonds)"]
potential model: ["card(9,spades)", "card(5,spades)", "card(king,hearts)", "card
    (7,diamonds)"]
potential model: ["card(8,spades)", "card(5,spades)", "card(3,hearts)", "card(9,
    clubs)"]
```

Figure 7.2: The hands of our opponent in hyper games found by our system which are consistent with the game so far.

As there is no card which can go between them, none of our hyper games shows a situation where the other player has a card which could go between them. Therefore our values of each move are exactly the same!

```
card(8,clubs):  -6.067260148844877
card(7,clubs):  -6.067260148844877
```

8 Appendix B: Final Output

For each of the three games we tested on we include our FCDL game rules and the pseudo-heuristics which were output from these.

8.1 Trumps

The flow chart for the game of trumps is actually slightly different to that found in the earlier sections. We simplified it slightly for the report. This is because FCDL doesn't have the not equal operator. This meant that we had to use two choice nodes in some places instead of one.

The game rules for our FCDL game of trumps:

```
state(0,startTerminal(1)).
state(54,endTerminal).
state(60,statement(51,assign(leader,playerNumber(trickWinner)))).
state(51,statement(11,"Stack.empty")).
state(49,statement(60,assign(score(trickWinner),operation(add,score(trickWinner)
    ,1)))).
state(36,statement(38,"playCard(currentPlayer,card(any,any))")).
state(33,statement(21,assign(i,operation(add,i,1)))).
state(31,statement(33,assign(trickWinner,currentPlayer))).
state(27,statement(29,"playCard(currentPlayer,card(any,leadSuit))")).
state(23,statement(25,assign(currentPlayer,player(operation(modulo,operation(add,
    leader,i),numberOfPlayers))))).
state(19,statement(21,assign(i,1))).
state(17,statement(19,assign(leadSuit,suit(middle(player(leader)))))).
state(15,statement(17,assign(trickWinner,player(leader)))).
state(13,statement(15,"playCard(player(leader),card(any,any))")).
state(9,statement(11,assign(r,0))).
state(70,statement(9,publish("#{trumps}~has~been~chosen~as~trumps."))).
state(7,statement(70,assign(trumps,suit(random(4))))).
state(5,statement(7,assign(leader,random(numberOfPlayers)))).
state(3,statement(5,"Deal(7)")).
state(1,statement(3,"Shuffle")).
state(42,choice(31,33,lessThan(rankValue(middle(trickWinner)),rankValue(middle(
    currentPlayer))))).
state(40,choice(42,31,equal(suit(middle(trickWinner)),trumps))).
state(38,choice(40,33,equal(suit(middle(currentPlayer)),trumps))).
state(29,choice(31,33,operation(and,equal(suit(middle(trickWinner)),leadSuit),
    lessThan(rankValue(middle(trickWinner)),rankValue(middle(currentPlayer)))))).
state(25,choice(27,36,playerHas(currentPlayer,card(any,leadSuit)))).
state(21,choice(23,49,lessThan(i,numberOfPlayers))).
state(11,choice(13,54,greaterThan(cardsRemaining(player(0)),0))).
```

As we have more choice nodes, we have more paths through the main loop of the game, leading to 12 pseudo-heuristics rather than the 8 given in the report. They are however equivalent.

The pseudo-heuristics for our FCDL game of trumps:

```
heuristicConjunct(lessThan(rankValue(myCard),rankValue(otherCard)),-1).
heuristicConjunct(equal(suit(otherCard),trumps),-1).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),-1).
heuristicConjunct(neg(playerHas(otherPlayer,card(any,suit(myCard)))),-1).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),-1).
heuristicConjunct(equal(player(leader),me),-1).
heuristicConjunct(equal(suit(myCard),trumps),-1).



heuristicConjunct(lessThan(rankValue(otherCard),rankValue(myCard)),1).
heuristicConjunct(equal(suit(myCard),trumps),1).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),1).
heuristicConjunct(neg(playerHas(me,card(any,suit(otherCard)))),1).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),1).
heuristicConjunct(equal(player(leader),otherPlayer),1).
heuristicConjunct(equal(suit(otherCard),trumps),1).



heuristicConjunct(equal(suit(otherCard),trumps),-1).
heuristicConjunct(equal(player(leader),otherPlayer),-1).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),-1).
heuristicConjunct(neg(playerHas(me,card(any,suit(otherCard)))),-1).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),-1).
heuristicConjunct(equal(suit(myCard),trumps),-1).
heuristicConjunct(neg(lessThan(rankValue(otherCard),rankValue(myCard))),-1).



heuristicConjunct(neg(equal(suit(myCard),trumps)),-1).
heuristicConjunct(equal(player(leader),me),-1).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),-1).
heuristicConjunct(neg(playerHas(otherPlayer,card(any,suit(myCard)))),-1).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),-1).
heuristicConjunct(equal(suit(otherCard),trumps),-1).



heuristicConjunct(neg(equal(suit(otherCard),trumps)),1).
heuristicConjunct(equal(player(leader),otherPlayer),1).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),1).
heuristicConjunct(neg(playerHas(me,card(any,suit(otherCard)))),1).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),1).
heuristicConjunct(equal(suit(myCard),trumps),1).



heuristicConjunct(lessThan(rankValue(myCard),rankValue(otherCard)),-1).
heuristicConjunct(cardInstanceOf(otherCard,card(any,suit(myCard))),-1).
heuristicConjunct(playerHas(otherPlayer,card(any,suit(myCard))),-1).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),-1).
heuristicConjunct(equal(player(leader),me),-1).
heuristicConjunct(lessThan(rankValue(otherCard),rankValue(myCard)),1).
heuristicConjunct(cardInstanceOf(myCard,card(any,suit(otherCard))),1).
heuristicConjunct(playerHas(me,card(any,suit(otherCard))),1).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),1).
heuristicConjunct(equal(player(leader),otherPlayer),1).



heuristicConjunct(equal(player(leader),otherPlayer),-1).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),-1).
heuristicConjunct(neg(playerHas(me,card(any,suit(otherCard)))),-1).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),-1).
heuristicConjunct(neg(equal(suit(myCard),trumps)),-1).



heuristicConjunct(equal(player(leader),otherPlayer),-1).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),-1).
heuristicConjunct(playerHas(me,card(any,suit(otherCard))),-1).
heuristicConjunct(cardInstanceOf(myCard,card(any,suit(otherCard))),-1).
heuristicConjunct(neg(lessThan(rankValue(otherCard),rankValue(myCard))),-1).



heuristicConjunct(equal(suit(myCard),trumps),1).
heuristicConjunct(equal(player(leader),me),1).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),1).
heuristicConjunct(neg(playerHas(otherPlayer,card(any,suit(myCard)))),1).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),1).
heuristicConjunct(equal(suit(otherCard),trumps),1).
heuristicConjunct(neg(lessThan(rankValue(myCard),rankValue(otherCard))),1).



heuristicConjunct(equal(player(leader),me),1).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),1).
heuristicConjunct(neg(playerHas(otherPlayer,card(any,suit(myCard)))),1).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),1).
heuristicConjunct(neg(equal(suit(otherCard),trumps)),1).



heuristicConjunct(equal(player(leader),me),1).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),1).
heuristicConjunct(playerHas(otherPlayer,card(any,suit(myCard))),1).
heuristicConjunct(cardInstanceOf(otherCard,card(any,suit(myCard))),1).
heuristicConjunct(neg(lessThan(rankValue(myCard),rankValue(otherCard))),1).
```

8.2 Kings and Jacks

The game rules for our FCDL game of Kings and Jacks:

```
state(0,startTerminal(1)).
state(36,endTerminal).
state(40,statement(7,"Stack.empty")).
state(38,statement(40,assign(leader,trickWinner))).
state(32,statement(38,assign(score(player(trickWinner)),operation(subtract,score(
    player(trickWinner)),10)))).
state(28,statement(30,assign(score(player(trickWinner)),operation(subtract,score(
    player(trickWinner)),10)))).
state(24,statement(26,assign(trickWinner,follower))).
state(19,statement(26,assign(trickWinner,leader))).
state(17,statement(21,"playCard(player(follower),card(any,leadSuit))")).
state(15,statement(19,"playCard(player(follower),card(any,any))")).
state(11,statement(13,assign(leadSuit,suit(middle(player(leader)))))).
state(9,statement(11,"playCard(player(leader),card(any,any))")).
state(45,statement(9,assign(follower,operation(modulo,operation(add,leader,1),2)))).
state(5,statement(7,assign(leader,random(numberOfPlayers)))).
state(3,statement(5,"Deal(7)")).
state(1,statement(3,"Shuffle")).
state(30,choice(32,38,operation(or,equal(rank(middle(player(1))),king),equal(rank(
    middle(player(1))),jack)))).
state(26,choice(28,30,operation(or,equal(rank(middle(player(0))),king),equal(rank(
    middle(player(0))),jack)))).
state(21,choice(24,19,lessThan(rankValue(middle(player(leader))),rankValue(middle(
    player(follower)))))).
state(13,choice(17,15,playerHas(player(follower),card(any,leadSuit)))).
state(7,choice(45,36,greaterThan(cardsRemaining(player(0)),0))).
stepState(0,startTerminal(1)).
```

The pseudo-heuristics for our FCDL game of Kings and Jacks:

```
heuristicConjunct(equal(player(leader),otherPlayer),20).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),20).
heuristicConjunct(playerHas(me,card(any,suit(otherCard))),20).
heuristicConjunct(cardInstanceOf(myCard,card(any,suit(otherCard))),20).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),me),20).
heuristicConjunct(neg(lessThan(rankValue(otherCard),rankValue(myCard))),20).
heuristicConjunct(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack))
    ,20).
heuristicConjunct(operation(or,equal(rank(otherCard),king),equal(rank(otherCard),
    jack)),20).



heuristicConjunct(equal(player(leader),me),20).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),20).
heuristicConjunct(playerHas(otherPlayer,card(any,suit(myCard))),20).
heuristicConjunct(cardInstanceOf(otherCard,card(any,suit(myCard))),20).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),
    otherPlayer),20).
heuristicConjunct(lessThan(rankValue(myCard),rankValue(otherCard)),20).
heuristicConjunct(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack))
    ,20).
heuristicConjunct(operation(or,equal(rank(otherCard),king),equal(rank(otherCard),
    jack)),20).



heuristicConjunct(neg(operation(or,equal(rank(otherCard),king),equal(rank(otherCard)
    ,jack))),10).
```

```
heuristicConjunct(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack))
    ,10).
heuristicConjunct(neg(lessThan(rankValue(otherCard),rankValue(myCard))),10).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),me),10).
heuristicConjunct(cardInstanceOf(myCard,card(any,suit(otherCard))),10).
heuristicConjunct(playerHas(me,card(any,suit(otherCard))),10).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),10).
heuristicConjunct(equal(player(leader),otherPlayer),10).



heuristicConjunct(neg(operation(or,equal(rank(otherCard),king),equal(rank(otherCard)
    ,jack))),10).
heuristicConjunct(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack))
    ,10).
heuristicConjunct(lessThan(rankValue(myCard),rankValue(otherCard)),10).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),
    otherPlayer),10).
heuristicConjunct(cardInstanceOf(otherCard,card(any,suit(myCard))),10).
heuristicConjunct(playerHas(otherPlayer,card(any,suit(myCard))),10).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),10).
heuristicConjunct(equal(player(leader),me),10).



heuristicConjunct(neg(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack)
    )),10).
heuristicConjunct(neg(lessThan(rankValue(otherCard),rankValue(myCard))),10).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),me),10).
heuristicConjunct(cardInstanceOf(myCard,card(any,suit(otherCard))),10).
heuristicConjunct(playerHas(me,card(any,suit(otherCard))),10).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),10).
heuristicConjunct(equal(player(leader),otherPlayer),10).
heuristicConjunct(operation(or,equal(rank(otherCard),king),equal(rank(otherCard),
    jack)),10).



heuristicConjunct(neg(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack)
    )),10).
heuristicConjunct(lessThan(rankValue(myCard),rankValue(otherCard)),10).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),
    otherPlayer),10).
heuristicConjunct(cardInstanceOf(otherCard,card(any,suit(myCard))),10).
heuristicConjunct(playerHas(otherPlayer,card(any,suit(myCard))),10).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),10).
heuristicConjunct(equal(player(leader),me),10).
heuristicConjunct(operation(or,equal(rank(otherCard),king),equal(rank(otherCard),
    jack)),10).



heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),me),20).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),20).
heuristicConjunct(neg(playerHas(me,card(any,suit(otherCard)))),20).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),20).
heuristicConjunct(equal(player(leader),otherPlayer),20).
heuristicConjunct(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack))
    ,20).
heuristicConjunct(operation(or,equal(rank(otherCard),king),equal(rank(otherCard),
    jack)),20).



heuristicConjunct(neg(operation(or,equal(rank(otherCard),king),equal(rank(otherCard)
    ,jack))),10).
heuristicConjunct(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack))
    ,10).
heuristicConjunct(equal(player(leader),otherPlayer),10).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),10).
heuristicConjunct(neg(playerHas(me,card(any,suit(otherCard)))),10).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),10).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),me),10).



heuristicConjunct(neg(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack)
    )),10).
heuristicConjunct(equal(player(leader),otherPlayer),10).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),10).
heuristicConjunct(neg(playerHas(me,card(any,suit(otherCard)))),10).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),10).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),me),10).
heuristicConjunct(operation(or,equal(rank(otherCard),king),equal(rank(otherCard),
    jack)),10).



heuristicConjunct(equal(player(leader),me),-20).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),-20).
heuristicConjunct(playerHas(otherPlayer,card(any,suit(myCard))),-20).
heuristicConjunct(cardInstanceOf(otherCard,card(any,suit(myCard))),-20).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),
    otherPlayer),-20).
heuristicConjunct(neg(lessThan(rankValue(myCard),rankValue(otherCard))),-20).
heuristicConjunct(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack))
    ,-20).
heuristicConjunct(operation(or,equal(rank(otherCard),king),equal(rank(otherCard),
    jack)),-20).



heuristicConjunct(equal(player(leader),otherPlayer),-20).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),-20).
heuristicConjunct(playerHas(me,card(any,suit(otherCard))),-20).
heuristicConjunct(cardInstanceOf(myCard,card(any,suit(otherCard))),-20).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),me),-20)
    .
heuristicConjunct(lessThan(rankValue(otherCard),rankValue(myCard)),-20).
heuristicConjunct(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack))
    ,-20).
heuristicConjunct(operation(or,equal(rank(otherCard),king),equal(rank(otherCard),
    jack)),-20).
```

```
heuristicConjunct(neg(operation(or,equal(rank(otherCard),king),equal(rank(otherCard)
    ,jack))),-10).
heuristicConjunct(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack))
    ,-10).
heuristicConjunct(neg(lessThan(rankValue(myCard),rankValue(otherCard))),-10).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),
    otherPlayer),-10).
heuristicConjunct(cardInstanceOf(otherCard,card(any,suit(myCard))),-10).
heuristicConjunct(playerHas(otherPlayer,card(any,suit(myCard))),-10).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),-10).
heuristicConjunct(equal(player(leader),me),-10).



heuristicConjunct(neg(operation(or,equal(rank(otherCard),king),equal(rank(otherCard)
    ,jack))),-10).
heuristicConjunct(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack))
    ,-10).
heuristicConjunct(lessThan(rankValue(otherCard),rankValue(myCard)),-10).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),me),-10)
    .
heuristicConjunct(cardInstanceOf(myCard,card(any,suit(otherCard))),-10).
heuristicConjunct(playerHas(me,card(any,suit(otherCard))),-10).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),-10).
heuristicConjunct(equal(player(leader),otherPlayer),-10).



heuristicConjunct(neg(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack)
    )),-10).
heuristicConjunct(neg(lessThan(rankValue(myCard),rankValue(otherCard))),-10).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),
    otherPlayer),-10).
heuristicConjunct(cardInstanceOf(otherCard,card(any,suit(myCard))),-10).
heuristicConjunct(playerHas(otherPlayer,card(any,suit(myCard))),-10).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),-10).
heuristicConjunct(equal(player(leader),me),-10).
heuristicConjunct(operation(or,equal(rank(otherCard),king),equal(rank(otherCard),
    jack)),-10).



heuristicConjunct(neg(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack)
    )),-10).
heuristicConjunct(lessThan(rankValue(otherCard),rankValue(myCard)),-10).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),me),-10)
    .
heuristicConjunct(cardInstanceOf(myCard,card(any,suit(otherCard))),-10).
heuristicConjunct(playerHas(me,card(any,suit(otherCard))),-10).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),-10).
heuristicConjunct(equal(player(leader),otherPlayer),-10).
heuristicConjunct(operation(or,equal(rank(otherCard),king),equal(rank(otherCard),
    jack)),-10).
```

```
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),
    otherPlayer),-20).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),-20).
heuristicConjunct(neg(playerHas(otherPlayer,card(any,suit(myCard)))),-20).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),-20).
heuristicConjunct(equal(player(leader),me),-20).
heuristicConjunct(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack))
    ,-20).
heuristicConjunct(operation(or,equal(rank(otherCard),king),equal(rank(otherCard),
    jack)),-20).



heuristicConjunct(neg(operation(or,equal(rank(otherCard),king),equal(rank(otherCard)
    ,jack))),-10).
heuristicConjunct(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack))
    ,-10).
heuristicConjunct(equal(player(leader),me),-10).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),-10).
heuristicConjunct(neg(playerHas(otherPlayer,card(any,suit(myCard)))),-10).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),-10).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),
    otherPlayer),-10).



heuristicConjunct(neg(operation(or,equal(rank(myCard),king),equal(rank(myCard),jack)
    )),-10).
heuristicConjunct(equal(player(leader),me),-10).
heuristicConjunct(cardInstanceOf(myCard,card(any,any)),-10).
heuristicConjunct(neg(playerHas(otherPlayer,card(any,suit(myCard)))),-10).
heuristicConjunct(cardInstanceOf(otherCard,card(any,any)),-10).
heuristicConjunct(equal(player(operation(modulo,operation(add,leader,1),2)),
    otherPlayer),-10).
heuristicConjunct(operation(or,equal(rank(otherCard),king),equal(rank(otherCard),
    jack)),-10).
```

8.3 Crazy Eights

The game rules for our FCDL game of Crazy Eights:

```
state(0,startTerminal(1)).
state(53,endTerminal).
state(64,statement(5,assign(lastRank,any))).
state(62,statement(64,assign(lastSuit,any))).
state(60,statement(62,assign(leader,random(2)))).
state(55,statement(53,assign(score(player(0)),operation(add,score(player(0)),1)))).
state(51,statement(53,assign(score(player(1)),operation(add,score(player(1)),1)))).
state(45,statement(5,assign(lastSuit,suit(middle(player(operation(modulo,operation(
    add,leader,1),2))))))).
state(43,statement(45,assign(lastRank,rank(middle(player(operation(modulo,operation(
    add,leader,1),2))))))).
state(41,statement(43,"pickUp(player(leader),4)")).
state(98,statement(11,"Stack.empty")).
state(38,statement(98,assign(lastSuit,suit(middle(player(leader)))))).
state(36,statement(38,assign(lastRank,rank(middle(player(leader)))))).
state(34,statement(36,"pickUp(player(follower),2)")).
state(30,statement(32,"playCardOrPass(player(follower),card(2,any))")).
state(25,statement(11,assign(lastRank,8))).
state(23,statement(25,assign(lastSuit,suit(middle(player(leader)))))).
state(18,statement(5,assign(lastSuit,suit(middle(player(leader)))))).
state(16,statement(18,assign(lastRank,rank(middle(player(leader)))))).
state(11,statement(5,assign(leader,follower))).
state(90,statement(11,"pickUp(player(leader),1)")).
state(70,statement(9,assign(follower,operation(modulo,operation(add,leader,1),2)))).
state(7,statement(70,"playCardOrPass(player(leader),union(card(any,lastSuit),union(
    card(lastRank,any),card(8,any))))")).
state(3,statement(60,"Deal(8)")).
state(1,statement(3,"Shuffle")).
state(49,choice(55,51,equal(cardsRemaining(player(0)),0))).
state(32,choice(34,41,equal(passed(player(follower)),true))).
state(28,choice(30,36,equal(rank(middle(player(leader))),2))).
state(21,choice(23,28,equal(rank(middle(player(leader))),8))).
state(14,choice(16,21,equal(rank(middle(player(leader))),king))).
state(9,choice(90,14,equal(passed(player(leader)),true))).
state(5,choice(7,49,operation(and,greaterThan(cardsRemaining(player(0)),0),
    greaterThan(cardsRemaining(player(1)),0)))).
stepState(0,startTerminal(1)).
```

The pseudo-heuristics for our FCDL game of Crazy Eights:

```
heuristicConjunct(neg(equal(rank(otherCard),8)),-3).
heuristicConjunct(neg(equal(passed(otherPlayer),true)),-3).
heuristicConjunct(cardInstanceOf(otherCard,union(card(any,lastSuit),union(card(
    lastRank,any),card(8,any)))),-3).
heuristicConjunct(equal(player(leader),otherPlayer),-3).
heuristicConjunct(neg(equal(rank(otherCard),king)),-3).
heuristicConjunct(equal(rank(otherCard),2),-3).
heuristicConjunct(cardInstanceOf(myCard,card(2,any)),-3).
heuristicConjunct(equal(player(follower),me),-3).
heuristicConjunct(equal(passed(me),true),-3).



heuristicConjunct(neg(equal(rank(myCard),8)),3).
heuristicConjunct(neg(equal(passed(me),true)),3).
heuristicConjunct(cardInstanceOf(myCard,union(card(any,lastSuit),union(card(lastRank
    ,any),card(8,any)))),3).
heuristicConjunct(equal(player(leader),me),3).
heuristicConjunct(neg(equal(rank(myCard),king)),3).
heuristicConjunct(equal(rank(myCard),2),3).
heuristicConjunct(cardInstanceOf(otherCard,card(2,any)),3).
heuristicConjunct(equal(player(follower),otherPlayer),3).
heuristicConjunct(equal(passed(otherPlayer),true),3).



heuristicConjunct(neg(equal(rank(myCard),8)),-4).
heuristicConjunct(neg(equal(passed(me),true)),-4).
heuristicConjunct(cardInstanceOf(myCard,union(card(any,lastSuit),union(card(lastRank
    ,any),card(8,any)))),-4).
```

```
heuristicConjunct(equal(player(leader),me),-4).
heuristicConjunct(neg(equal(rank(myCard),king)),-4).
heuristicConjunct(equal(rank(myCard),2),-4).
heuristicConjunct(cardInstanceOf(otherCard,card(2,any)),-4).
heuristicConjunct(equal(player(follower),otherPlayer),-4).
heuristicConjunct(neg(equal(passed(otherPlayer),true)),-4).



heuristicConjunct(neg(equal(rank(otherCard),8)),4).
heuristicConjunct(neg(equal(passed(otherPlayer),true)),4).
heuristicConjunct(cardInstanceOf(otherCard,union(card(any,lastSuit),union(card(
    lastRank,any),card(8,any)))),4).
heuristicConjunct(equal(player(leader),otherPlayer),4).
heuristicConjunct(neg(equal(rank(otherCard),king)),4).
heuristicConjunct(equal(rank(otherCard),2),4).
heuristicConjunct(cardInstanceOf(myCard,card(2,any)),4).
heuristicConjunct(equal(player(follower),me),4).
heuristicConjunct(neg(equal(passed(me),true)),4).



heuristicConjunct(neg(equal(rank(myCard),2)),1).
heuristicConjunct(neg(equal(rank(myCard),king)),1).
heuristicConjunct(equal(player(leader),me),1).
heuristicConjunct(cardInstanceOf(myCard,union(card(any,lastSuit),union(card(lastRank
    ,any),card(8,any)))),1).
heuristicConjunct(neg(equal(passed(me),true)),1).
heuristicConjunct(neg(equal(rank(myCard),8)),1).



heuristicConjunct(neg(equal(rank(myCard),king)),1).
heuristicConjunct(equal(player(leader),me),1).
heuristicConjunct(cardInstanceOf(myCard,union(card(any,lastSuit),union(card(lastRank
    ,any),card(8,any)))),1).
heuristicConjunct(neg(equal(passed(me),true)),1).
heuristicConjunct(equal(rank(myCard),8),1).



heuristicConjunct(equal(rank(myCard),king),1).
heuristicConjunct(equal(player(leader),me),1).
heuristicConjunct(cardInstanceOf(myCard,union(card(any,lastSuit),union(card(lastRank
    ,any),card(8,any)))),1).
heuristicConjunct(neg(equal(passed(me),true)),1).



heuristicConjunct(equal(passed(me),true),-1).
heuristicConjunct(cardInstanceOf(myCard,union(card(any,lastSuit),union(card(lastRank
    ,any),card(8,any)))),-1).
heuristicConjunct(equal(player(leader),me),-1).



heuristicConjunct(neg(equal(rank(otherCard),2)),-1).
heuristicConjunct(neg(equal(rank(otherCard),king)),-1).
heuristicConjunct(equal(player(leader),otherPlayer),-1).
heuristicConjunct(cardInstanceOf(otherCard,union(card(any,lastSuit),union(card(
    lastRank,any),card(8,any)))),-1).
heuristicConjunct(neg(equal(passed(otherPlayer),true)),-1).
heuristicConjunct(neg(equal(rank(otherCard),8)),-1).



heuristicConjunct(neg(equal(rank(otherCard),king)),-1).
heuristicConjunct(equal(player(leader),otherPlayer),-1).
heuristicConjunct(cardInstanceOf(otherCard,union(card(any,lastSuit),union(card(
    lastRank,any),card(8,any)))),-1).
heuristicConjunct(neg(equal(passed(otherPlayer),true)),-1).
heuristicConjunct(equal(rank(otherCard),8),-1).



heuristicConjunct(equal(rank(otherCard),king),-1).
heuristicConjunct(equal(player(leader),otherPlayer),-1).
heuristicConjunct(cardInstanceOf(otherCard,union(card(any,lastSuit),union(card(
    lastRank,any),card(8,any)))),-1).
heuristicConjunct(neg(equal(passed(otherPlayer),true)),-1).



heuristicConjunct(equal(passed(otherPlayer),true),1).
heuristicConjunct(cardInstanceOf(otherCard,union(card(any,lastSuit),union(card(
    lastRank,any),card(8,any)))),1).
heuristicConjunct(equal(player(leader),otherPlayer),1).
```

References

[1] M. Schofield, T. Cerexhe, M. Thielscher, HyperPlay : A Solution to General Game Playing with Imperfect Information (2012) 1606–1612.

[2] M. Genesereth, N. Love, B. Pell, General Game Playing 26 (2) (2005) 62–72.

[3] M. Ponsen, J. Ramon, T. Croonenborghs, K. Driessens, K. Tuyls, Bayes-Relational Learning of Opponent Models from Incomplete Information in No-Limit Poker Learning an Opponent Model (2008) 1485–1486.

[4] J. Clune, Heuristic Evaluation Functions for General Game Playing (2007) 1134–1139.

[5] http://migo.sixbit.org/papers/AI_and_Perl/game-tree.gif%0A.
URL `http://migo.sixbit.org/papers/AI_and_Perl/game-tree.gif`

[6] B. Banerjee, P. Stone, General game learning using knowledge transfer, The 20th International Joint Conference on Artificial ... (2007) 672–677.
URL `http://www.aaai.org/Papers/IJCAI/2007/IJCAI07-107.pdf`

[7] G. Kuhlmann, K. Dresner, P. Stone, Automatic Heuristic Construction in a Complete General Game Player 1457–1462.

[8] M. Thielscher, A General Game Description Language for Incomplete Information Games 994–999.

[9] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, ... Conference on Logic programming.
URL `http://www.cse.unsw.edu.au/~cs4415/2010/resources/stable.pdf`

[10] M. Gebser, A Users Guide to gringo clasp clingo and iclingo.

[11] Ruby on Rails getting started guide.
URL `http://guides.rubyonrails.org/getting_started.html`

[12] Faye: simple pub/sub messaging for the web.
URL `http://faye.jcoglan.com/`

[13] H. Finnsson, Y. Björnsson, Simulation-based approach to general game playing, The Twenty-Third AAAI Conference on Artificial ... (2008) 259–264.
URL `http://www.aaai.org/Papers/AAAI/2008/AAAI08-041.pdf`

9 Appendix C: Instructions for Viewing the Implementation on the Server

Our working version of the implementation can be found at http://176.58.104.120:3000/.

The user will be directed to sign in or create an account. Once this has been done there is a link to the game definitions created by that user. After clicking on this link the user will see an (empty) list of game definitions. To create a new game definition the user should click "Define new game".

This will take the user to the game creation page. Here a user can create a game by inputting a flow chart. (The elements are created by clicking on the outgoing connection of the previous element and then clicking on the button to create the new element). However, the chances of creating a correct flow chart this way first time are very low (and debugging is difficult). We therefore included an advanced edit button. This will take the user to a screen with a box containing the code for the flow chart (this will be empty if the flow chart hasn't been saved yet).

To test the game of trumps please input the following code:

```
terminal(0,0,0,"StartGame").
statement(1,0,200,"Shuffle").
connection(2,0,2,1,0).
statement(3,0,400,"Deal(7)").
connection(4,1,2,3,0).
statement(5,0,600,"leader=random(numberOfPlayers)").
connection(6,3,2,5,0).
statement(7,0,800,"trumps=suit(random(4))").
connection(8,5,2,7,0).
statement(9,0,1000,publish("#{trumps} has been chosen as trumps.")).
connection(10,7,2,9,0).
choice(11,0,1200,"cardsRemaining(player(0))>0").
connection(12,9,2,11,0).
statement(13,0,1400,"playCard(player(leader),card(any,any))").
connection(14,11,2,13,0).
statement(15,0,1600,"trickWinner=player(leader)").
connection(16,13,2,15,0).
statement(17,0,1800,"leadSuit=suit(middle(player(leader)))").
connection(18,15,2,17,0).
statement(19,0,2000,"i=1").
connection(20,17,2,19,0).
choice(21,0,2200,"i<numberOfPlayers").
connection(22,19,2,21,0).
statement(23,0,2400,"currentPlayer=player((leader+i)%numberOfPlayers)").
connection(24,21,2,23,0).
choice(25,0,2600,"playerHas(currentPlayer,card(any,leadSuit))").
connection(26,23,2,25,0).
statement(27,0,2800,"playCard(currentPlayer,card(any,leadSuit))").
connection(28,25,2,27,0).
choice(29,0,3000,"suit(middle(trickWinner))==leadSuit&rankValue(middle(trickWinner))
    <rankValue(middle(currentPlayer))").
```

```
connection(30,27,2,29,0).
statement(31,0,3200,"trickWinner=currentPlayer").
connection(32,29,2,31,0).
statement(33,0,3400,"i=i+1").
connection(34,31,2,33,0).
connection(35,29,1,33,0).
statement(36,400,2600,"playCard(currentPlayer,card(any,any))").
connection(37,25,1,36,0).
choice(38,400,2800,"suit(middle(currentPlayer))==trumps").
connection(39,36,2,38,0).
choice(40,400,3000,"suit(middle(trickWinner))==trumps").
connection(41,38,2,40,0).
choice(42,400,3200,"rankValue(middle(trickWinner))<rankValue(middle(currentPlayer))
    ").
connection(43,40,2,42,0).
connection(44,38,1,33,0).
connection(45,40,1,31,0).
connection(46,42,1,33,0).
connection(47,42,2,31,0).
connection(48,33,2,21,0).
statement(49,400,2200,"score(trickWinner)=score(trickWinner)+1").
connection(50,21,1,49,0).
statement(51,400,2400,"Stack.empty").
statement(60,400,2800,"leader=playerNumber(trickWinner)").
connection(52,49,2,60,0).
connection(61,60,2,51,0).
connection(53,51,2,11,0).
terminal(54,400,1200,"TerminateGame").
connection(55,11,1,54,0).
```

The name of the game isn't important but the number of players *must* be 2.

After updating the flow chart the user will be redirected to a page where they will see the code for the flow chart they have just entered.

They should then first click the generate rules link. After this they should click the generate heuristics link (this may take a few minutes).

After this they can view the (pseudo) heuristics, or play the game.

Clicking play game will take them to the page containing the list of games. There are two options: playing the game against an AI player or against another user. To play another user, the link to the created game must be shared with this user.