Bridging the gap between logic and cognition: a translation method for centipede games

Jordi Top **September 2016**

Master's thesis
Artificial Intelligence
University of Groningen

Supervisors:
Prof. dr. L.C. Verbrugge, University of Groningen
Trudy Buwalda, MSc., University of Groningen

university of groningen

Bridging the gap between logic and cognition: a translation method for centipede games

Jordi Top **September 2016**

Master's thesis: MSc Human-Machine Communication
Faculty of Mathematics and Natural Sciences
University of Groningen

European credits: 45 ECTS

First supervisor: Prof. dr. L.C. Verbrugge
University of Groningen
Faculty of Mathematics and Natural Sciences
Multi-Agent Systems
Nijenborgh 9, 9747 AG Groningen, The Netherlands
Room: 5161.0355
L.C.Verbrugge@rug.nl
+31 50 363 6334

Second supervisor: Trudy Buwalda, MSc.
University of Groningen
Faculty of Mathematics and Natural Sciences
Cognitive Modeling
Nijenborgh 9, 9747 AG Groningen, The Netherlands
Room: 5161.0316
t.a.buwalda@rug.nl
+31 50 363 6860

university of groningen

Abstract

Human strategic reasoning in turn-taking games has been extensively investigated by game theorists, logicians, cognitive scientists, and psychologists. Whereas game theorists and logicians use formal methods to formalize strategic behaviour, cognitive scientists use cognitive models of the human mind to predict and simulate human behaviour. In the present body of work, we create a translation system which, starting from a strategy represented in formal logic, automatically generates a computational model in the PRIMs cognitive architecture. This model can then be run to generate response times and decisions made in centipede games, a subset of dynamic perfect-information games. We find that the results of our automatically generated models are similar to our hand-made models, verifying our translation system. Furthermore, we use our system to predict that human players' strategies correspond more closely to extensive-form rationalizable strategies than to backward induction strategies, and we predict that response times may be a function of the number of possibilities a strategy can prescribe.

List of Abbreviations

AC	Action Buffer
ACT-R	Adaptive Control of Thought - Rational
BDP	Backward Dominance Procedure
BI	Backward Induction
EFR	Extensive-Form Rationalizable
FI	Forward Induction
G	Goal Buffer
ICDP	Iterated Conditional Dominance Procedure
LCA	Latent Class Analysis
PRIMs	Primitive Information Processing Elements
RT	Retrieval Buffer
ToM	Theory Of Mind
V	Visual Buffer
WM	Working Memory Buffer

Contents

Chapter 1

Introduction

1.1 Marble drop

Many real-world interactions are comparable to turn-taking games. Examples are presidential debates, negotiating a division of labour or competing with other students, employees or even companies. When involved in such an interaction we continuously have to ask ourselves whether we should accept the current outcome, or continue - hoping for a better one.

Dynamic perfect-information games can be used to model such interactions. Dynamic perfect-information games are *dynamic* because both players take turns choosing an action, and both players can see which actions the other player has chosen in the past before they have to choose their next action. This contrasts with simultaneous games, where both players choose an action at the same time, after which these actions are revealed to either player, such as in the prisoner's dilemma. *perfect-information* games are games where both players know everything there is to know about the game - all possible actions and all possible outcomes. There are no hidden elements, and there are no chance elements.

Dynamic perfect-information games can be presented as *game trees*. A game tree is a graph where each node represents a turn and each outgoing edge represents an action that can be performed at a turn. These edges are not symmetric: you cannot traverse an edge back to the previous node. The game ends when a leaf node is reached: at each leaf node the payoff for each player is specified, which is the outcome of the game when that node is reached.

To get a better intuitive understanding of dynamic perfect-information games and game trees, let's consider the example in Figure 1.1. Black dots are non-leaf nodes and arrows are edges (and indicate their direction). Let's suppose player C is Claudia and player P is Paul. At the leaf nodes, payoffs can be found between parentheses, where the number on the left is Claudia's payoff and the number on the right is Paul's payoff. The game starts at the node on the far left with Claudia. In the remainder of this thesis, we will simplify these trees by omitting the black dots and using line segments instead of arrows.

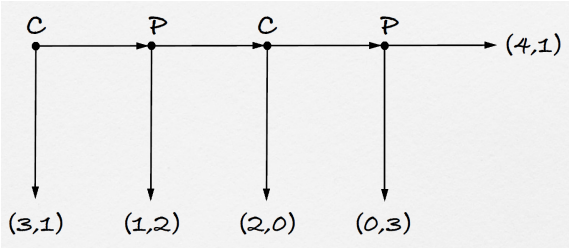

Figure 1.1: An example of a game tree of a two-player dynamic perfect-information game (adapted from Ghosh & Verbrugge (online first)). Within the payoffs (between parentheses), the number on the left is Claudia's payoff and the number on the right is Paul's payoff.

On the first turn, Claudia has two options. She can move down and end the game, giving her three points and Paul one point. She can also move right, giving Paul a turn. If she moves right, Paul has to decide whether to move down, in which case Claudia gets one point and Paul gets two points. Paul may also move right, giving Claudia a turn. The game either continues until someone moves down, or until Paul moves right in the last turn. This example is not just a dynamic perfect-information game, but also a centipede game.

In this thesis we will focus on these centipede games. Centipede games are a subset of dynamic perfect-information games. In centipede games, at each decision point, one option ends the game while the other option gives the other player a turn, until the last turn where both options end the game. Furthermore, in a centipede game, ending the game in your current turn will always give you more points than when the other player ends the game in the next turn.

Centipede games can be visually presented as the game of *marble drop*, a game where a marble rolls through a set of pipes and both players take turns deciding where the marble goes. The game of marble drop in Figure 1.2 is the same game as the game tree in Figure 1.1. Because it is intuitively easier to understand than game trees, marble drop has been used in empirical studies of centipede games (such as Ghosh, Heifetz & Verbrugge (2015) and Ghosh, Heifetz, Verbrugge & de Weerd (2017)).

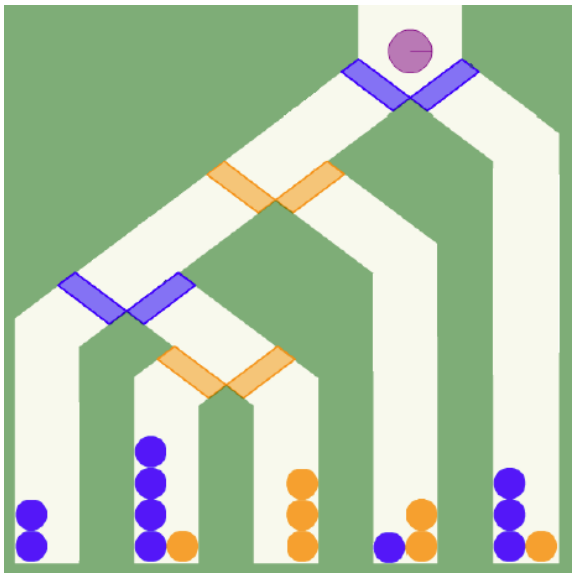

Figure 1.2: Marble drop version of the game in Figure 1.1 (adapted from Figure 4 in Ghosh & Verbrugge (online first))

1.2 Strategies

A *strategy* is a specification of how an agent should act at each decision point where it has a turn. In the example in Section 1.1, Claudia's strategy could be moving down at the first node, and moving down at the third node. In game theory, a *Nash equilibrium* is achieved when no one can change his strategy without losing points. For example, this happens when Claudia's strategy is to move down in both of her nodes, and Paul's strategy is to move down in both of his nodes as well. If Claudia changes her strategy by moving right in the first node, Paul will move down (by his strategy), and Claudia will receive one point instead of three. This can be verified for all nodes in the game tree. A *subgame perfect equilibrium* is a more general case of a Nash equilibrium. It is achieved when the current strategies achieve a Nash equilibrium in each *subgame*, which is a smaller version of a game, represented by a subtree of the corresponding game tree. For example, a subgame of the game in Figure 1.1 can be obtained by removing the first node, starting with

Paul instead. The corresponding subtree is Game 1′ in Figure 1.3. A game is also a subgame of itself.

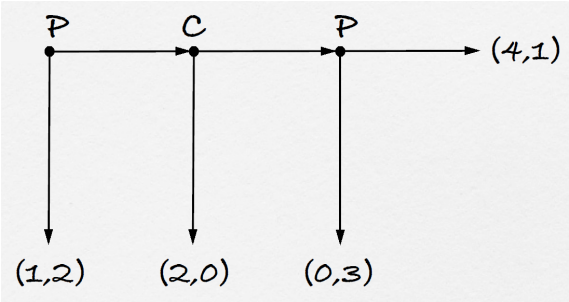

Figure 1.3: A subgame of Game 1 from Ghosh & Verbrugge (online first), obtained by removing the first node.

In dynamic perfect-information games, a subgame-perfect equilibrium can be achieved using the strategy of *backward induction*. With backward induction you start reasoning from the leaf nodes and continue backwards to the current node, ignoring any past nodes. You assume that the other player does the same. At a node that only has outgoing edges to two leaf nodes, you assume that the current player will select the action that gains him the highest number of points. You then assign this outcome to this node, and continue with the same reasoning from the previous node.

To illuminate backward induction, let us use Figure 1.1 on page 11 as an example. Suppose Claudia is using backward induction. The only node that only has outgoing edges to leaf nodes is the rightmost non-leaf node, which is Paul's. Paul has to choose between going down for $(0, 3)$ and going right for $(4, 1)$. Claudia assumes that Paul will go for three points instead of one, so she assumes that Paul will go down. Therefore, she assigns the value $(0, 3)$ to the rightmost non-leaf node. Using this value, she would have to choose between going down and getting $(2, 0)$ and going right to get $(0, 3)$ at the third node, so she will choose $(2, 0)$. She then assigns $(2, 0)$ to the third node and starts thinking about the second node, where Paul would have to choose between $(1, 2)$ when going down and $(2, 0)$ when going right. She assumes Paul would prefer two points over one point, so she assumes that Paul will go down for $(1, 2)$. Therefore, she assigns the value $(1, 2)$ to the second node. Moving to the first node, she has to choose between going down for $(3, 1)$, or going right for $(1, 2)$. Obviously, she prefers three points over one, so she decides to go down. These choices remain the same when one or more of the first nodes are removed: past actions do not influence backward induction behaviour. The full game tree with corresponding value assignments can be found in Figure 1.4.

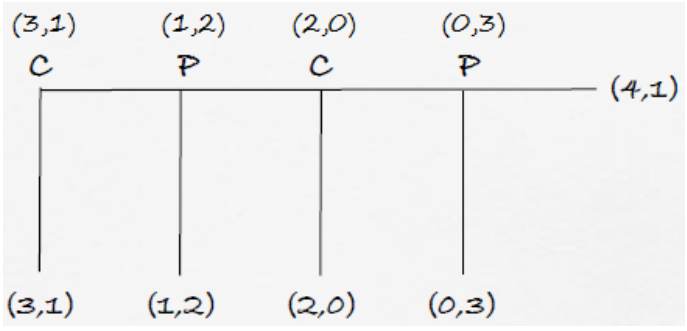

Figure 1.4: The game tree from Figure 1.1 with backward induction payoffs assigned to each decision point.

As opposed to backward induction, the strategy of *forward induction* does take past actions into account. Suppose Claudia decides to move right in her turn in the game in Figure 1.1. In forward induction, players try to rationalize their opponent's past moves. One such rationalization may be 'Claudia is not going down to get three points, because she wants to reach the four points on the far right'. If Paul is using forward induction, he may think that Claudia's strategy is to move right in both of her turns. Paul can take advantage of this by moving right at his first node, and moving down at his second node, denying Claudia four points and getting three points for himself, instead of the two points he would have gotten if he moved down immediately. According to the findings of Ghosh et al. (2015), people usually do not use backward induction. Their behaviour often corresponds to forward induction, but there may be alternative explanations, such as the extent of risk aversion people attribute to their opponent.

1.3 Cognitive modelling and previous work

This work continues from Ghosh & Verbrugge (online first). In their paper, they try to understand how people make decisions in centipede games and how to classify players and their strategies. They perform an analysis of the results of the experiment performed in Ghosh et al. (2015). In Ghosh et al. (2015), subjects had to play centipede games such as the one above against a computer. The computer often deviated from backward induction by moving right instead of moving down in its first node. There were 50 subjects who played 48 games each, for a grand total of 2400 games played. Ghosh et al. (2015) found that people often do not use backward induction: they use forward induction or a seemingly random strategy.

Ghosh & Verbrugge (online first) performed a latent class analysis and a theory-of-mind analysis on these findings. Latent Class Analysis is a statistical method used to assign subjects to groups using a probability of group membership (instead of absolute membership). They found three classes: players who use forward induction, players who play randomly, and players who start by playing randomly and learn to use forward induction over the course of the experiment.

They also performed a theory-of-mind analysis on the same data. Theory of mind refers to the ability to attribute beliefs and thoughts to others. Zero-order theory of mind is thinking about the world. First-order theory of mind is thinking about how other people think about the world. Second-order theory of mind is thinking about how others think about how others think about the world. For example, suppose two people, Paul and Claudia, are playing hide-and-seek. There are two locations to hide: behind a fence and in a bush. Paul knows that in the past, Claudia hid behind the fence more often than in the bush. If Paul thinks "Claudia often hides behind the fence, so she will probably hide behind the fence this time", he is using zero-order theory of mind, because he only reasons about the world. If Paul thinks "Claudia often hides behind the fence. She knows I know that she often hides behind the fence, so she may think that I think that she will hide behind the fence again. Therefore she may expect that I will look behind the fence, so perhaps she will hide behind the bush instead", he is using second-order theory of mind, because he thinks about what Claudia thinks that he, himself, thinks. According to the analysis of Ghosh & Verbrugge (online first), most players used first-order theory of mind, but the other two levels were also present. No usage of theory of mind of an order above two was found.

Ghosh & Verbrugge (online first) not only explain and classify behaviour in centipede games, they also work towards the creation of computational models of strategies in centipede games and how to represent them in formal logic. Their formal logic extends the one created in Ghosh, Meijering & Verbrugge (2014). The logic can be used to represent dynamic perfect-information games as well as strategies and beliefs used in them.

They implement two of these strategies in PRIMs, a cognitive architecture (Taatgen, 2013b). PRIMs models the mind as a set of separate modules, such as a procedural, visual and motor module. These modules exchange information using chunks, basic pieces of information. PRIMs is specialized in modelling transfer of skill and learning. Transfer of skill has occurred when skills learned in one task are beneficial for performance in another task. In PRIMs, tasks are performed through primitive information-processing elements (each of which is called a PRIM), which either compare information or pass it around. One of PRIMs' most important features is

production compilation: when PRIMs are executed in the same order often enough, these PRIMs are combined into larger productions, which models speed-ups in learning, among other things.

The two strategies implemented in PRIMs in Ghosh & Verbrugge (online first) are a myopic strategy and an own-payoff strategy. In the myopic strategy, players only look at the current and the next payoffs. In the own-payoff strategy, players only look at their own payoffs, and not at the payoffs of other players. Ghosh and Verbrugge performed the initial leaps in bridging the gap between logic (by creating a formal logic used to represent strategies and centipede games) and cognition (by creating two models in PRIMs and classifying human strategies). We will continue on this line: our goal is to create a general, preferably automated, method of translating strategies, as specified in their logic, into PRIMs models. We will fit our models on and compare their behaviour to the results of Ghosh et al. (2017). His results are the most recent and also consist of 2400 game items in total.

Implementing such strategies in PRIMs allows us to explain human behaviour in centipede games from a cognitive modelling perspective. Unlike the previously mentioned formal logic, PRIMs can be used to model errors, deviations from a strategy, and learning. It can also make concrete predictions on reaction times, loci of attention, and brain activity. Creating models in PRIMs is often a laborious and time-consuming task, requiring considerable expertise on model creation. A system that automatically translates strategies to PRIMs models will alleviate these problems, as strategies only need to be specified in the formal logic. Such a system will be a first step in automated model creation, as well as the next step in connecting game theory, logic, and cognitive modelling.

1.4 Research goals

In this thesis, we will investigate how to translate strategies in dynamic perfect-information games, represented in the formal logic of Ghosh & Verbrugge (online first), into models in the PRIMs cognitive architecture. Our goal is to create a general translation method: a system that automatically creates a PRIMs model given a strategy represented in formal logic.

To do so, we will first implement two strategies by hand, and use our findings in the creation of our translation system. We will implement models of the myopic and own-payoff strategies, which is were Ghosh & Verbrugge (online first) end. Not only will this give us insight into translating strategies into PRIMs models, it may also validate the findings of Ghosh & Verbrugge (online first).

Because we use PRIMs, we are obliged to find out what the smallest elements of 'skill' are in centipede games. In PRIMs, action sequences are built from primitive elements, which either compare or move pieces of information. Are the smallest elements in our models PRIMs themselves or are they sequences of PRIMs?

Our models should make predictions of reaction times, scores, and choices made. We wish to compare our model results to those in Ghosh & Verbrugge (online first) and Ghosh et al. (2017).

We will not create a graphic user interface in our system, which would be a possible extension for future work. The logical formulae corresponding to the to-be-translated strategy will be hardcoded into the system. In future versions, the system could be extended with a parser for the formal logic, which would allow the user to enter strategies into the system without having to access the code.

1.5 Thesis outline

In Chapter 2 on page 17 we will discuss the previous research relevant to this thesis. We will begin by giving an in-depth description of the centipede games used in Ghosh & Verbrugge (online first) and Ghosh et al. (2017). We will also provide the reader with a full explanation of the logic

created in Ghosh & Verbrugge (online first), as well as a more detailed explanation of the PRIMs cognitive architecture. In Chapter 3 on page 37 we will describe our findings in designing the myopic and own-payoff models, as well as the model results. Chapter 4 on page 52 elaborates on the translation method we found and the encompassing system. Finally, Chapter 5 on page 70 contains a summary, discussion and interpretation of our findings, as well as directions for future research.

Chapter 2

Theoretical Background

2.1 Marble drop

In this section, we first give an overview of the relevant papers preceding this thesis. We then give an in-depth explanation of the set of centipede games we are going to use, as well as the possible strategies in these games.

This thesis continues the line of work started by Ghosh et al. (2014). Until their paper, empirical studies and cognitive modelling of centipede games were mostly separated from logical studies of centipede games. Ghosh et al. view these methods as complementary and investigate how to bridge the gap between them. In order to do so, Ghosh et al. (2014) depart from the common practice of describing idealised agents using formal logic. Instead, they focus on describing limited agents, which can be used to describe the empirically observed reasoning of human players. For this purpose, Ghosh et al. (2014) present a formal logic that can describe game trees and strategies in extensive-form games. Their logic does not include knowledge and belief operators, yet. They also create cognitive models in the ACT-R cognitive architecture capable of playing marble drop. The strategies these models use are based on strategies represented in their formal logic. In doing so, they make the first steps in bridging the gap between logic and cognitive modelling.

This line of research continues in Ghosh et al. (2015). In this paper, an experiment is performed where people play games of marble drop against a computer, one such game being depicted in Figure 1.1 on page 11. There were fifty participants, each of which played forty-eight games. The computer often deviated from backward induction by moving right in the first turn instead of moving down. They find that players often play corresponding to the forward induction strategy when this happens. However, this does not necessarily imply they actually applied forward induction. Their strategies could also have been caused by cardinality effects and the extent of risk aversion attributed to the computer opponent.

The data collected in Ghosh et al. (2015) has been analyzed in Ghosh & Verbrugge (online first). They perform two analyses: a latent class analysis and a theory-of-mind analysis. In their theory-of-mind analysis they find three classes: players who use zero-order theory of mind, of which there were five, players who use first-order theory of mind, of which there were twenty-seven, and players who use second-order theory of mind, of which there were sixteen. Their latent class analysis was performed on the same set of participants. They found three types of players in their latent class analysis: *expected* players, who played in correspondence to the forward induction strategy, of which there were twenty-four, *learners*, who learned to play in correspondence to forward induction throughout the trials, of which there were nine, and *random* players, who deviated from forward induction, of which there were seventeen. Because this analysis was performed on the data from Ghosh et al. (2015), the same uncertainties arise: players may be playing in correspondence to forward induction because they are actually using forward induction, but their behaviour may also be explained by cardinality effects and the extent of risk aversion attributed to the computer.

Ghosh & Verbrugge (online first) continue by extending the logic presented in Ghosh et al. (2014) with belief operators. This allows them to express players' actual strategies in more detail.

They demonstrate this extended logic by expressing two strategies commonly seen in players in it: the *myopic* strategy and the *own-payoff* strategy. In the own-payoff strategy, a player only looks at their own payoffs at each leaf node and tries to move to the first leaf node with the highest payoff. The myopic strategy is similar to the own-payoff strategy, except that the player only looks at the current and next leaf nodes, ignoring any other future leaf nodes.

Finally, Ghosh & Verbrugge (online first) create PRIMs models of these two strategies, based on their corresponding logical formulae. They compare the models' reaction times to human reaction times, and find a good fit for the own-payoff model, but not for the myopic model. In doing so they demonstrate how the logical framework can be used to make models in a cognitive architecture, which in turn can be used to make empirical predictions.

These papers show that people do not use backward induction, but due to cardinality effects it remains to be seen whether they apply forward induction or not. To investigate whether the results of Ghosh et al. (2015) are still valid when cardinality effects are removed, Ghosh et al. (2017) replicated the experiments in Ghosh et al. (2015). Their centipede games have different payoff structures to prevent cardinality effects. These payoff structures do not include payoffs of zero. Another advantage over the games in Ghosh et al. (2015) lies in the fact that these newer games are more similar. Only the first and last leaf nodes differ across games. Nonetheless, the actions corresponding to backward and forward induction are the same in both papers. In our thesis we will use the games of Ghosh et al. (2017) and results to fit our models. In the remainder of this section, we will describe the games of Ghosh et al. (2017)

The first four of these games are depicted in Figure 2.1. In these games, C is the computer and

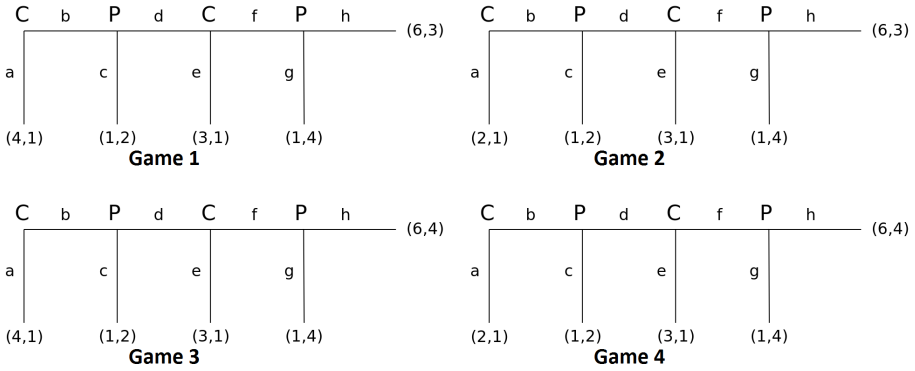

Figure 2.1: Games 1 through 4 of Ghosh et al. (2017)

P is the player. In the leaf nodes, payoffs for the computer are on the left and the player's payoffs are on the right. The differences between these games lie in the computer's payoff in the first leaf node, and the player's payoff in the last leaf node. In games 1 and 3, the computer's payoff is four in the first leaf node, whereas it is two in games 2 and 4. In games 1 and 2, the player's payoff is three in the last leaf node, whereas it is four in games 3 and 4. Ghosh et al. (2017) uses two more games, which are truncated versions of the four games in Figure 2.1. They can be found in Figure 2.2. In Figure 2.2, Game $1'$ is the same as Games 1 and 2 in Figure 2.1, except that the first node has been removed. Similarly, Game $3'$ is the same as Games 3 and 4 but with the first node removed.

For comparison, the games used in Ghosh & Verbrugge (online first) can be found in Figure 2.3 and Figure 2.4.

We continue by giving an in-depth explanation of the how to find the backward and forward induction strategies in a single game, such that the reader understands how to find the actions corresponding to backward and forward induction in the other games.

To find all sequences of actions corresponding to forward induction, we use the Iterated Conditional Dominance Procedure (ICDP) from Gradwohl & Heifetz (2011). For backward induction, we use the Backward Dominance Procedure from Gradwohl & Heifetz (2011). Due to the similarity between these procedures, we will only give an example of the ICDP. We will use Game 3 from

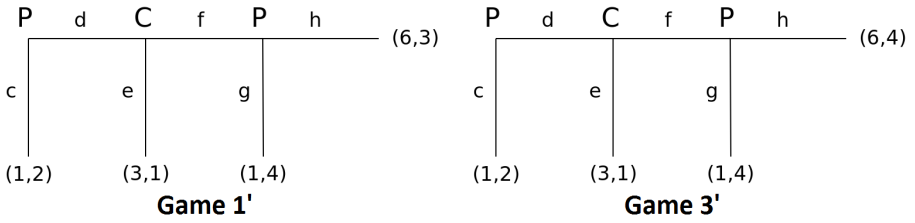

Figure 2.2: Games $1'$ and $3'$ of Ghosh et al. (2017)

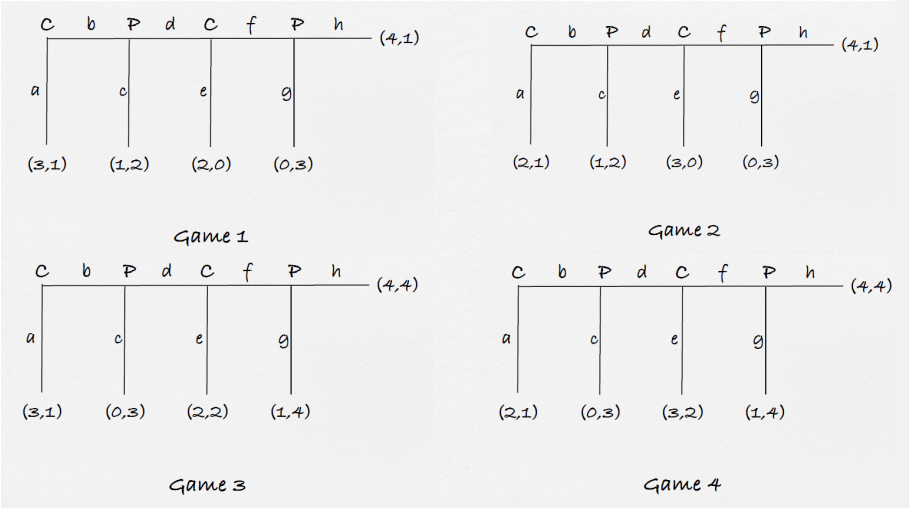

Figure 2.3: Games 1 through 4 of Ghosh & Verbrugge (online first)

Ghosh & Verbrugge (online first) as our example game, which can be found in Figure 2.3. The algorithm is as follows:

- **Initial Step:** For every decision node n let $\Phi^0(n) = S(n)$ be the full decision problem at n.
- **Inductive Step:** Let $k \geq 1$, and suppose that the decision problems $\Phi^{k-1}(n)$ have already been defined for every node n. Then for every player $i \in I$ and each decision node $n \in N_i$ delete from $\Phi_i^{k-1}(n)$ all the strategies of player i that are strictly dominated at some $\Phi^{k-1}(n')$, $n' \in N$, *unless* this would remove all the strategies in $\Phi_i^{k-1}(n)$. In the latter case, do not remove any strategies from $\Phi_i^{k-1}(n)$. The resulting reduced decision problem is denoted by $\Phi^k(n)$.

At some point no more strategies are eliminated at any node n. Denote the resulting reduced decision problem at n by $\Phi(n)$.

A strategy s_i in a game G is extensive-form rationalizable if and only if $s_i \in \Phi_i(r)$, where r is the root of the game tree. The above procedure has been copied from Gradwohl & Heifetz (2011).

Here, I is the set of players and N_i is the set of player i's decision nodes. The full decision problem at n is a tuple with for each player the set of strategies possible at that node. For example, in the first node of Game 3 this would be $(\{ae, af, be, bf\}, \{cg, ch, dg, dh\})$. In the last decision node, belonging to player P, this would be $(\{bf\}, \{dg, dh\})$, because this node can only be reached if C played according to the strategy bf, and if P played according to a strategy that includes d. A decision problem, in general, is a tuple with for each player a set of strategies.

We use as our decision node's names n_1, n_2, n_3 and n_4. If we apply the initial step to Game 3 of Ghosh & Verbrugge (online first), we obtain the following decision problems:

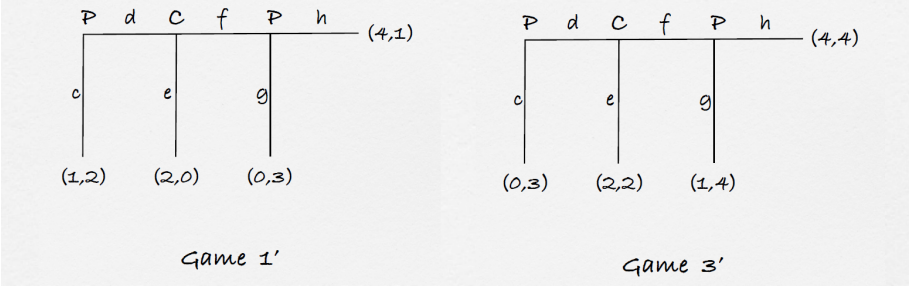

Figure 2.4: Games $1'$ and $3'$ of Ghosh & Verbrugge (online first)

$$\Phi^0(n_1) = (\{ae, af, be, bf\}, \{cg, ch, dg, dh\})$$

$$\Phi^0(n_2) = (\{be, bf\}, \{cg, ch, dg, dh\})$$

$$\Phi^0(n_3) = (\{be, bf\}, \{dg, dh\})$$

$$\Phi^0(n_4) = (\{bf\}, \{dg, dh\})$$

Now we use $k = 1$ and find the decision problems for $\Phi^1(n)$. Therefore we must find all strategies in $\Phi^0(n)$ that are *strictly dominated* in some decision problem, for some player.

A strategy s_i of player i is strictly dominated at a decision problem $D(n)$ if, assuming that players can only play according to the strategies present in the decision problem $D(n)$, for every belief player i can have about its opponent's strategy, there exists a strategy s_i' in $D(n)$ belonging to i such that the strategy $s'(i)$ yields player i a higher expected payoff than does s_i (rephrased from Gradwohl & Heifetz (2011)).

For example, in $\Phi^0(n_1)$, ae would be strictly dominated if for each of player P's strategies cg, ch, dg, *and* dh, there is a player C strategy that would give player C a higher outcome than ae.

To find the strictly dominated strategies in Game 3, we use a *payoff table*, which can be found in Table 2.1. A payoff table contains the payoffs for each combination of player C and player P strategies.

		P			
		cg	ch	dg	dh
C	ae	$(3,1)$	$(3,1)$	$(3,1)$	$(3,1)$
	af	$(3,1)$	$(3,1)$	$(3,1)$	$(3,1)$
	be	$(0,3)$	$(0,3)$	$(2,2)$	$(2,2)$
	bf	$(0,3)$	$(0,3)$	$(1,4)$	$(4,4)$

Table 2.1: Payoff table for Game 3 in Ghosh & Verbrugge (online first)

The lines in this table separate it into sections which are relevant for each of the four decision problems. This table can be verified with Figure 2.3 on page 19. If we wish to see whether a strategy is strictly dominated, we simply have to select a strategy, such as ae for player C, and then ascertain whether there is a strategy that yields a higher payoff for C for *each* of player P's strategies. If there is even one player P strategy where there is no strategy for player C that yields a higher payoff, the strategy is not strictly dominated. In this case, consider player C's belief cg. In this column, there is no strategy that yields a higher payoff than ae, which is 3, so ae is *not* strictly dominated.

However, be *is* strictly dominated at $\Phi^0(n_1)$. The strategy be yields player C payoffs of 0, 0, 2, and 2, respectively. Both ae and af always yield player C a payoff of 3, and bf yields player C a payoff of 4 under the belief that player P plays dh. Therefore, for each belief about player P's strategy, there is a player C strategy that yields player C a higher payoff than be. Therefore we must eliminate be from each decision problem in $\Phi^0(n)$.

There are no other strategies that are strictly dominated in $\Phi^0(n)$ (the reader is invited to verify this herself), so we obtain the following reduced decision problem:

$\Phi^1(n_1) = (\{ae, af, bf\}, \{cg, ch, dg, dh\})$

$\Phi^1(n_2) = (\{bf\}, \{cg, ch, dg, dh\})$

$\Phi^1(n_3) = (\{bf\}, \{dg, dh\})$

$\Phi^1(n_4) = (\{bf\}, \{dg, dh\})$

In these reduced decision problems, player C cannot play be and player P cannot believe that player C plays be. Therefore we can revise our payoff table, obtaining the one in Table 2.2.

		P			
		cg	ch	dg	dh
C	ae	$(3,1)$	$(3,1)$	$(3,1)$	$(3,1)$
	af	$(3,1)$	$(3,1)$	$(3,1)$	$(3,1)$
	bf	$(0,3)$	$(0,3)$	$(1,4)$	$(4,4)$

Table 2.2: Payoff table for Game 3 in Ghosh & Verbrugge (online first), with strategy be removed

Now consider the decision problem $\Phi^1(n_2) = (\{bf\}, \{cg, ch, dg, dh\})$. In this decision problem, player C plays according to bf, because it is the only strategy left for C. We can look at the lower half of Table 2.2 to see that dg and dh will give player P 4 points, whereas cg and ch yield 3 points for player P in $\Phi^1(n_2)$. The strategies cg and ch are strictly dominated at $\Phi^1(n_2)$ so we must eliminate cg and ch from all decision problems in $\Phi^1(n)$.

Because there are no other strategies that are strictly dominated at this stage (the reader is invited to verify this herself), we obtain the following reduced decision problem:

$\Phi^2(n_1) = (\{ae, af, bf\}, \{dg, dh\})$

$\Phi^2(n_2) = (\{bf\}, \{dg, dh\})$

$\Phi^2(n_3) = (\{bf\}, \{dg, dh\})$

$\Phi^2(n_4) = (\{bf\}, \{dg, dh\})$

At this point, there are no more strictly dominated strategies (the reader is invited to verify this herself). The root of the game tree is n_1, so the forward induction strategies are ae, af, bf, dg and dh.

The Backward Dominance Procedure is very similar to the Iterated Conditional Dominance Procedure. There is only one difference: in the Iterated Conditional Dominance Procedure, for some node n, if a strategy is strictly dominated at a decision problem $\Phi^k(n)$, it must be deleted from *all* decision problems $\Phi^k(n')$ (including $\Phi^k(n)$ itself). In the Backward Dominance Procedure, for some node n, if a strategy is strictly dominated at the decision problem $\Phi^k(n)$, it must be deleted from $\Phi^k(n)$, and from any decision problem $\Phi^k(n')$ where n' comes *before* node n.

The BI and FI strategies for the games in Ghosh & Verbrugge (online first) are the same as the BI and FI strategies for the games in Ghosh et al. (2017). In both papers, the strategy table as found in Table 2.3 on page 22 is presented.

However, we make a few notes with regard to this table. First of all, be should not be among the BI strategies for Game 3. This is because be is strictly dominated at $\Phi^0(n_1)$, as seen in the first step of our previous example. Because n_1 is the root node of the tree, be should not be in the BI strategies.

Secondly, the strategies ae and af of player C, and the strategies cg and ch for player P, always yield the same outcome, which can be verified in Table 2.1 on page 20. Because a and c both end the game, the second action in these strategies will never be played. In the rows corresponding to Games 1, 2 and $1'$ in Table 2.3 on page 22, only ae and cg are present, suggesting af and ch

	BI strategies	FI strategies
Game 1	C: a;e P: c;g	C: a;e P: d;g
Game 2	C: a;e P: c;g	C: a;e P: c;g
Game 3	C: a;e, a;f, b;e, b;f P: c;g, c;h, d;g, d;h	C: a;e, a;f, b;f P: d;g, d;h
Game 4	C: a;e, a;f, b;e, b;f P: c;g, c;h, d;g, d;h	C: a;e, a;f, b;e, b;f P: c;g, c;h, d;g, d;h
Game $1'$	C: e P: c;g	C: e P: c;g
Game $3'$	C: e;f P: c;g, c;h, d;g, d;h	C: e;f P: c;g, c;h, d;g, d;h

Table 2.3: BI and FI strategies for the games in Ghosh & Verbrugge (online first) and Ghosh et al. (2017). Actions are separated by semicolons, strategies are separated by commas.

could be eliminated in the procedure. This would imply they are strictly dominated at some point. However, if af or ch are strictly dominated, ae and cg must also be strictly dominated, because they yield the same outcome. Therefore either both strategies or neither of these strategies must be eliminated: it is impossible to separate ae from af and cg from ch.

The BI and FI strategies, according to our own calculations, can be found in Table 2.4.

	BI strategies	FI strategies
Game 1	C: a;e, a;f P: c;g, c;h	C: a;e, a;f P: d;g
Game 2	C: a;e, a;f P: c;g, c;h	C: a;e, a;f P: c;g, c;h
Game 3	C: a;e, a;f, b;f P: c;g, c;h, d;g, d;h	C: a;e, a;f, b;f P: d;g, d;h
Game 4	C: a;e, a;f, b;e, b;f P: c;g, c;h, d;g, d;h	C: a;e, a;f, b;e, b;f P: c;g, c;h, d;g, d;h
Game $1'$	C: e P: c;g, c;h	C: e P: c;g, c;h
Game $3'$	C: e, f P: c;g, c;h, d;g, d;h	C: e, f P: c;g, c;h, d;g, d;h

Table 2.4: BI and FI strategies for the games in Ghosh & Verbrugge (online first) and Ghosh et al. (2017), second calculation. Actions are separated by semicolons, strategies are separated by commas.

However, it has to be noted that Ghosh & Verbrugge (online first) appear to be aware of the equivalence of ae and af, as they state that there is only one unique outcome in Games 1, 2 and $1'$, namely C playing a and ending the game immediately. Due to the equivalance of ae and af and cg and ch, omitting af and ch may be seen a simplification for the reader.

2.2 Logic

In the current section we describe the formal logic used to describe marble drop in Ghosh & Verbrugge (online first). This logic is an adaptation of the logic introduced in Ghosh et al. (2014). Most of this section has been adapted from Section 2 of Ghosh & Verbrugge (online first), but we provide some additional information for readers who are less proficient in logic. However, we do assume that the reader has some basic knowledge of logic and set theory.

Representing centipede games In this formal logic, $N = \{C, P\}$ is the set of players. The notation i is used to denote a player, and $\bar{\imath}$ to denote i's opponent. In this case, $\overline{C} = P$ and $\overline{P} = C$. The set Σ is a finite set of actions, where a and b range over Σ (that is, a and b are variables that can bind to any element in Σ). Lastly, suppose we have a set X and a finite sequence $\rho = x_1x_2...x_m \in X^*$. Then $last(\rho) = x_m$ is the last element in this sequence. Here, * is the Kleene star (Kleene, 1956): If X is a set, then X^* is the set of all concatenations of the elements in X (including the empty concatenation λ). For example, if $X = \{a, b, c\}$, then $X^* = \{\lambda, a, b, c, aa, ab, ac, ba, bb, ...\}$. For empty concatenations, $last(\lambda) = \emptyset$.

Let $\mathbb{T} = (S, \Rightarrow, s_0)$ be a tree where S is a set of vertices (which are the choice points and leaf nodes in our games). The function $\Rightarrow$: $(S \times \Sigma) \to S$ is a partial function specifying the edges, or actions, of the tree. Here, $\times$ is the Cartesian product of sets, which results in ordered pairs of the elements of both sets. For example, $\{a, b\} \times \{c, d\} = \{(a, c), (a, d), (b, c), (b, d)\}$. In our case these will be node-action pairs. Because $\Rightarrow$ is a partial function, a subset of $(S \times \Sigma)$ may be used. In the case of centipede games, we omit any pairs containing leaf nodes and we only use those node-action pairs (s, a) where a can be played at s. So, $\Rightarrow$ specifies for each of these node-action pairs (s, a) which node is reached when a is played at s. The element s_0 is the root node of the tree.

For a node $s \in S$, $\vec{s} = \{s' \in S \mid s \stackrel{a}{\Rightarrow} s' \text{ for some } a \in \Sigma\}$. Or, $\vec{s}$ is the set of all nodes that can be reached by playing some action a at s. A node s is called a leaf node if $\vec{s} = \varnothing$, that is, s is a leaf node if no other nodes can be reached from it.

A tree $\mathbb{T}$ is said to be finite if S is a finite set, or, the tree is finite if it has a finite number of nodes.

An extensive-form game tree $T = (\mathbb{T}, \widehat{\lambda})$ is a pair where $\mathbb{T}$ is a tree (which has been previously explained) and $\widehat{\lambda} : S \to N$ is a turn function which maps each node in the game tree to a player. Even though only non-leaf nodes need labelling, Ghosh & Verbrugge (online first) opted to keep labelling for leaf nodes for the sake of uniform representation. For a player $i \in N$, one defines $S^i = \{s \mid \widehat{\lambda}(s) = i\}$, that is, S^i is the set of all nodes belonging to player i. The set $frontier(\mathbb{T})$ is the set of all leaf nodes in $\mathbb{T}$.

An extensive-form game tree $T = (\mathbb{T}, \widehat{\lambda})$ is finite if $\mathbb{T} = (S, \Rightarrow, s_0)$ is finite, which, as we have previously seen, is finite if S is finite. Therefore an extensive-form game tree is finite if it has a finite number of nodes.

Strategies A strategy for some player i is a function $\mu^i : S^i \to \Sigma$ which specifies a move at every node where i has a turn. For a player $i \in N$, the notation μ^i is used for i's strategy, often abbreviated μ, and $\tau^{\bar{\imath}}$ for i's opponent's strategy, often abbreviated τ. A strategy μ can also be seen as a subtree of T where for nodes belonging to i, there is a unique outgoing edge, and for nodes belonging to $\bar{\imath}$, all outgoing edges are included. For example, if we take Game 1 from Figure 2.1, and consider player P's strategy of playing d in his first node and g in his second node, we would obtain the strategy tree as seen in Figure 2.5.

Ghosh & Verbrugge (online first) formally define a strategy tree, recursively, as follows: for a player $i \in N$ and his strategy $\mu^i : S^i \to \Sigma$, the strategy tree $T_\mu = (S_\mu, \Rightarrow_\mu, s_0, \widehat{\lambda}_\mu)$ associated with μ is the least subtree of T satisfying the following property:

- $s_0 \in S_\mu$
- For any node $s \in S_\mu$
  - if $\widehat{\lambda}(s) = i$ then there exists a unique $s' \in S_\mu$ and action a such that $s \stackrel{a}{\Rightarrow}_\mu s'$, where $\mu(s) = a$ and $s \stackrel{a}{\Rightarrow} s'$.
  - if $\widehat{\lambda}(s) \neq i$ then for all s' such that $s \stackrel{a}{\Rightarrow} s'$, we have $s \stackrel{a}{\Rightarrow}_\mu s'$.
- $\widehat{\lambda}_\mu = \widehat{\lambda} \restriction S_\mu$

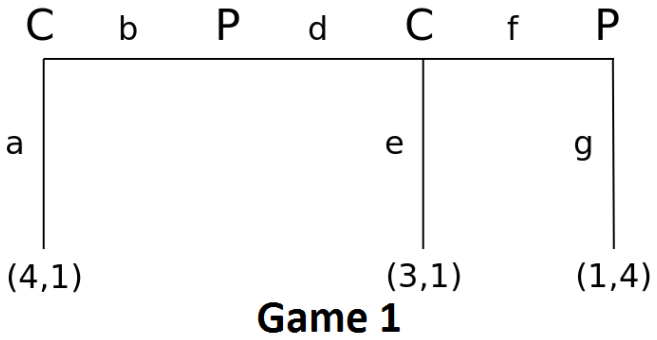

Figure 2.5: A subtree of Game 1 of Ghosh et al. (2017)

In words: the root node of the game tree is always in the strategy tree. From the root node, edges and nodes are recursively added. If a node belongs to the opponent, both outgoing edges and the next nodes are added. If a node belongs to player i, one outgoing edge (the one corresponding to his strategy), as well as the node that is followed by it, is added. The symbol $\restriction$ restricts a function to a subset of its domain.[1] From this property, S_μ is the set of nodes relevant to the strategy tree, $\Rightarrow_\mu$ is the set of edges (dependent upon which strategy is used), s_0 is the root node, and $\widehat{\lambda}_\mu$ is the turn function for those nodes in the strategy tree.

They then let $\Omega^i(T)$ denote the set of all strategies for player i in the extensive-form game tree T. In Game 1 (see Figure 2.1), all strategies for player C are $a; e, a; f, b; e$ and $b; f$. Then, a play $\rho : s_0 a_0 s_1 \ldots$ is said to be *consistent* with μ if for all $j \geqslant 0$, we have that $s_j \in S^i$ implies $\mu(s_j) = a_j$. Or, "for all nodes and actions in the play, if a node s_j is in player i's nodes, then the action a_j is prescribed by strategy μ at s_j".

A pair (μ, τ) is called a *strategy profile* which consists of a pair of strategies, one for each player.

Partial strategies A partial strategy for a player i is a strategy that specifies an action at some, but not necessarily all, of player i's nodes. For example, a partial strategy for player P could be to play d at his first decision node without specifying what to do at his second node. A partial strategy is a function $\sigma^i : S^i \rightharpoonup \Sigma$ which maps some nodes s to an action a. Here, $\rightharpoonup$ denotes a *partial function*. The notation $\mathfrak{D}_{\sigma^i}$ is used to denote the domain of the partial function σ^i, that is, $\mathfrak{D}_{\sigma^i}$ is the set of possible input values in S^i for the function σ^i. The notation σ^i will be used for i's partial strategies, and $\pi^{\bar{\imath}}$ for i's opponent's partial strategies. Superscripts are omitted when unnecessary. A partial strategy σ can also be seen as a subtree of T where for some nodes belonging to i, there is a unique outgoing edge. For all other nodes, every outgoing edge is included. For example, player P's strategy of playing g at his second decision node in Game 1 of Ghosh et al. (2017) can be found in Figure 2.6. Note that both actions c and d are still enabled. A partial strategy can be seen as a set of total strategies. Consider the previous example in Figure 2.6. Here P's strategy is to play g, which may be viewed as the set of strategies $c; g$ and $d; g$.

Given a partial strategy tree $T_\sigma = (S_\sigma, \Rightarrow_\sigma, s_0, \widehat{\lambda}_\sigma)$, a set of trees $\widehat{T_\sigma}$ of total strategies can be defined as follows: a tree $T = (S, \Rightarrow, s_0, \widehat{\lambda}) \in \widehat{T_\sigma}$ if and only if

- if $s \in S$ then for all $s' \in \vec{s}, s' \in S$ implies $s' \in S_\sigma$
- if $\widehat{\lambda}(s) = i$ then there exists a unique $s' \in S$ and action a such that $s \overset{a}{\Rightarrow} s'$.

In words: if a node is in this tree, then if a node that follows it is in the tree, it is in the partial strategy tree. All nodes in this tree T must come from the partial strategy tree. Furthermore, if node s belongs to player i, then there is a unique node that follows it as well as an action that can be played to reach this node.

[1] Appendix F on page 108 contains a list of relatively uncommon LaTeX symbols used in this thesis, such as $\restriction$. Hopefully this appendix will prove to be a useful tool for other students working on this topic.

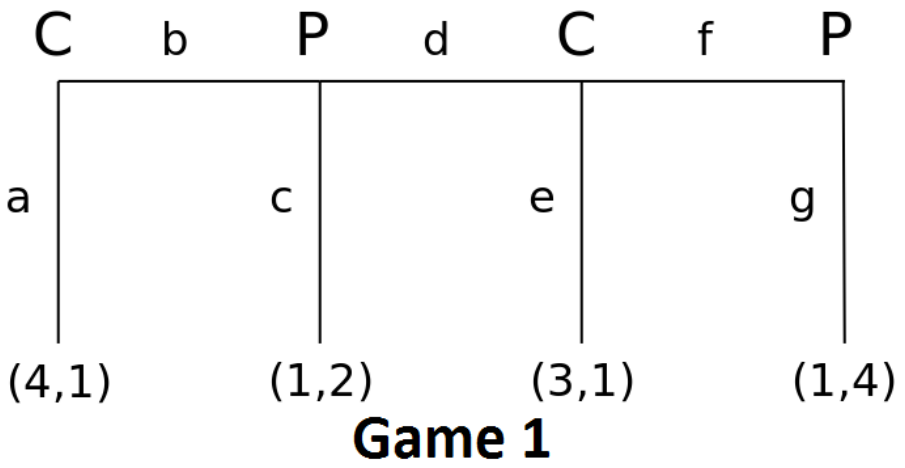

Figure 2.6: A partial strategy for player P in Game 1 of Ghosh et al. (2017)

By construction, $\widehat{T_\sigma}$ is the set of all total strategy trees for player i that are subtrees of the partial strategy tree T_σ for i. Any total strategy can also be viewed as a partial strategy, where the corresponding set of total strategies becomes a singleton set. For example, Figure 2.5 on page 24 also depicts a partial strategy for player P, where the set of total strategy trees only contains the tree in Figure 2.5. This simply shows that all total strategies are partial strategies, but not all partial strategies are total strategies.

Syntax for extensive-form game trees Ghosh & Verbrugge (online first) then continue by building a syntax for game trees. This syntax is used to parametrize the belief operators introduced later, such that one can distinguish between belief operators at different nodes of the game tree. $N = \{C, P\}$ is used as the set of players, where i and $\bar{\imath}$ range over the set N. Σ denotes a finite set of actions, and a and b range over Σ. Since we have explained these items before (see page 23), we will not do so again.

Now, *Nodes* is a finite set. The syntax for specifying finite extensive-form game trees is as follows:

$$\mathbb{G}(Nodes) ::= (i, x) \mid \Sigma_{a_m \in J}((i, x), a_m, t_{a_m})$$

where $i \in N$, $x \in Nodes$, $J(\text{finite}) \subseteq \Sigma$, and $t_{a_m} \in \mathbb{G}(Nodes)$.

Note that within $\Sigma_{a_m \in J}$, Σ denotes a formal sum and *does not* denote the set of actions. The notation '::=' can be translated as '*is recursively defined as*'. The symbol '|' is used as '*or*'. There are two options: first of all, $\mathbb{G}(Nodes)$ can be a pair (i, x) where x is a node and i is a player. This is a leaf node (recall that leaf nodes were also player-labelled). Secondly, $\mathbb{G}(Nodes)$ can be a formal sum of triples, where the first item in such a triple is always a pair (i, x) consisting of a player and the node's label, the second is always an action, and the third is either a pair (i, x) or another sum of triples. In short, such a sum of triples is simply a non-leaf node. Each item in this sum corresponds to one of the actions that can be played at this non-leaf node.

To clarify, consider the game tree as found in Figure 2.7 on page 26.

The only relevant player is P, and the relevant actions are g and h. We use P_1, l_1, and l_2 as names for the nodes. Using the previously defined syntax, we can represent this game tree as follows. Note that we use P as player label for the leaf nodes:

$$((P, P_1), g, (P, l_1)) + ((P, P_1), h, (P, l_2))$$

Given $h \in \mathbb{G}(Nodes)$, a tree T_h generated inductively by h is defined as follows:

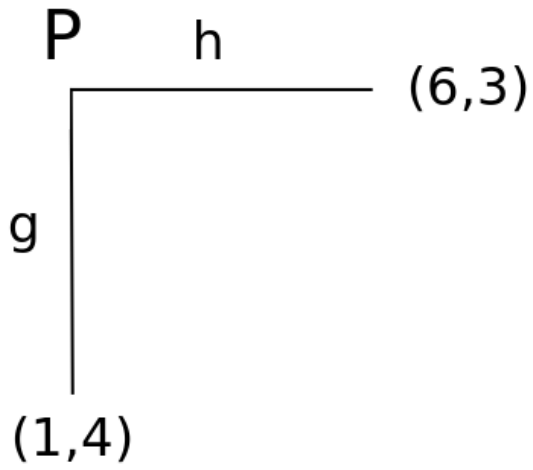

Figure 2.7: A small example game tree adapted from Game 1 of Ghosh et al. (2017)

- $h = (i, x) : T_h = (S_h, \Rightarrow_h, \widehat{\lambda}_h, s_x)$ where $S_h = \{s_x\}, \widehat{\lambda}(s_x) = i$.
- $h = ((i, x), a_1, t_{a_1}) + \ldots + ((i, x), a_k, t_{a_k})$: Inductively we have trees $T_1, \ldots, T_k$ where for $j : 1 \leqslant j \leqslant k, T_j = (S_j, \Rightarrow_j, \widehat{\lambda_j}, s_{j,0})$.

  Define $T_h = (S_h, \Rightarrow_h, \widehat{\lambda_h}, s_x)$ where
  - $S_h = \{s_x\} \cup S_{T_1} \cup \ldots \cup S_{T_k}$;
  - $\widehat{\lambda_h}(s_x) = i$ and for all j, for all $s \in S_{T_j}, \widehat{\lambda_h}(s) = \widehat{\lambda_j}(s)$;
  - $\Rightarrow_h = \bigcup_{j:1 \leqslant j \leqslant k}(\{(s_x, a_j, s_{j,0})\} \cup \Rightarrow_j)$.

  In words: if h is a leaf node, the tree consists of just this leaf node (including its edge function and turn function). If h is a non-leaf node, and therefore is a sum of triples, create a tree for each item in this sum. Then, add the current node to all nodes in these trees, add the turn function corresponding to the current node to all turn functions in these trees, and add the edge function of the current node to all edge functions in these trees.

  Since $\Rightarrow$ is not only a relation but also a function ($S \times \Sigma \to S$), the following notation, which is more in line with the notation used for $\widehat{\lambda_h}$, may be clearer:

- For all j, $\Rightarrow_h (s_x, a_j) = s_{j,0}$ and for all j, for all $(s, a) \in S_{T_j} \times \Sigma_{T_j}, \Rightarrow_h (s, a) = \Rightarrow_j (s, a)$.

  Lastly, given $h \in \mathbb{G}(Nodes)$, $Nodes(h)$ is used to denote the set of distinct pairs (i, x) that occur in the expression of h. In our example in Figure 2.7, this would be $\{(P, P_1), (P, l_1), (P, l_2)\}$.

2.2.1 Specifying strategies

Ghosh & Verbrugge (online first) provide the syntax and semantics required to specify strategies within their logic. First of all, $BPF(X)$ is defined: for any countable set X (a *countable* set is a set that is bijective to a subset of the natural numbers, a set is *bijective* to another set if each element of the first set pairs with exactly one element of the second set, and each element of the second set pairs with exactly one element of the first set, and there are no unpaired elements in either set), $BPF(X)$ is the set of formulae given by the following syntax:

$$BPF(X) ::= x \in X \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid \langle a^+\rangle\psi \mid \langle a^-\rangle\psi,$$

where $a \in \Sigma$, a countable set of actions. BPF is short for "the boolean, past and future combinations of the members of X". In words, a formula in $BPF(X)$ is either an element in X, or a formula constructed from a formula in $BPF(X)$ using negation, disjunction, or one of the $\langle a^+\rangle$

and $\langle a^- \rangle$ operators. Note that negation and disjunction can be used to construct any formula in propositional logic. Formulae in $BPF(X)$ are interpreted at game positions. The operator $\langle a^+ \rangle \psi$ means "there is an outgoing action a at the current node, and if we follow it to the next node, ψ holds at that node". The operator $\langle a^- \rangle \psi$ means "there is an incoming action to the current node, and if we follow it backwards to the previous node, ψ holds at that node". These operators can be used iteratively. For example, if we consider Game 1 in Figure 2.2, and we are at player P's first node, the formula $\langle d^+ \rangle \langle h^+ \rangle \psi$ would state that ψ holds at player P's second node.

$Bool(X)$ is used to denote just the boolean formulae in $BPF(X)$, without the $\langle a^+ \rangle$ and $\langle a^- \rangle$ operators:

$$Bool(X) ::= x \in X \mid \neg\psi \mid \psi_1 \vee \psi_2.$$

For each $h \in \mathbb{G}(Nodes)$ and $(i, x) \in Nodes(h)$, a new operator to the syntax of $BPF(X)$ is added: $\mathbb{B}_h^{(i,x)}$. The resulting set of formulae is denoted as $BPF_b(X)$. The notation $\mathbb{B}_h^{(i,x)} \psi$ can be read as "in the game tree h, player i believes at node x that ψ holds".

$$BPF_b(X) ::= x \in X \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid \langle a^+ \rangle \psi \mid \langle a^- \rangle \psi \mid \mathbb{B}_h^{(i,x)} \psi.$$

Syntax Ghosh & Verbrugge (online first) present the syntax required to formulate strategies. The set $P^i = \{p_0^i, p_1^i, \ldots\}$ is used as a countable set of observables (dynamic variables that can be measured) where $i \in N$ (i is in the set of players) and $P = \bigcup_{i \in N} P^i$, that is, P is the union of each player's P^i. Two kinds of propositional variables are added to this set of observables: $(u_i = q_i)$ to denote "player i's payoff is q_i", and $(r \leqslant q)$ to denote "the rational number r is less than or equal to the rational number q", which can be used to compare payoffs in strategy specifications.

The syntax of strategy specifications is as follows:

$$Strat^i(P^i) ::= [\psi \mapsto a]^i \mid \eta_1 + \eta_2 \mid \eta_1 \cdot \eta_2,$$

where $\psi \in BFP_b(P^i)$. In words, a strategy is one of three things: a formula $[\psi \mapsto a]^i$, which means "player i has the following strategy: if ψ holds, play a", or a combination of strategies η_1 and η_2 using the operators $+$ and $\cdot$. The formula $\eta_1 + \eta_2$ means "the strategy of player i conforms to either η_1 or η_2, or both". The formula $\eta_1 \cdot \eta_2$ means "the strategy of player i conforms to both η_1 and η_2". In these formulae, ψ is either a payoff $(u_i = q_i)$, a comparison $(r \leqslant q)$, or a formula constructed from other formulae using negation $\neg$, disjunction $\vee$, the edge operators $\langle a^+ \rangle$ and $\langle a^- \rangle$, and the belief operator $\mathbb{B}_h^{(i,x)}$. It is important to note that in the strategy specification $[\psi \mapsto a]^i$, player i must play action a when ψ holds, but when ψ does not hold, player i is free to choose any possible action.

Semantics Ghosh & Verbrugge (online first) consider perfect-information games with belief structures as models. The model $M = (T, \{\longrightarrow_i^x\}, V)$ where $T = (S, \Rightarrow, s_0, \widehat{\lambda}, \mathcal{U})$. The object $(S, \Rightarrow, s_0, \widehat{\lambda})$ is an extensive-form game tree. The utility function $\mathcal{U} : frontier(T) \times N \to \mathbb{Q}$ maps each combination of leaf nodes and players to a payoff. For each node $s_x \in S$ where the turn function $\widehat{\lambda}(s_x) = i$, there is a binary relation $\longrightarrow_i^x$ over the set of nodes S. A binary relation over S is a collection of ordered pairs of elements in S. These ordered pairs are presumably of the type $\langle s_x, s_y \rangle$, where s_y can be reached by playing some action at s_x. Lastly, $V : S \to 2^P$ is a valuation function. The *powerset* of P is denoted by 2^P. The powerset of P is the set of all (inclusive) subsets of P. Recall that P contained payoffs $(u_i = q_i)$ and comparisons $(r \leqslant q)$. The valuation function V maps each node s to the set of payoffs and comparisons that are true in said node. For example, suppose $P = \{x, y, z\}$. In this case, $2^P = \{\emptyset, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$. Now suppose x and y are true in s_x, but z is not. In this case, V maps s_x from S to $\{x, y\}$ in 2^P.

The truth value of a formula $\psi \in BFP_b(P)$ at a state (or node) s, denoted $M, s \models \psi$, is defined inductively as follows:

1. $M, s \models p$ iff $p \in V(s)$ for atomic formulae $p \in P$.
2. $M, s \models \neg\psi$ iff $M, s \not\models \psi$.
3. $M, s \models \psi_1 \vee \psi_2$ iff $M, s \models \psi_1$ or $M, s \models \psi_2$.

4. $M, s \models \langle a^+ \rangle \psi$ iff there exists an s' such that $s \stackrel{a}{\Rightarrow} s'$ and $M, s' \models \psi$.

5. $M, s \models \langle a^- \rangle \psi$ iff there exists an s' such that $s' \stackrel{a}{\Rightarrow} s$ and $M, s' \models \psi$.

6. $M, s \models \mathbb{B}_h^{(i,x)}$ iff the underlying game tree of T_m is the same as T_h and for all s' such that $s \longrightarrow_i^x s', M, s' \models \psi$ in model M at state s.

In short, a formula is true if it is one of the following: (1) it is an atomic formula in $V(s)$, (2) it is negated and the remainder is not true, (3) it consists of a disjunction between two formulae and at least one of them is true, (4) it is of the type $\langle a^+ \rangle \psi$ and ψ is true in a next node after following edge a, (5) it is of the type $\langle a^- \rangle \psi$ and ψ is true in a previous node after backtracking over edge a, or (6) if it is a belief formula and if the belief's game tree corresponds to the actual game tree and the believed formula is true in each node that can be reached from the current node s.

There are two new propositions, also with accompanying truth definitions:

1. $M, s \models (u_i = q_i)$ iff $\mathcal{U}(s, i) = q_i$.

2. $M, s \models (r \leqslant q)$ iff $r \leqslant q$ where r and q are rational numbers.

In words: (1) a payoff u_i is indeed equal to q_i if the payoff function $\mathcal{U}$ says so, and (2) $(r \leqslant q)$ is true if r is equal to or smaller than q and r and q are both rational numbers.

Ghosh & Verbrugge (online first) interpret strategy specifications on strategy trees of T. Two special propositions $\mathbf{turn}_1$ and $\mathbf{turn}_2$ are added, which specify which player's turn it is in the current node s. The valuation function satisfies the property

– for all $i \in N$, $\mathbf{turn}_i \in V(s)$ iff $\widehat{\lambda}(s) = i$.

In words: $\mathbf{turn}_i$ is in the valuation function $V(s)$ if it is player i's turn at s.

The last special proposition that is added is **root**. The proposition **root** is true if the current node s is the root node:

– $\mathbf{root} \in V(s)$ iff $s = s_0$.

Semantics for strategy specifications are also given. Given a model M and a partial strategy specification $\eta \in \mathit{Strat}^i(P^i)$, there is the semantic function $[\![\cdot]\!]_M : \mathit{Strat}^i(P^i) \to 2^{\Omega^i(T_M)}$. Here, $\Omega^i(T)$ is the set of all of player i's possible strategies in the game tree T. Furthermore, each partial strategy specification is associated with a set of total strategy trees. There is an important difference between the strategies specified with $\mathit{Strat}^i(P^i)$ and with $\Omega^i(T)$. Strategies in $\Omega^i(T)$ only specify a move at each of player i's nodes. Strategies in $\mathit{Strat}^i(P^i)$ are logical formulae which may contain beliefs or other operators previously introduced.

For any $\eta \in \mathit{Strat}^i(P^i)$, the semantic function $[\![\eta]\!]_M$ is defined inductively:

1. $[\![[\psi \mapsto a]^i]\!] = \Upsilon \in 2^{\Omega^i(T_M)}$ satisfying $\mu \in \Upsilon$ iff μ satisfies the condition that, if $s \in S_\mu$ is a player i node then $M, s \models \psi$ implies $\mathit{out}_\mu(s) = a$.

2. $[\![\eta_1 + \eta_2]\!]_M = [\![\eta_1]\!]_M \cup [\![\eta_2]\!]_M$.

3. $[\![\eta_1 \cdot \eta_2]\!]_M = [\![\eta_1]\!]_M \cap [\![\eta_2]\!]_M$.

Here, $\mathit{out}_\mu(s)$ is the unique outgoing edge in μ at s.

In words, a strategy η is one of the following, as defined by the semantic function $[\![\eta]\!]_M$: (1) $[\![[\psi \mapsto a]^i]\!]$ is a set of strategies μ_x having the property that for each player i node s_x in the strategy tree of μ, if ψ is true in this node, then the unique outgoing edge at this node is a. (2) $[\![\eta_1 + \eta_2]\!]_M$ is the union of the underlying strategies η_1 and η_2, and (3) $[\![\eta_1 \cdot \eta_2]\!]_M$ is the intersection of the underlying strategies η_1 and η_2.

2.2.2 Abbreviations and examples

Ghosh & Verbrugge (online first) continue by introducing a few more new concepts and notations. First of all, it is assumed that actions are part of the observables, so $\Sigma \subseteq P$. n_1 through n_4 are used to denote each of the four nodes in Game 1 through 4 (see Figure 2.1). Player C controls nodes n_1 and n_3, and player P controls nodes n_2 and n_4. Therefore, in Game 1, there are four belief operators: $\mathbb{B}_{g1}^{n_1,C}$, $\mathbb{B}_{g1}^{n_2,P}$, $\mathbb{B}_{g1}^{n_3,C}$ and $\mathbb{B}_{g1}^{n_4,P}$. In $\langle a^+ \rangle$, the superscript may be dropped, using $\langle a \rangle$ instead.

Ghosh & Verbrugge (online first) describe strategies for player P at node n_2. Because this node is fixed, the actions required to reach each leaf node are fixed. Therefore, they can abbreviate the formulae describing the payoff structure of the game:

$$\alpha := \langle d \rangle \langle f \rangle \langle h \rangle ((u_C = p_C) \wedge (u_P = p_P))$$

$$\beta := \langle d \rangle \langle f \rangle \langle g \rangle ((u_C = q_C) \wedge (u_P = q_P))$$

$$\gamma := \langle d \rangle \langle e \rangle ((u_C = r_C) \wedge (u_P = r_P))$$

$$\delta := \langle c \rangle ((u_C = s_C) \wedge (u_P = s_P))$$

$$\chi := \langle b^- \rangle \langle a \rangle ((u_C = t_C) \wedge (u_P = t_P))$$

The payoffs these formulae refer to can be found in Figure 2.8. The conjunction of these five

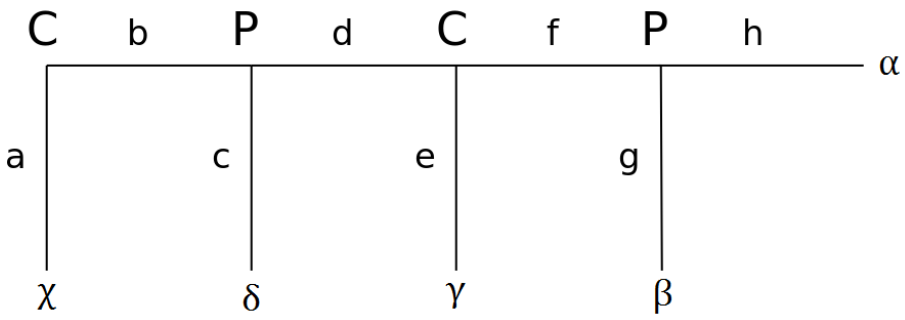

Figure 2.8: Locations of payoffs corresponding to abbreviated formulae

descriptions is defined as

$$\varphi := \alpha \wedge \beta \wedge \gamma \wedge \delta \wedge \chi.$$

Lastly, ψ_i is used to denote the conjunction of all the order relations of the rational payoffs for player $i \in \{P, C\}$ given in the game. Formally, α through χ and ψ_i are used to describe Game 1, so subscript is used when another game is considered. In Games $1'$ and $3'$, χ is not used. As an example, consider player P's payoffs in Game $3'$ in Figure 2.2. Here, his payoffs can be 2, 1, or 4. In this case, $\psi_{P3'} = (1 \leqslant 2) \wedge (1 \leqslant 4) \wedge (2 \leqslant 4)$.

Finally, we'll describe two examples given by Ghosh & Verbrugge (online first). The first of these considers the so-called *myopic* strategy. A player using the myopic strategy only looks at his current payoff, should he play down, and the payoff he would get if he plays right and his opponent plays down. This strategy is described for Game $1'$ and $3'$ as follows:

$$\mathcal{K}_P^{1'} : [(\delta_{1'} \wedge \gamma_{1'} \wedge (0 \leqslant 2) \wedge \mathbf{root}) \mapsto c]^P$$

$$\mathcal{K}_P^{3'} : [(\delta_{3'} \wedge \gamma_{3'} \wedge (2 \leqslant 3) \wedge \mathbf{root}) \mapsto c]^P$$

In Game $1'$ (in Ghosh & Verbrugge (online first), not in Ghosh et al. (2017)), a c move from P's first node leads to the payoffs $(1, 2)$, corresponding to $\delta_{1'}$. A d move followed by an e move by C leads to the payoffs $(2, 0)$, corresponding to $\gamma_{1'}$. Player P compares his payoff of 0 to his payoff of 2, which corresponds to $(0 \leqslant 2)$. The proposition **root** holds because P's first node is indeed

the root node. Lastly, the formula prescribes P to play c due to '$\mapsto c$'. In short, P compares his current and next payoffs and plays c because the first one is bigger, corresponding to the myopic strategy. Note that almost the same formulae can be used to describe the myopic strategy in Game 1 through 4, because the relevant payoffs are the same (the subscripts would have to be replaced) and the inclusion of a C node before the first P node only requires **root** to be replaced by $\langle b^{-} \rangle$**root**.

Lastly, we'll give the description of the *own-payoff* strategy from Ghosh & Verbrugge (online first). A player using the own-payoff strategy will only look at his own payoffs, ignoring his opponent's payoffs as well as any past nodes. If playing down yields a higher payoff than or an equal payoff to any payoff that can be reached by playing right, the player will play down. Otherwise, he will play right. This strategy is described for Game $1'$ and $3'$ as follows:

$$\mathcal{X}_P^{1'} : [(\alpha_{1'} \wedge \beta_{1'} \wedge \gamma_{1'} \wedge \delta_{1'} \wedge (0 \leqslant 2) \wedge (2 \leqslant 3) \wedge (1 \leqslant 2) \wedge \textbf{root}) \mapsto d]^P$$

$$\mathcal{X}_P^{3'} : [(\alpha_{3'} \wedge \beta_{3'} \wedge \gamma_{3'} \wedge \delta_{3'} \wedge (2 \leqslant 3) \wedge (3 \leqslant 4) \wedge \textbf{root}) \mapsto d]^P$$

These formulae can also be rewritten to fit Game 1 through 4 in a similar manner as the formulae corresponding to the myopic strategy. In these formulae, P considers all payoffs, and makes the comparisons corresponding to his own payoffs. **root** still ensures that the current node is the root node. If all of these hold, player P should play d. Otherwise he is free to choose any possible action.

In the next chapter, Chapter 3 on page 37 we will begin by translating these two sets of logical formulae to cognitive models as a test case.

It is important to stress that the present section has merely echoed the logic as presented in Ghosh & Verbrugge (online first). Aside from the added examples an in-depth explanations, no new concepts were introduced.

2.3 PRIMs

PRIMs, short for *primitive information processing elements*, also known as ACTransfer, is a cognitive architecture initially presented in Taatgen (2013b). Broadly speaking, PRIMs is a system that models the human mind as it performs experimental tasks. PRIMs has been used to model arithmetics (Taatgen, 2013a), counting, semantic reasoning, text editing, verbal and spatial working memory tasks and the Stroop task (Taatgen, 2013b), distraction (Taatgen, Katidioti, Borst & van Vugt, 2015), theory of mind (Wierda & Arslan, 2014), and card sorting tasks and false belief tasks (Arslan, Verbrugge & Taatgen, in press), among others. PRIMs arose from the ACT-R cognitive architecture (short for Adaptive Control of Thought - Rational). For more information on ACT-R, see Anderson (2007). Unlike in ACT-R, all processes in a PRIMs model are performed by executing *primitive elements*, which move or compare information, among others. PRIMs can be used to model transfer of skill through *production compilation*. In production compilation, primitive elements that are used in the same sequence often enough are combined into *production rules*, which may be used in a novel task. Like ACT-R, PRIMs can be used to predict reaction times, decisions, and neural activity. Most of the information in the present section is based on Taatgen (2013b) and Taatgen (2016).

2.3.1 PRIMs modules

Specific areas in the human brain have specific functions, as well as corresponding input and output. For example, the visual cortex receives, processes, and sends visual information. The motor cortex does the same for information regarding actions and movement. Because PRIMs models the human mind, it also has a set of modules with specific functions: a visual module, a manual module, a working memory module, a declarative memory module and a task control module. Different modules can work in parallel, but within a single module only one thing can be done at a time. Modules communicate with each other through their buffers. Each buffer has a

number of slots (temporary storage locations that can hold a single piece of information, such as a number or a word) that can be used to exchange information with other buffers. All the buffers together are the system's global workspace. (Taatgen, 2016)

The *visual module* receives visual input from the task being performed and places it in its slots. The visual module only sends output to the global workspace, it does not receive input from it.

The *manual module* receives actions to be performed from the global workspace and places them in its slots. Tasks and experiments implemented in PRIMs can then react to these actions, for example by giving feedback as visual input after an answer has been uttered.

The *working memory module* can be seen as a mental scratchpad. Each slot in working memory, named *WM1*, *WM2*, et cetera, can hold a single piece of information. The system can write to and read from working memory slots. Information stored in memory cannot be forgotten unless it is cleared or overwritten, but the amount of information that can be stored in *working* memory is limited. Working memory is also used to send information to declarative memory.

Before we continue by explaining the declarative memory module and the task control module, we need to introduce operators and goals.

Operators can perhaps best be compared to lines of code in computer programs. An operator consists of a set of conditions, which usually test whether some value is in some slot in some buffer, and a set of actions, which usually place values in buffer slots. While running, a PRIMs model searches for an operator that can be applied to the current values in all buffer slots. As an example, consider the following operator:

```
operator see−fish {
        V1 = fish
        V2 <> nil
==>
        say −> AC1
        V2 −> AC2
}
```

This operator can be used in a very simple task: a participant is presented with a picture of a fish, and has to state the colour of said fish. In the experiment, the type of the object is placed in *V1* and the colour of the object in *V2*. The condition $V1 = fish$ ensures the perceived object is a fish. The condition *V2 <> nil* tests whether there exists a value in the second slot of the visual buffer. The actions *say -> AC1* and *V2 -> AC2* copy this value into the second slot of the manual buffer and ensure it is said by the model. Note that this operator can also be used if the value in *V2* is not red or blue. Values can also be read from and written to the working memory, declarative memory, and goal buffer.

Operators are organized within *goals*. Goals within slots of the goal buffer are currently active goals, and operators defined in active goals are more likely to fire. Let's expand our previous example to task-switching by adding goals:

```
define goal state−fish−colour {
        operator see−fish {
                V1 = fish
                V2 <> nil
        ==>
                say −> AC1
                V2 −> AC2
        }

        operator switch−task {
                V1 = message
                V2 = switch
        ==>
                state−red−animal −> G1
        }
```

```
}
define goal state−red−animal {
        operator see−red {
                V1 <> nil
                V2 = red
        ==>
                say −> AC1
                V1 −> AC2
        }

        operator switch−task−2 {
                V1 = message
                V2 = switch
        ==>
                state−fish−colour −> G1
        }
}
```

These goals and operators can be used for the following task: a participant is presented with pictures of animals in different colours. There are two tasks: stating the colour of the animal if the animal is a fish, or stating the name of the animal if the animal is red. Only one of these tasks has to be performed at a time. If the message 'switch' appears on-screen, the participant has to switch to the other task.

The operators *see-red* and *see-fish* ensure that the model gives an answer to the task at hand. The operators *switch-task* and *switch-task-2* replace the first slot value in the goal buffer with the new goal whenever the message 'switch' appears on-screen. In a PRIMs model, one or more initial goals can be defined: these are goals that are already in the goal buffer when the model starts running.

Note that the model we described can also use working memory instead of goals to perform task-switching. It could place the values *saycolour* and *sayanimal* in *WM1* to keep track of the current task.

We continue by describing the task control module and the declarative memory module, now that we have explained operators and goals.

The *declarative memory module* handles long-term storage of information. Items stored in declarative memory are called *chunks*. A chunk has a name and a set of attribute-value pairs. For example, some chunks used to compare numbers to each other are as follows:

```
larger−2−1
        first  two
        relation bigger
        second one
        isa  comparison

notlarger−1−1
        first  one
        relation not
        second one
        isa  comparison

notlarger−1−2
        first  one
        relation not
        second two
        isa  comparison
```

A chunk's name is mostly used to make a model easier to understand. The values connected to each of a chunk's attributes refer to other chunks. The value *one* would refer to the chunk that corresponds to the concept of the number one.

Each chunk has an activation value, which determines how easily a chunk can be recalled. The higher the activation, the more likely it is that the chunk will be remembered. A chunk's activation value consists of three components: its *base-level activation*, its *spreading activation*, and *activation noise*.

A chunk's *base-level activation* is determined by how often and how recently the chunk has been used. If a chunk has been used more often and more recently, its base-level activation will be higher. The equation used to calculate base-level activation in PRIMs is the same as the one used in ACT-R, which is

$$B_i(t) = ln(\sum_k (t - t_k)^{-d})$$

(from Anderson & Schooler (1991), see also Anderson (2007); Taatgen (2016)). In this equation, t is the current time. Each point in time t_k is a moment where the chunk has previously been recalled. The parameter d is the decay parameter, which specifies how quickly the activation of a chunk decreases. It is usually set to 0.5. The subscript i is used as an index for each chunk. Consider a chunk that is recalled at 0, 2, 10, and 11 seconds. The corresponding base-level activation over the first fifteen seconds of the chunk's existence, given a decay parameter of 0.5, would look like the plot in Figure 2.9. As can be seen in Figure 2.9, a chunk's activation decreases over time after it has been recalled. This phenomenon is called *temporal decay*.

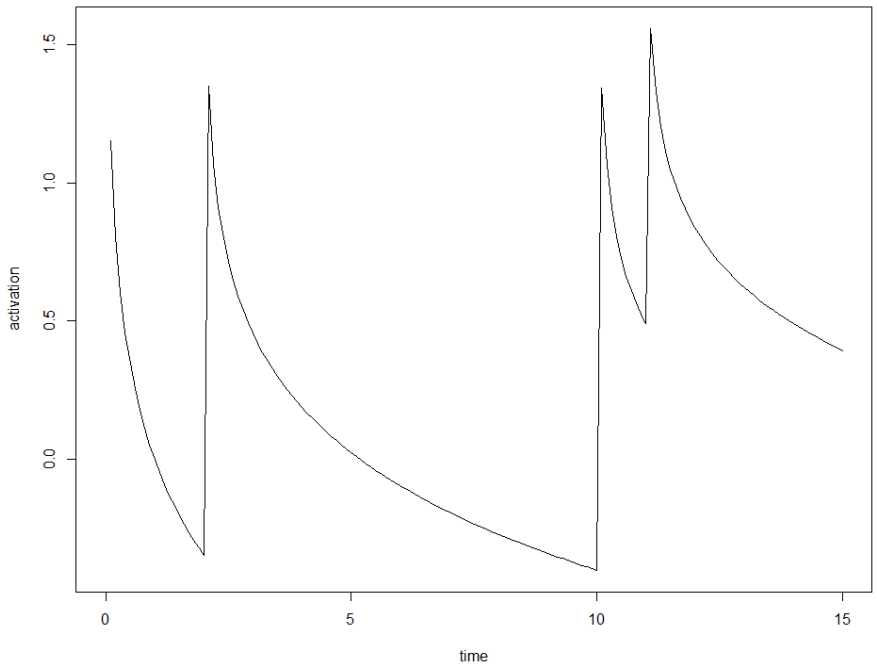

Figure 2.9: An example plot of PRIMs' base-level activation

A chunk's *spreading activation* is determined by the associative strength chunks have with each other. We will not discuss it in further detail as our models will not use spreading activation between chunks in declarative memory. For more information see Taatgen (2016).

Lastly, *activation noise* is added to or subtracted from each chunk when a recall is made.

Activation serves three purposes: being able to select between multiple relevant chunks, forgetting information, and calculating retrieval times. For more detailed information, see Taatgen (2016).

Operators are small structures of chunks, and all of them have their own activation. If multiple operators match the current buffer contents, the operator with the highest activation is used. Operators defined within one of the currently active goals receive spreading activation from that goal.

Memory retrievals are performed by placing values in slots of the declarative memory buffer. Given the previous example of chunks, a model can send a retrieval request to figure out whether two is larger than one as follows:

```
operator retrieve−two−successor {
        −conditions−
==>
        two −> RT1
        one −> RT3
        comparison −> RT4
}
```

Within the set of actions of the operator *retrieve-two-successor*, the value *two* is placed in the first slot of the declarative memory buffer. A retrieval is made for a chunk where the chunk's first attribute has the value *two*, the third attribute has the value *one*, and the fourth attribute has the value *comparison*. The only chunk that has these values is *larger-2-1*, so it is retrieved (assuming its activation is above the retrieval threshold) and its values are sent to the slots in the retrieval buffer. In this case, *RT1* will contain *two*, *RT2* will contain *bigger*, *RT3* will contain *one*, and *RT4* will contain *comparison*. The model can then use this retrieved information and, for example, state that two is bigger:

```
operator say−bigger−or−not {
        RT1 <> nil
        RT2 <> nil
        RT3 <> nil
        RT4 = comparison
==>
        say −> AC1
        RT2 −> AC2
}
```

The operator *say-bigger-or-not* can be used to state whether two numbers are bigger or not, after retrieving them. If a chunk has been retrieved that is labelled as a *comparison* and its slots are not empty, then the model states the second slot in this chunk. Note that the fourth slot value of the comparison chunks can be omitted if these are the only chunks in declarative memory. However, if there are multiple chunks of different natures *RT4 = comparison* prevents these chunks from being retrieved.

2.3.2 Production compilation

A primitive element (PRIM) is a single condition or action in an operator. In the above example, primitive elements are *RT1 <> nil* and *RT4 = comparison*, as well as every other condition and action in the operator. As can be seen, PRIMs either move, copy, or compare information. PRIMs are seen as the smallest elements of skill, and are used to model performance speed-ups caused by training (Taatgen, 2013b).

Each PRIM takes a small amount of time to be carried out. A model can become faster through *production compilation*: if two PRIMs fire together often enough, they are combined into a single production rule, which can be carried out more quickly than both individual PRIMs. This process is also applied to production rules themselves: if two production rules fire together often enough,

they are combined into one production rule, which takes less time to be carried out. PRIMs and production rules are stored in declarative memory, and the time needed to retrieve them depends on their activation. A single production rule takes less time to carry out than a set of PRIMs because only one memory retrieval has to be performed.

Production compilation has been used to model both speed-ups caused by training and transfer of skill. Transfer of skill occurs when training on one task improves proficiency in another task. For more detailed examples, see Taatgen (2016).

2.3.3 Visual representation in PRIMs

A PRIMs file contains a model, which is comprised of goals and operators, and a script, which runs the experiment and sends visual input, such as feedback, to the visual module of the PRIMs model. Visual input is sent to a PRIMs model using the function *screen*. For example, *screen("three", "plus", "five")* will send *three* to *V1*, *plus* to *V2*, and *five* to *V3*. This is analogous to displaying *three plus five* on-screen in an experiment with human participants.

More complex hierarchical structures can be displayed, where each item has a set of properties. In this case, each item on display is a list. The first item in this list is the item itself (such as *plate*). Each next item is a property of this item (such as *blue* and *large)*. After the item's properties, the next items in this list are any sub-items. If the item itself is a plate, then a sub-item could be an object on this plate (such as *chicken* and *potato*). These sub-items follow the same syntax.

To illustrate such a hierarchy, consider Figure 2.10.

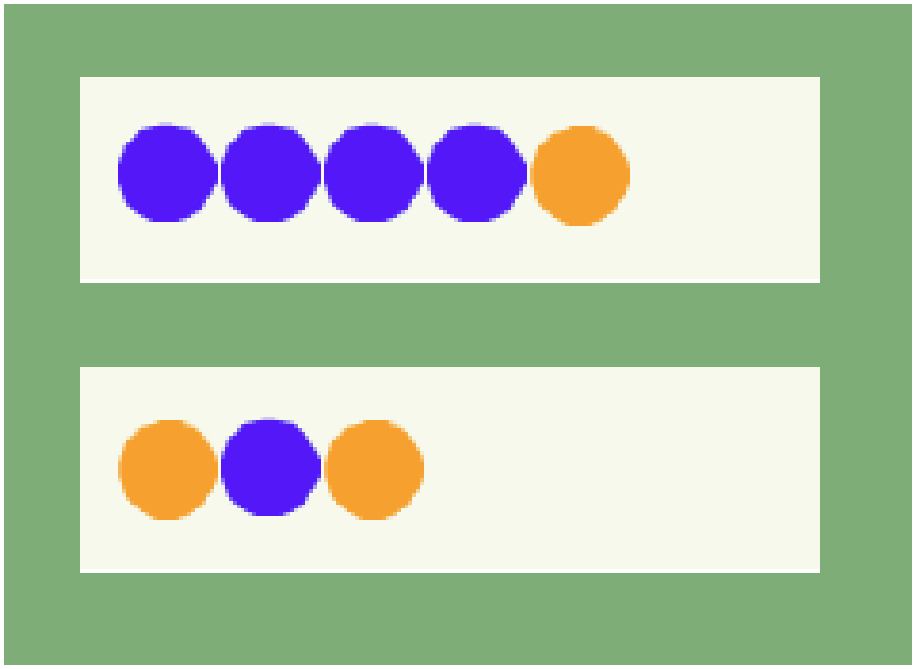

Figure 2.10: Two bins with marbles (adapted from Ghosh et al. (2017))

In Figure 2.10 there are two bins with marbles in them. This image could be represented as follows:

```
screen(
        [''bin'',''one'',
                [''marble'',''blue''],
                [''marble'',''blue''],
                [''marble'',''blue''],
                [''marble'',''blue''],
```

```
                ['marble','orange'],
        ],
        ['bin','two',
                ['marble','orange'],
                ['marble','blue'],
                ['marble','orange'],
        ]
)
```

A PRIMs model can only focus on one item at a time. When a model starts running, it will focus on the first item. In this case, the model will receive *bin* in *V1* and *one* in *V2*, and nothing else. To look at other items, it has to change focus. It can do so by sending focus actions to the manual module. For example, *focus-next -> AC1* would have to be in the actions of an operator. There are four focus actions in PRIMs:

- *focus-next* moves the focus to the next item on the current level. In the example above, both placemats are on the same level. If the focus is on the first placemat, *focus-next* moves the focus to the second placemat.
- *focus-down* moves the focus to the first sub-item of the current item. If the focus is on the first placemat, *focus-down* would move the focus to the first fork on this placemat.
- *focus-up* moves the focus up one level, and then moves the focus to the next item on this level.
- *focus-first* moves the attention back to the first item on the current level.

These actions allow a PRIMs model to look around a more complex hierarchical representation of an image. We will use such a representation and the accompanying actions to represent the game of Marble Drop starting with Chapter 3 on page 37.

Chapter 3

Translating the myopic and own-payoff models

3.1 The myopic and own-payoff strategies in logic

In Section 2.2.2 on page 29, logical formulae are given for the myopic and own-payoff strategies. The former can also be found in Section 2.2.2. We will repeat them here. The myopic strategies for Games $1'$ and $3'$ of Figure 2.4 on page 20 as represented in the logic are as follows:

$\mathcal{K}_P^{1'} : [(\delta_{1'} \wedge \gamma_{1'} \wedge (0 \leqslant 2) \wedge \mathbf{root}) \mapsto c]^P$

$\mathcal{K}_P^{3'} : [(\delta_{3'} \wedge \gamma_{3'} \wedge (2 \leqslant 3) \wedge \mathbf{root}) \mapsto c]^P$

The own-payoff strategies for Games $1'$ and $3'$ as represented in the logic are as follows:

$\mathcal{X}_P^{1'} : [(\alpha_{1'} \wedge \beta_{1'} \wedge \gamma_{1'} \wedge \delta_{1'} \wedge (0 \leqslant 2) \wedge (2 \leqslant 3) \wedge (1 \leqslant 2) \wedge \mathbf{root}) \mapsto d]^P$

$\mathcal{X}_P^{3'} : [(\alpha_{3'} \wedge \beta_{3'} \wedge \gamma_{3'} \wedge \delta_{3'} \wedge (2 \leqslant 3) \wedge (3 \leqslant 4) \wedge \mathbf{root}) \mapsto d]^P$

These formulae apply to the games as presented in Ghosh & Verbrugge (online first) (see Figure 2.4 on page 20), which differ from the games we use. The games we use are presented in Ghosh et al. (2017) and can be found in Figures 2.1 on page 18 and Figure 2.2 that immediately follows it. Furthermore, the abbreviations α through δ are shorthand for both player C *and* player P's payoffs. Because the myopic and own-payoff strategies only consider player P's payoffs, we will introduce a set of new abbreviations and use those in our formulae:

$\alpha_C := \langle d \rangle \langle f \rangle \langle h \rangle (u_C = p_C)$

$\beta_C := \langle d \rangle \langle f \rangle \langle g \rangle (u_C = q_C)$

$\gamma_C := \langle d \rangle \langle e \rangle (u_C = r_C)$

$\delta_C := \langle c \rangle (u_C = s_C)$

$\chi_C := \langle b^- \rangle \langle a \rangle (u_C = t_C)$

$\alpha_P := \langle d \rangle \langle f \rangle \langle h \rangle (u_P = p_P)$

$\beta_P := \langle d \rangle \langle f \rangle \langle g \rangle (u_P = q_P)$

$\gamma_P := \langle d \rangle \langle e \rangle (u_P = r_P)$

$\delta_P := \langle c \rangle (u_P = s_P)$

$\chi_P := \langle b^- \rangle \langle a \rangle (u_P = t_P)$

The formulae for the myopic and own-payoff strategies in Ghosh et al. (2017)'s games are as follows:

$\mathcal{K}_P^{1'} : [(\delta_{P,1'} \wedge \gamma_{P,1'} \wedge (1 \leqslant 2) \wedge \mathbf{root}) \mapsto c]^P$

$\mathcal{K}_P^{3'} : [(\delta_{P,3'} \wedge \gamma_{P,3'} \wedge (1 \leqslant 2) \wedge \mathbf{root}) \mapsto c]^P$

$\mathcal{X}_P^{1'} : [(\alpha_{P,1'} \wedge \beta_{P,1'} \wedge \gamma_{P,1'} \wedge \delta_{P,1'} \wedge (1 \leqslant 2) \wedge (2 \leqslant 4) \wedge (2 \leqslant 3) \wedge \mathbf{root}) \mapsto d]^P$

$\mathcal{X}_P^{3'} : [(\alpha_{P,3'} \wedge \beta_{P,3'} \wedge \gamma_{P,3'} \wedge \delta_{P,3'} \wedge (1 \leqslant 2) \wedge (2 \leqslant 4) \wedge \mathbf{root}) \mapsto d]^P$

Note that only the values within the comparisons have changed. Since we are interested in writing a PRIMs model that can play Games 1 through 4 of Figure 2.1 and Games $1'$ and $3'$ of Figure 2.2, we are also interested in the myopic and own-payoff strategies for games 1 through 4, as represented in the logic. As mentioned in Section 2.2.2 on page 29, we can create these formulae by changing the subscripts and **root** in the formulae we already have:

$\mathcal{K}_P^1 : [(\delta_{P,1} \wedge \gamma_{P,1} \wedge (1 \leqslant 2) \wedge \langle b^- \rangle \mathbf{root}) \mapsto c]^P$

$\mathcal{K}_P^2 : [(\delta_{P,2} \wedge \gamma_{P,2} \wedge (1 \leqslant 2) \wedge \langle b^- \rangle \mathbf{root}) \mapsto c]^P$

$\mathcal{K}_P^3 : [(\delta_{P,3} \wedge \gamma_{P,3} \wedge (1 \leqslant 2) \wedge \langle b^- \rangle \mathbf{root}) \mapsto c]^P$

$\mathcal{K}_P^4 : [(\delta_{P,4} \wedge \gamma_{P,4} \wedge (1 \leqslant 2) \wedge \langle b^- \rangle \mathbf{root}) \mapsto c]^P$

$\mathcal{X}_P^1 : [(\alpha_{P,1} \wedge \beta_{P,1} \wedge \gamma_{P,1} \wedge \delta_{P,1} \wedge (1 \leqslant 2) \wedge (2 \leqslant 4) \wedge (2 \leqslant 3) \wedge \langle b^- \rangle \mathbf{root}) \mapsto d]^P$

$\mathcal{X}_P^2 : [(\alpha_{P,2} \wedge \beta_{P,2} \wedge \gamma_{P,2} \wedge \delta_{P,2} \wedge (1 \leqslant 2) \wedge (2 \leqslant 4) \wedge (2 \leqslant 3) \wedge \langle b^- \rangle \mathbf{root}) \mapsto d]^P$

$\mathcal{X}_P^3 : [(\alpha_{P,3} \wedge \beta_{P,3} \wedge \gamma_{P,3} \wedge \delta_{P,3} \wedge (1 \leqslant 2) \wedge (2 \leqslant 4) \wedge \langle b^- \rangle \mathbf{root}) \mapsto d]^P$

$\mathcal{X}_P^4 : [(\alpha_{P,4} \wedge \beta_{P,4} \wedge \gamma_{P,4} \wedge \delta_{P,4} \wedge (1 \leqslant 2) \wedge (2 \leqslant 4) \wedge \langle b^- \rangle \mathbf{root}) \mapsto d]^P$

Each of these formulae specifies that the currently active node is player P's first node, using **root** or $\langle b^- \rangle$**root**. The myopic strategies prescribe the action c, while the own-payoff strategies prescribe d. In the myopic strategies, δ and γ are used to indicate that the player looks at his current and next payoffs, and compares these according to $(1 \leqslant 2)$. In the formulae corresponding to the own-payoff strategy, α, β, γ and δ indicate that player P looks at the current and all next payoffs. χ is not present: player P ignores past payoffs, if present. The comparisons indicate that player P compares his current payoff, 2, to each of his future payoffs. These are 1, 4 and 3 in Games $1'$, 1 and 2, and 1 and 4 in Games $3'$, 3 and 4.

3.2 The myopic and own-payoff models in PRIMs in Ghosh & Verbrugge (online first)

The myopic and own-payoff models implemented in PRIMs in Ghosh & Verbrugge (online first) are composed of four components: the model definition, the initial chunks in declarative memory, the script that runs the task and the set of goals and operators. The model definition and the initial chunks in declarative memory are the same for both models. The script that runs the task and the specific operators differ between both models. We describe each of these in the next four subsections. Note that the models presented in Ghosh & Verbrugge (online first) are not full models - they have been written to quickly validate the logic and are not full-scale models.

3.2.1 Model definition

Each PRIMs model has a *model definition* which is used to define which goals are initially in the goal buffer, which task constants exist, and what values each PRIMs parameter has. If a PRIMs parameter is not set within the model definition, it is set to the PRIMs default value, if a default exists. The model definitions for the myopic and own-payoff models are both as follows:

```
define task MyopicForwardReasoning {
    initial−goals: (find)
    goals: (compare next)
    task−constants: (bigger compare)
    imaginal−autoclear: nil
    default−activation: 1.0
    rt: −1.0
    ol: t
    batch−trace: t
}
```

MyopicForwardReasoning is the task's name. It is *ForwardReasoning* in the own-payoff model. The goal *find* is initially in the first slot of the goal buffer. The goals *compare* and *next* also exist, but they are not yet in the goal buffer. *bigger* and *compare* are task constants. Task constants are placed in slots $GC1$, $GC2$, et cetera. Any occurrence of *bigger* and *compare* in an operator is replaced by these slots. Task constants are not restricted to an operator, allowing them to be reused.

All parameters can be found in Taatgen (2016), except *batch-trace*, which is unique to the version of PRIMs we use. When set to t, it increases the amount of information logged when the model is run using a batch file.

3.2.2 Initial memory chunks

The initial chunks in declarative memory are also the same for the myopic and own-payoff models. A PRIMs model can have two scripts: a main script that runs the task, and an initialization script, which is run once at the start of a model run. The initial chunks in declarative memory are specified in the initialization script. They are created iteratively as follows:

```
define init−script {
    for i in 0 to 4 {
        for j in 0 to 4 {
            if (i > j) {
                name = "larger−" + i
                name = name + "−"
                name = name + j
                add−dm(name, i, "bigger", j)
            } else {
                name = "notlarger−" + i
                name = name + "−"
                name = name + j
                add−dm(name, i, "not", j)
            }
        }
    }
}
```

There is a chunk for each combination of two of the numbers 0 through 4. If the first number is larger than the second, the chunk's name will be *larger-i-j* (where i and j are two numbers), and its second slot value will be *bigger*. If the first number is not larger than the second, the chunk's name will be *notlarger-i-j*, and its second slot value will be *not*. The first slot contains the first number, and the third slot contains the second number. Let's give an example to demonstrate:

```
name  larger−3−1
slot 1  3
slot 2  bigger
slot 3  1
```

This chunk indicates that 3 is larger than 1. A model can retrieve whether 3 is larger than 1 by placing 3 in *RT1* and 1 in *RT3*. If the retrieval succeeds, *bigger* will be placed in *RT2*.

3.2.3 Task script

A PRIMs model always contains a script that is used to determine what should be displayed on-screen for the task at hand, and how the task should respond to the model's actions. The task script for the myopic model is as follows:

```
define script {
    params = batch−parameters()
    if (params == "NA") {
        params = [1, 0]
    }
    trial−start()
    if (params[0] == 1) {
        screen(2, 0)
        run−until−action("play")
        trial−end()
    } else {
        screen(3, 2)
        run−until−action("play")
        trial−end()
    }
}
```

On each trial, parameters are first read from a batch file. The parameters of a task execution script differ from the PRIMs parameters a model is initialized with, the latter of which can be found in the model definition in Section 3.2.1 on page 38. PRIMs parameters change the way the PRIMs cognitive architecture behaves, by, for example, increasing the amount of activation noise chunks have or decreasing the activation required to retrieve a chunk. The parameters of a task execution script change the task that is being performed, by, for example, using different games, different rewards, or different stimuli. If a batch file is not used or no parameters are specified in it, the parameters will be set to 1 and 0, respectively. The first value in the list of parameters determines which game the model will play. If it is set to 1, Game $1'$ from Ghosh & Verbrugge (online first) will be used, which can be found in Figure 2.4 on page 20. If set to any other value, Game $3'$ will be used. The function call *trial-start()* starts a trial, as well as the model, and *trial-end()* ends a trial. The function call *screen(2,0)* is used to display 2 in *V1* and 0 in *V2*, whereas *screen(3,2)* displays 3 in *V1* and 2 in *V2*. The function call *run-until-action("play")* pauses script execution until *play* is sent to *AC1*.

This script does not display the entire game tree. It only displays the information that is relevant when using the myopic strategy in Game $1'$ and $3'$. In *V1*, player P's payoff is displayed should he play c. In *V2*, player P's payoff is displayed if he plays d and player C plays e. This can be verified in Figure 2.4 on page 20.

The task script for the own-payoff model is similar, but iteratively displays two values for each comparison that the model makes. The task script for the own-payoff model can be found in appendix A on page 76.

In the own-payoff model's task script, the second parameter determines whether the model should compare his payoff when c is played to his payoffs when e or g are played, or whether the model should compare his payoff when c is played to his payoffs when e, g, or h are played. It does the latter when this parameter is set to 1.

The function call *run-until-action("focus")* pauses script execution until *focus* is placed in *AC1*. This is used to emulate the focus actions available to the model. Only the values relevant to the comparisons necessary to use the own-payoff strategy are displayed. The payoff that P gets if he plays c is always displayed in *V1*, whereas the payoffs he gets if e, g, or h are played are always

displayed in *V2*. The script does not display the entire game tree, only those payoff values relevant to the own-payoff model.

3.2.4 Goals and operators

Both the myopic model and the own-payoff model have three goals: *find*, *compare*, and *next*. The operators in the goal *find* make the model look at the 'screen' and remember the perceived values in its working memory slots. The operators in the goal *compare* are used to perform a retrieval request using the values remembered in working memory and, based on which value is bigger, act accordingly. The operators in the goal *next* are used to prepare for a next comparison, if necessary.

No more than two goals are ever in the goal buffer at once. The three goals are always placed in the goal buffer in the order *find-compare-next*. Because in the myopic model only one comparison needs to be made, the model only traverses through these goals once. In the own-payoff model, multiple comparisons have to be made, so the model traverses through these goals once for each comparison, or until the model decides to play an action.

To further clarify the models created in Ghosh & Verbrugge (online first), the full model code for the myopic model can be found in appendix A on page 76.

Each of the myopic model's operators should be self-explanatory. At the start of a model run, the current payoff value can be found in *V1* and the next payoff value in *V2*. The operator *find-location-1* places the first of these values in working memory slot *WM1* and the operator *find-location-2* places the second of these values in working memory slot *WM2*. It also changes the current goal to *compare*.

The goal *compare* contains three operators. The operator *check-largest-payoff-retrieve* sends a retrieval request to determine which of the values in working memory is bigger. If the model retrieves that the first value is bigger, the operator *check-largest-payoff-bigger* plays left (which is the same as down). If the model retrieves that the first value is not bigger, the operator *check-largest-payoff-not-bigger* plays right.

In the actions of both *check-largest-payoff-bigger* and *check-largest-payoff-not-bigger*, the value *done* is placed in the first slot of the goal buffer to ensure that no new operators will fire.

Note that this model only plays an action at the player's first decision node. It does not play an action at its second node.

The own-payoff model is almost the same as the myopic model in terms of operators. It can be found in appendix A on page 76 as well.

The goal *find* in the own-payoff model and its associated operators are identical to the ones in the myopic model. Within the goal *compare* in the own-payoff model, the operator *check-largest-payoff-retrieve* is the same as in the myopic model. However, the operator *check-largest-payoff-bigger* is different in both models. As opposed to the myopic model, the own-payoff model compares the current payoff to *every* future payoff. Therefore, if the current payoff is bigger, the model does not necessarily have to play down. There may be another future payoff that is bigger still. It can only play down if there is no future payoff that is bigger. Therefore, *focus* and *next* are placed in the first two slots of the action buffer, causing the script to display the next two values needed for comparison.

The operator *check-largest-payoff-not-bigger* does not differ between both models. The own-payoff strategy dictates that if there is a future payoff that is bigger than the current payoff, the model should play right.

Within the goal *next*, the operator *new-comparison* resets the model such that it can compare the next two payoffs.

3.3 Our myopic and own-payoff models

3.3.1 Requirements

Our work starts where Ghosh & Verbrugge (online first) left off. We begin by creating full myopic and own-payoff models and next, we use our findings to construct a general translation system. If

we want to build general models based on the concepts in Ghosh & Verbrugge (online first), they should satisfy the following requirements:

First of all, our models should have access to the complete game tree, and should use the focus actions available in PRIMs to direct their attention to different aspects of the game tree, because human participants can also see the complete game tree and also change their focus of attention between different aspects of the game tree.

It should be possible to set the computer opponent's strategy in our task script, and the model should be able to play both of participant P's moves, if necessary. The model should perform the entire task with all information available, just like a human participant would.

Because the myopic strategy and the own-payoff strategy are general strategies, which can be applied to any (in the case of the own-payoff strategy, finite) centipede game, our implementation should be general. Our new models should be able to play at least all of the six games we are interested in (which can be found in Figure 2.1 on page 2.2 and Figure 2.2 that immediately follows it). We will also use the games used in Ghosh et al. (2017) instead of those in Ghosh & Verbrugge (online first), to avoid cardinality effects.

3.3.2 Representing centipede games

Our new models have access to each node in the game tree throughout the entire task. This is in contrast with the setup introduced in Payne, Bettman & Johnson (1993), where some payoffs are covered. We give the model the same information as the human participants in Ghosh et al. (2017), who played the game of marble drop (see Figure 1.2 on page 12). To do so, we use the function *screen()* already present in PRIMs and translate the relevant games into the hierarchial representation used in PRIMs. Our representation requires the following information, as this information can also be found in the games themselves:

1. How are all nodes connected in the tree.
2. Which nodes are leaf nodes and which nodes are non-leaf nodes (non-leaf nodes are decision nodes).
3. For leaf nodes: what is player C's payoff and what is player P's payoff.
4. For non-leaf nodes: which player's turn it is.
5. For non-leaf nodes: whether the node is the currently active decision node.
6. Which node is the root node.
7. Which non-leaf node is the last non-leaf node.

The first five of these can all be found in Figure 1.2 on page 12 and Figure 2.1 on page 18 as well, and hence are visible to a human participant. The last two are not explicitly represented in any of the previously mentioned visual representations of centipede games. However, if a PRIMs model performs a *focus-up* action at the highest-level visual item on display, it gets stuck with *error* in *V1* and it will no longer be able to focus back at any previous items. Since we place the root node at this highest level, we must mark it to prevent the model from focusing up at the root node. The root node also differs from all other nodes, as it has no incoming edges. Secondly, the last non-leaf node differs from all other non-leaf nodes. Whereas each other non-leaf node has an outgoing edge to one other non-leaf node and one leaf node, the last non-leaf node has two outgoing edges to leaf nodes. The model should know this difference so it can adapt its focus actions to it.

Recall from Section 2.3.3 on page 35 that visual displays in PRIMs are hierarchically nested lists, where each list is an object on display. Each list consists of several non-list items followed by several lists. Each list is an object, in which each of these non-list items is a property of this object. Each list in this list is a sub-object of this object.

We use the root node as the object at the highest level. The lists in a node's list are any nodes that can be reached by playing an action at this node. The first of these is always the node that is reached by playing a right action, and the second of these is always the node that is reached

by playing a down action. We opt to use this order because we assume that participants, when looking for a certain leaf node, first traverse across the non-leaf nodes and then direct their gaze to the relevant leaf node. The model would accomplish this by first performing several *focus-down* actions followed by a *focus-down-last* action (which will be discussed in more detail later).

As an example, Game 1 (from Ghosh et al. (2017)) would be represented as follows:

```
screen(
["decision−node","c","notcurrent","root","notend",
["decision−node","p","current","notroot","notend",
["decision−node","c","notcurrent","notroot","notend",
["decision−node","p","notcurrent","notroot","end",["leaf",6,3],["leaf",1,4]],
["leaf",3,1]],
["leaf",1,2]],
["leaf",4,1]])
```

The first property of each item is whether it is a leaf node or a non-leaf node, where the former has the value 'leaf' and the latter has the value 'decision-node'. For leaf nodes, the second property is player C's payoff and the third property is player P's payoff. For non-leaf nodes, the second property denotes who controls the node. The third property denotes whether the node is the currently active decision node. The fourth property denotes whether the node is the root node. The fifth and last property denotes whether the node is the last node.

Because of the complexity of the display and the importance of moving focus we have added several new focus actions to the PRIMs cognitive architecture, some of which we use in our new models:

focus-up-stay moves the focus to the superitem of the current level.

focus-prev moves the focus back an item on the current level.

focus-last moves the focus to the last item on the current level.

focus-up-prev moves the focus to the superitem of the current level, and back one item on that level.

focus-down-last moves the focus down a level and to the last item on that level.

The task execution script in our models take three parameters: which game from Ghosh et al. (2017) is used, what C will play at its first node, and what C will play at its second node. It may seem odd that player C's second action is known before the game has even started, but it corresponds to the computer-controlled player C in the experiments in Ghosh et al. (2015) and Ghosh et al. (2017). Furthermore, if player P plays down, the game will end, so player C's second action is always a response to player P playing right, which can be determined beforehand. The model plays as player P. If a leaf node is reached, the game ends. If a player P node is reached, the model script will use the function call *run-until-action("play")* to pause script execution until the model places *play* in *AC1*. The model should place down or right in *AC2*. Based on the current node, the action the model places in *AC2*, and player C's preset actions, the game will either end, or the visual representation of the game will be updated. For example, if player C is set to play b and f in Game 1, and the model decides to play right at its first node, the display will be changed to:

```
screen(
["decision−node","c","notcurrent","root","notend",
["decision−node","p","notcurrent","notroot","notend",
["decision−node","c","notcurrent","notroot","notend",
["decision−node","p","current","notroot","end",["leaf",6,3],["leaf",1,4]],
["leaf",3,1]],
["leaf",1,2]],
["leaf",4,1]])
```

Note that a player C node will never be the current node. The model can deduce player C's previous actions from its own currently active node. The display immediately transitions between the two previous examples, without intermediately making the third node, player C's second node, the current node.

3.3.3 Initial memory chunks and model initialization

The initial chunks in the declarative memory of our new models are almost the same as those in Ghosh & Verbrugge (online first). Because we use Ghosh et al. (2017)'s games, where the payoff values range from 1 to 6, the values in our memory chunks also range from 1 to 6, instead of from 0 to 4. Furthermore, we place *comparison* in the fourth slot of these chunks to mark them as comparison chunks and to prevent the model from retrieving chunks that can be created when the first and third slots of working memory contain a number.

The PRIMs parameters of our new models are almost the same as those in the models of Ghosh & Verbrugge (online first). These can be found in the model definition in Section 3.2.1 on page 38. There are two differences. We use -2.0 as our retrieval threshold, which is the default value, instead of -1.0. Also, we set the parameter *default-operator-self-assoc* to 0, instead of -1.0. This is the association between an operator and itself, which prevents the same operator from firing multiple times. However, because we want the operator *not-found-current-node*, which we explain later, to fire multiple times in succession, we must set it to 0 to ensure that *not-found-current-node* can function properly.

Our task constants have expanded to those values that can be found in our new representation of the game tree: *decision-node*, *leaf*, *c*, *p*, *current*, *notcurrent*, *root* and *end*, as well as the two non-numerical values that can be retrieved from memory, *bigger* and *not*, the two actions the model can play, down and right, and the new memory value *comparison*.

There is one initial goal, *findcurrent*, and there are three non-initial goals, *findvals*, *compare*, and *preparenext*.

The myopic model and the own-payoff model are the same when it comes to the visual representation of the game, the initial chunks in memory, the model initialization, and the available goals. Their only difference lies in their operators, although they share many. We will discuss the structure of their goals and operators in the next section.

3.3.4 Goals and operators

The most important part of a PRIMs model are its goals and operators, which determine its decision-making process during a task. The myopic and own-payoff models have the same set of goals and almost the same set of operators. The structure of goals and operators shared by both models is as follows:

- Goal *findcurrent*
  - Operator *found-current-node*
  - Operator *not-found-current-node*
- Goal *findvals*
  - Operator *init-read-current*
  - Operator *read-current-leaf*
  - Operator *move-next-node*
  - Operator *find-next-node*
  - Operator *find-next-leaf*
- Goal *compare*
  - Operator *start-retrieval*

- Operator *first-bigger*
- Operator *second-bigger*

– Goal *preparenext*

- Operator *reset-game*

In Figure 3.1, it can be seen which operators are responsible for which focus actions.

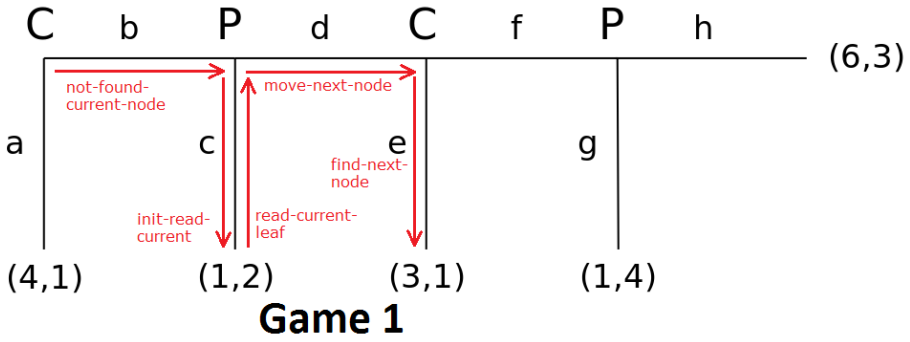

Figure 3.1: Game 1 from Ghosh et al. (2017), with focus actions in red.

The own-payoff model has two more operators within the goal *findvals*: the operators *move-next-node-end-of-tree* and *find-next-leaf-end-of-tree*. It also has one more operator within the goal *compare*, namely *first-bigger-end-of-tree*. Some of the operators are not the same across both models. We decided to use the same operator names such that the models can be compared more easily. Operators with the same name have the same function in both models, but sometimes have slight differences in implementation. Due to the similarity of both models, we first describe the myopic model and then describe the own-payoff model by describing its difference with the myopic model.

The goal *findcurrent* and its two operators are the same for both our myopic model and our own-payoff model. Whenever a model starts running, its focus is placed on the first item at the highest hierarchical level of the visual display. In our case this is the root node. The operators in the goal *findcurrent* move the focus to the currently active node. If the node currently in focus is not the active node, the operator *not-found-current-node* moves the focus down a level and to the first item on that level using a *focus-down* action. This operator may be repeated several times before the currently active node is found. If the node currently in focus *is* the currently active node, the operator *found-current-node* changes the currently active goal to *findvals*.

The goal *findvals* and its operators ensure that the relevant values are placed in working memory for comparison. We first describe it for the myopic model. The operator *init-read-current* fires first after the model focuses on the currently active node. It moves the focus to the current node's leaf node using a *focus-down-last* action. Then, the operator *read-current-leaf* stores the model's own payoff in *WM1* and moves the focus back up using a *focus-up-stay* action.

After the model has looked at the first leaf and stored its own payoff, the operator *move-next-node* moves the focus to the next node using a *focus-down* action. The operator *find-next node* then moves the focus from this node to its leaf node using a *focus-down-last* action. Once it is there, the operator *find-next-leaf* places the model's own payoff at this leaf in *WM3* and changes the currently active goal to *compare*.

As a clarification, the focus actions and the operators that execute them can be found in Figure 3.1 on page 45, using Game 1 of Ghosh et al. (2017) as an example. The focus actions we use may be substantiated using Figure 3 of Meijering, van Rijn, Taatgen & Verbrugge (2012) which can be found in Figure 3.2.

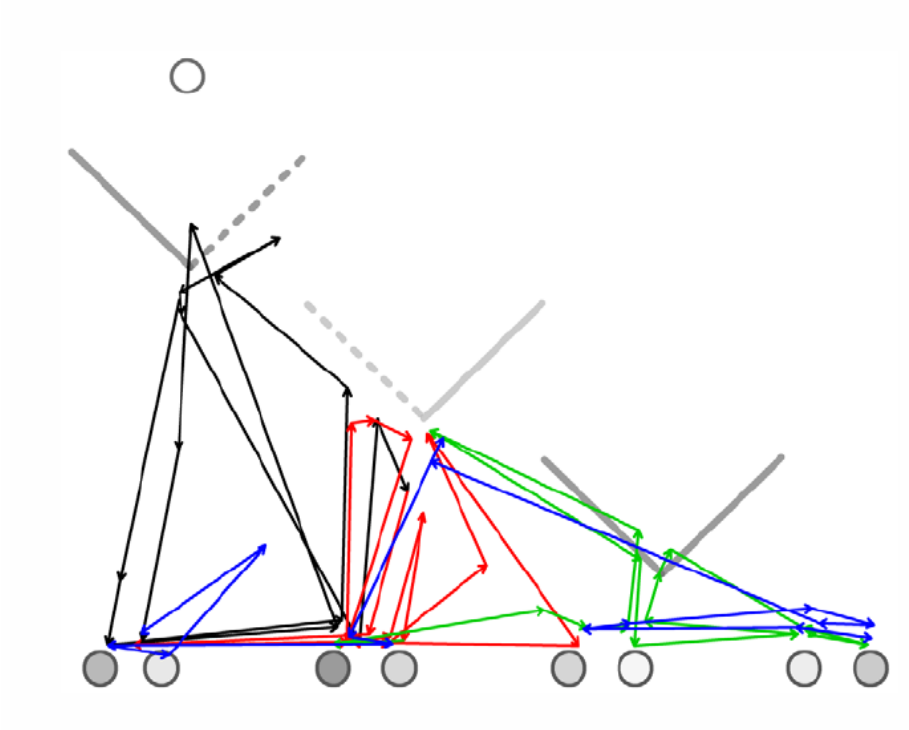

Figure 3.2: Figure 3 from Meijering et al. (2012), an example of a participant's fixations in a centipede game.

In Figure 3.2, it can be seen that the participant moves their attention between the decision points themselves, and between the decision points and bins.

The goal *findcurrent* and its operators are exactly the same for both the myopic and the own-payoff model. In the myopic model, the operators *init-read-current*, *move-next-node* and *find-next-node* all have an extra condition the versions used in the own-payoff model do not. In the myopic model, these operators test whether *current* is or is not in $V3$. The model passes two non-leaf nodes during its focus actions while it has the same contents in its working memory buffer. As seen in Figure 3.1 on page 45, it has to perform two different actions at both of these nodes. To distinguish between these nodes, it verifies whether the relevant non-leaf node is the current node or not.

The own-payoff model has to look at *all* future payoffs. Therefore it also has to look at the last leaf node. Because it has to act differently at this last leaf node, *read-current-leaf* should not fire at it. Therefore the condition $V4 = nil$ is added to *read-current-leaf*. The operator *find-next-leaf* should also not fire, so the condition $V4 = nil$ has also been added to it.

The own-payoff model has the same problem as the myopic model: while it is traversing through the nodes using focus actions, there are nodes where it has to perform a *focus-down* in one case and a *focus-down-last* in another. Because there are more than two relevant nodes, it cannot use the visual properties *current* and *notcurrent* to distinguish between these. Because of this, the model uses its second slot in the working memory buffer to remember that it already has passed the first of these nodes. It does this using the value *notcurrent*.

The operator *first-bigger* also differs in the own-payoff model. According to the own-payoff strategy, if one of the model's future payoffs is larger than its current payoff, it should play right. However, if the model's current payoff is larger than *one* of the model's future payoffs, there may still be *another* future payoff that is bigger than the model's current payoff. Therefore it has to look at *all* future payoffs before it can play down. So, if the current payoff is bigger, it continues by clearing *WM2* and *WM3* (the latter of which always holds the value of a to-be-compared future payoff) and placing *findvals* in *G1* again.

The new operator *find-next-leaf-end-of-tree* in *findvals* is similar to *find-next-leaf*. However, unlike *read-current-leaf*, it has the condition $V4 = end$, so it only fires if the leaf the model currently looking at is the rightmost leaf. It does not have the condition $WM2 = notcurrent$, because it is redundant at this leaf node. Most importantly, it places *end* in *WM2* so other operators can react accordingly.

To ensure that the model acts properly to memory retrievals when it has reached the last decision node, *first-bigger* does not fire when *end* is in *WM2*. When it is, *first-bigger-end-of-tree* fires instead, which has the same conditions except that it tests $WM2 = end$ instead. In this case it does not have to look for any more future payoffs, so it can safely play down.

The full models can be found in appendix B on page 79.

Finally, an example of a model run can be found in Table 3.1. In this example, the own-payoff model plays Game 1 against a computer that plays *b* and *f*. Double horizontal lines are used to indicate the transition to a new goal.

Operator	Notable actions
not-found-current-node	focus-down ->*AC1*
found-current-node	findvals ->*G1*
init-read-current	focus-down-last ->*AC1*
read-current-leaf	*V3* ->*WM1* focus-up-stay ->*AC1*
move-next-node	focus-down ->*AC1*
find-next-node	focus-down-last ->*AC1*
find-next-leaf	*V3* ->*WM3* focus-up-stay ->*AC1* compare ->*G1*
start-retrieval	*WM1* ->*RT1* *WM3* ->*RT3*
first-bigger	findvals ->*G1*
move-next-node	focus-down ->*AC1*
find-next-node	focus-down-last ->*AC1*
find-next-leaf	*V3* ->*WM3* focus-up-stay ->*AC1* compare ->*G1*
start-retrieval	*WM1* ->*RT1* *WM3* ->*RT3*
second-bigger	play ->*AC1* right ->*AC2* preparenext ->*G1*
reset-game	findcurrent ->*G1*
not-found-current-node	focusdown ->*AC1*
not-found-current-node	focusdown ->*AC1*
not-found-current-node	focusdown ->*AC1*
found-current-node	findvals ->*G1*
init-read-current	focus-down-last ->*AC1*
read-current-leaf	*V3* ->*WM1* focus-up-stay ->*AC1*
move-next-node-end-of-tree	focus-down ->*AC1* end ->*WM2*
find-next-leaf-end-of-tree	*V3* ->*WM3* compare ->*G1*

start-retrieval	*WM1* ->*RT1* *WM3* ->*RT3*
first-bigger-end-of-tree	play ->*AC1* down ->*AC2*

Table 3.1: Operators fired by the own-payoff model when running through Game 1

3.4 Model results

To test our new myopic and own-payoff models, we will use the same method as used in Ghosh & Verbrugge (online first). We run both models one hundred times, corresponding to one hundred virtual participants each. Fifty of these virtual participants play Game 1′, whereas the other fifty play Game 3′. Each virtual participant plays fifty rounds of a game. The times required to make the first decision and the decisions made are recorded at each round. The results are compared to the experiments with human participants in Ghosh et al. (2015) and also to the experiments with human participants in Ghosh et al. (2017). Our models play the games used in Ghosh et al. (2017).

The proportions of players who played down in Game 1′ and Game 3′ can be found in Table 3.2. The myopic model always plays down. This happens because the myopic model plays down

	Game 1′	Game 3′
Human (Ghosh et al. (2015))	0.63	0.16
Human (Ghosh et al. (2017))	0.42	0.24
Myopic	1	1
Own-payoff	0	0

Table 3.2: The proportion of human and virtual participants who played down in Game 1′ and Game 3′

whenever its current payoff is higher than the payoff it would get if it plays right and the computer opponent, C, plays down afterwards. This is always true in centipede games ("*The payoffs are arranged in such a way that at each decision point, if a player does not 'go down' to take the first possible exit and the opponent takes the next possible exit, the player receives less than if she had taken the first possible exit*", from Ghosh & Verbrugge (online first)), so a player using the myopic strategy should always play down in centipede games.

The own-payoff model always plays right. This corresponds to what the own-payoff strategy prescribes. In Game 1′ (see Figure 2.2 on page 19), the only own-payoff strategy for P is *dg*, or *right-down*. in Game 3′, the own-payoff strategies for P are *dg* and *dh*, or *right-down* and *right-right*.

Because human participants do not play down or right with proportions of 0 or 1, not all human players use the own-payoff strategy, and not all human players use the myopic strategy. Perhaps they do not use these strategies at all, using forward induction or backward induction, or something different altogether. There are more reasons to play down other than using the myopic strategy, and there are more reasons to play right other than the own-payoff strategy.

Even though our models do not resemble human participants in terms of their actions, they may resemble human participants in terms of the number of mental 'steps' required to make a decision. The myopic model looks at two payoffs and makes a single comparison, whereas the own-payoff model looks at four payoffs (in this set of games) and makes three comparisons. By looking at the difference in reaction times, we can make an informed guess about whether human participants use more, or less, information than our models. This allows us to investigate whether human participants play more like the myopic model or more like the own-payoff model.

Apparently, not all human players use the own-payoff strategy, and not all human players use the myopic strategy. Perhaps they do not use these strategies at all, using forward induction or

backward induction, or something different altogether. There are more reasons to play down other than using the myopic strategy, and there are more reasons to play right other than the own-payoff strategy.

Even though our models do not resemble human participants in terms of their actions, they may resemble humans in terms of the amount of mental 'steps' required to make a decision. The myopic model looks at two payoffs and makes a single comparison, whereas the own-payoff model looks at four payoffs (in this set of games) and makes three comparisons. By looking at the difference in reaction times we can make an informed guess about whether human participant use more, or less, information than our models.

The time required to play the first action for our models and for the human participants in Ghosh et al. (2015) and Ghosh et al. (2017) can be found in Figure 3.3.

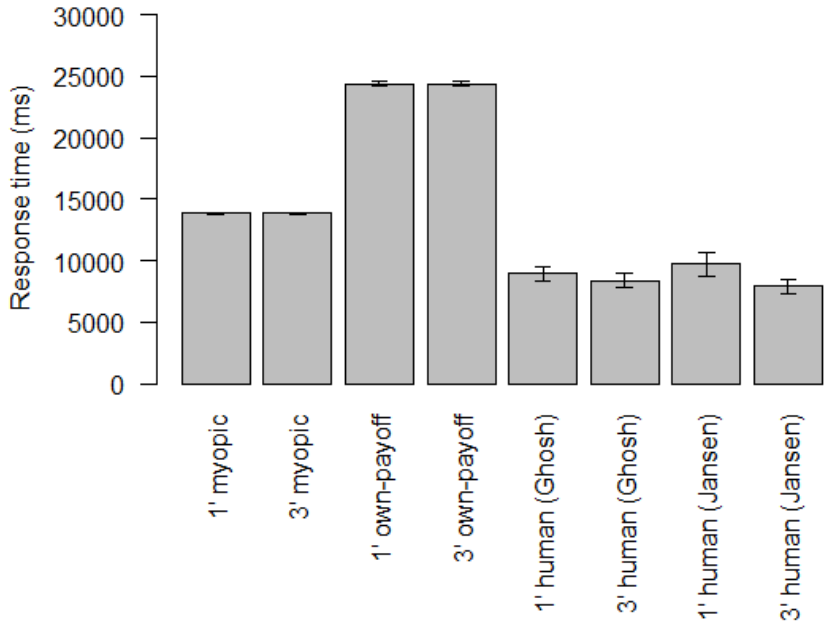

Figure 3.3: Reaction times for our myopic and own-payoff models and for the human participants in Ghosh et al. (2015) and Ghosh et al. (2017)

In Figure 3.3, it can be seen that even the myopic model is several seconds slower than the human average. Can we conclude that human participants use even less information than required for the myopic strategy? No. The own-payoff models presented in Ghosh & Verbrugge (online first) *do* give a good fit for human reaction times. Our own-payoff models are about three times slower. Therefore the slow-down in our models must be explained by something we introduced in our models that was not yet present in the models in Ghosh & Verbrugge (online first). Both models store each payoff in working memory, and both models compare payoffs stored in working memory using a retrieval from declarative memory. However, the models created in Ghosh & Verbrugge (online first) emulate gaze actions by instantly moving the model's focus from one payoff to the next, whereas our models actually look through the game to find the next payoff. Not only do these gazes require more focus actions, they also require more primitive elements to ensure that the right focus action is used at the location the model is looking at. The eye-tracking results from Meijering et al. (2012) tell us that participants seem to look at all bins in a game of marble drop before playing an action. Because even our myopic model is much slower than human

behaviour, we have to conclude that the speed of focus actions in an untrained PRIMs model does not resemble the speed of human fixations. However, it must be said that this is the first time focus actions in PRIMs have been used to predict reaction times. Focus actions have only been used in Arslan, Wierda, Taatgen & Verbrugge (2015) and related work, where they were used to model task accuracy and not reaction times. Perhaps the speed of focus actions in PRIMs can best be compared to that of very young children, who have not had as much practice as the adults who were used as participants in Ghosh et al. (2015).

3.5 Training the models

PRIMs models seem to look through a game of marble drop much more slowly than human participants. Human participants have been looking through tree-like structures such as maps all their lives, allowing them to look through the marble drop games very quickly. For the PRIMs models, the game of marble drop was the first thing they had ever seen, having no experience with looking through such tree-like structures whatsoever.

This problem can be solved for the PRIMs models by emulating the practice human participants have had. Naturally, we cannot show the PRIMs models everything a human participant has seen throughout their lives, but we can give them an introduction in looking through games of marble drop before they start playing the actual games. To do this, we create two new models in PRIMs. These models are identical to the myopic and the own-payoff models, except that they *only* look through the game tree. They do not compare payoffs by retrieving chunks from memory, and they do not make decisions about what action they should play.

These training models will play what we will call *Game 0*, which can be found in Figure 3.4. We use this game to make sure the model cannot learn anything about the payoffs of the games it will be playing.

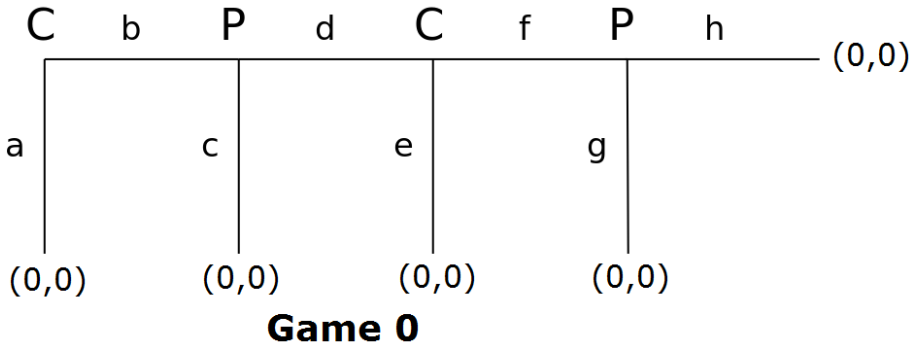

Figure 3.4: Game 0, constructed to train PRIMs models in focus actions.

The full training models can be found in appendix C on page 93.

Now we run the models again. This time we let each virtual participant run through one of the training models one hundred times before playing the actual game. Virtual participants who use the myopic strategy use the myopic training model and virtual participants who use the own-payoff strategy use the own-payoff training model. The reaction times we obtain can be found in Figure 3.5.

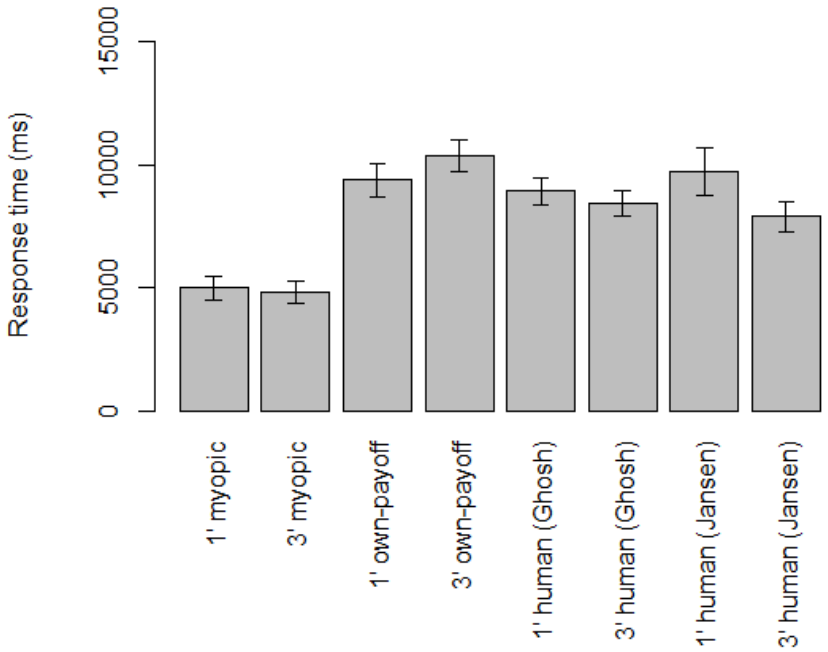

Figure 3.5: Reaction times for our myopic and own-payoff models after training on focus actions, and for the human participants in Ghosh et al. (2015) and Ghosh et al. (2017)

With one hundred training trials, the own-payoff model fits the human data quite well: there is no significant difference in mean reaction times for Game $1'$ between the own-payoff model and the human participants in Ghosh et al. (2015) (two-sided t-test with $p = 0.3208$, $t = -0.997$ and 120.5 degrees of freedom), and there is no significant difference in mean reaction times for Game $1'$ between the own-payoff model and the human participants in Ghosh et al. (2017) (two-sided t-test with $p = 0.5293$, $t = 0.62994$ and 265.85 degrees of freedom). However, one may wonder why we used one hundred training trials, and not, for example, eighty or one hundred and twenty, which is a perfectly legitimate question. In fact, we use this number because we wanted to know how many training trials would be required to obtain a good fit. If humans indeed use the own-payoff strategy (which they probably do not), then PRIMs models human performance with one hundred training trials on focus actions. However, human participants may use a much slower strategy than the own-payoff strategy and they may look around much faster than a PRIMs model that has had one hundred training trials. In order to find a number of training trials that would properly resemble the amount of skill humans already have in looking around a tree structure, we would need to know *exactly* what strategy they are using, which is a problem we would like to help solve using our PRIMs models.

Because of this, we will postpone the problem of predicting reaction times, and focus on the main goal of this thesis: translating strategies as represented in logical formulae into PRIMs models. We will use the things we learned in this chapter to make such a system in Chapter 4.

Chapter 4

A general translation method

4.1 The logic and the models

In Chapter 3 on page 37 we created PRIMs models using the myopic and own-payoff strategies by hand. In the current Section, we investigate the differences between these models and the logical formulae that correspond to the myopic and own-payoff models.

4.1.1 The myopic and own-payoff models

In Section 3.3 on page 41, we have created PRIMs models capable of playing the myopic and own-payoff strategies in centipede games. They are based on the intuitive myopic and own-payoff strategies, as is also the case for the logical formulae in Ghosh & Verbrugge (online first) (as found in Section 2.2.2 on page 29). However, our models in Section 3.3 are not based on these formulae themselves.

According to the *myopic* strategy, a player using it should end the game immediately if doing so yields a higher payoff than what he would get if he does not end the game and his opponent ends the game at the next turn. If ending the game immediately yields a player a lower payoff than if he does not end the game and his opponent ends the game in the next turn, then he should not end the game.

According to the *own-payogg* strategy, a player using it should simply play those actions that are needed to reach the highest future payoff, even if his opponent may not help him in moving towards that payoff.

According to the myopic strategy, a player using it should end the game immediately if doing so yields a higher payoff than what he would get if he does not end the game, and his opponent ends the game at the next turn. If ending the game immediately yields a player a lower payoff than if he does not end the game, and his opponent ends the game in the next turn, then he should not end the game.

According to the own-payoff strategy, a player using it should simply play those actions that are needed to reach the highest future payoff, even if his opponent may not help him in moving towards that payoff.

The PRIMs models we created in Section 3.3 on page 41 can use these strategies to play Games 1 through 4, $1'$, and $3'$ from Ghosh et al. (2017), which can be found in Figure 2.1 on page 18 and Figure 2.2 on page 19.

The models we created use PRIMs' built-in focus actions to emulate how a human participant would gaze through a game. However, because an untrained PRIMs model has no experience with focus actions, we found that our PRIMs models are too slow, and require training before their reaction times approximate those of human players.

We created these models in order to understand how to create a general translation system, which is the main goal of this thesis. This general translation system should take logical formulae from the logic presented in Ghosh & Verbrugge (online first) and automatically generate PRIMs models from them.

4.1.2 Strategies represented in the logic

In Section 3.1 on page 37, we display a logical formula for each game, for each strategy. Consider the myopic strategy. The formulae required to represent the myopic strategy for Games 1 through 4, $1'$, and $3'$ from Section 3.1 on page 37 are displayed below, where the superscript refers to the game number:

$\mathcal{K}_P^{1'} : [(\delta_{P,1'} \wedge \gamma_{P,1'} \wedge (1 \leqslant 2) \wedge \mathbf{root}) \mapsto c]^P$

$\mathcal{K}_P^{3'} : [(\delta_{P,3'} \wedge \gamma_{P,3'} \wedge (1 \leqslant 2) \wedge \mathbf{root}) \mapsto c]^P$

$\mathcal{K}_P^{1} : [(\delta_{P,1} \wedge \gamma_{P,1} \wedge (1 \leqslant 2) \wedge \langle b^- \rangle \mathbf{root}) \mapsto c]^P$

$\mathcal{K}_P^{2} : [(\delta_{P,2} \wedge \gamma_{P,2} \wedge (1 \leqslant 2) \wedge \langle b^- \rangle \mathbf{root}) \mapsto c]^P$

$\mathcal{K}_P^{3} : [(\delta_{P,3} \wedge \gamma_{P,3} \wedge (1 \leqslant 2) \wedge \langle b^- \rangle \mathbf{root}) \mapsto c]^P$

$\mathcal{K}_P^{4} : [(\delta_{P,4} \wedge \gamma_{P,4} \wedge (1 \leqslant 2) \wedge \langle b^- \rangle \mathbf{root}) \mapsto c]^P$

Regardless of the game it applies to, the corresponding formula always needs to specify where two payoffs are, what they are, which two values should be compared, and where the root of the game tree is. This can differ between games. For example, for Games $1'$ and $3'$ the currently controlled node is the root node, whereas for Games 1 through 4, a $\langle b^- \rangle$ move is required to reach the root node from P's current decision point.

As shown in Section 2.2.1 on page 26 we can use the conjunction or disjunction of multiple strategy formulae. However, if we use, for example, the conjunction of $\mathcal{K}_P^{1'}$ and $\mathcal{K}_P^{3'}$, we would have to adhere to both $\mathcal{K}_P^{1'}$ and $\mathcal{K}_P^{3'}$, even though one of these may not apply to the current game. On the other hand, if we use the disjunction of two strategy formulae, we would be allowed to adhere to either of them, even though one of them may not apply to the current game. Because of this, we are required to use one formula for each game, for each strategy.

Our goal is to create a general translation system that translates strategies represented in logical formulae into PRIMs models. Because these formulae apply to a single game each, the PRIMs models they are translated into should also apply to a single game each. In Section 4.1.3 on page 53, we provide a thorough exploration of such differences between the logic presented in Ghosh & Verbrugge (online first) and the models we present in Section 3.3 on page 41.

4.1.3 Differences between the logic and the models

In the previous section, we have shown that our models can play all six games from (Ghosh et al., 2017), whereas a logical formula specifies a strategy for a single game. Furthermore, these logical formulae are very similar to a Horn clause (Horn (1951)). They consist of a conjunction of propositions (sometimes preceded by operators) such as $(u_C = p_C)$, $(p \leqslant q)$, or **root**. If all of the propositions in this conjunction are true, a specified action should be played. If at least one of them is not true, anything may be played. Our models do not play in this manner. They do not require that some node is the root of the tree and that some specific payoff has some specific value. The models treat these payoffs as variable values, which are stored in working memory and retrieved to perform a declarative memory retrieval. Furthermore, the models do not 'play anything' when some payoff is not smaller than or equal to another payoff. Instead, their actions are based on how some payoffs in the game differ from each other. Another difference is that our own-payoff models play right as soon as a future payoff is found that is larger than the current payoff. The formula inspects each payoff and each comparison before right can be played.

In general, the PRIMs models we created in Chapter 3 use the myopic and own-payoff strategies as if they are algorithms, where payoffs are variables and the outcome of this algorithm is determined by comparing these variables. The logical formulae, on the other hand, specify a list of conditions that should all hold if some action is to be played. If at least one of them does not hold, the formula allows for any action.

Because we wish to translate logical formulae into PRIMs models, we have to take these differences into account. The PRIMs models we create have the same properties as the logical formulae

and they are not PRIMs models generated from a strategy as represented intuitively, or as an algorithm. A general translation system does not have access to human intuition. It only has access to a logical formula. Consider the myopic strategy for Game $1'$ as presented above, which, without abbreviations, is as follows:

$$\mathcal{K}_P^{1'} : [(\langle c^+ \rangle (u_P = 2) \wedge \langle d^+ \rangle \langle e^+ \rangle (u_P = 1) \wedge (1 \leqslant 2) \wedge \mathbf{root}) \mapsto c]^P$$

This formula states that if all of the following hold

- After a c move from the current node, player P's payoff has the value 2;
- After a d move followed by an e move from the current node, player P's payoff has the value 2;
- The value 1 is smaller than or equal to the value 2;
- The current node is the root node;

then c should be played. If at least one of these statements does not hold, then anything may be played.

This formula is not the myopic strategy itself, but this formula specifies which propositions have to be tested in order to play the prescribed action. The models we create with our translation system should also test these propositions. Furthermore, the models should play the specified action when all of these propositions prove to be true, and should play any action when at least one of them is not true. In a conjunction of strategy formulae this does not apply. In this case, when at least one of the items in the conjunction is not true, the model should look for a strategy formulae that *is* applicable. However, in this section, we will assume a single strategy formula. We create a system that translates these formulae into PRIMs models, and we have to take into account the specifications given by the logical formulae. Because of this, the PRIMs models our translation system creates are quite different from the PRIMs models we have created in Section 3.3 on page 41. In the remainder of this chapter, we explain how we create such a translation system.

4.2 Representations

We create our system in Java 1.8. For an introduction on Java, see, for example, (Savitch, 2012). Class hierarchies in Java allow us to represent games and strategies more easily. Furthermore, Java's regular expressions and IO are very useful in creating PRIMs files.

4.2.1 Representing games

We use six Java classes to create game trees. They are the following:

- Game
- Treeobject
- Edge
- Node
- Nonleaf
- Leaf

Each of these classes has *name* as an instance variable, which is used to identify the object. Instance variables are properties of the objects created using this class. For example, the class Game is a class. Objects of this class could be be *game_1* and *game_2*. For *game_1*, the instance variable *name* has the value 'Game 1'. For *game_2*, the instance variable *name* has the value 'Game 2'. Instance variables refer to other objects in Java. In this case, the name 'Game 1' is a string object.

The Treeobject class is a superclass for all objects that constitute a tree. Its subclasses are Node and Edge. The Edge class has as instance variables the node it comes from, the node it goes to, and which name its corresponding direction has (such as down or right). The Node class is a superclass for leaf nodes and decision nodes. It has as instance variable whether the node is the root of the tree, as a boolean value. The Nonleaf class is for decision nodes. It has as instance variables the player who controls it (C or P) and whether it is the current node. The Leaf class is for leaf nodes. It has as instance variables what player C's payoff is at this leaf node, and what player P's payoff is at this leaf node.

A game has as instance variables a list of leaf nodes, a list of non-leaf nodes, and a list of edges. It also has a list of nodes, which is simply the list of leaf nodes and the list of non-leaf nodes combined. This list is useful for performing computations. Furthermore, within a game, it is specified within a boolean instance variable what the root node is, and what the current node is. Games do not have a subclass or a superclass.

A class diagram containing the classes required to create games can be found in Figure 4.1.

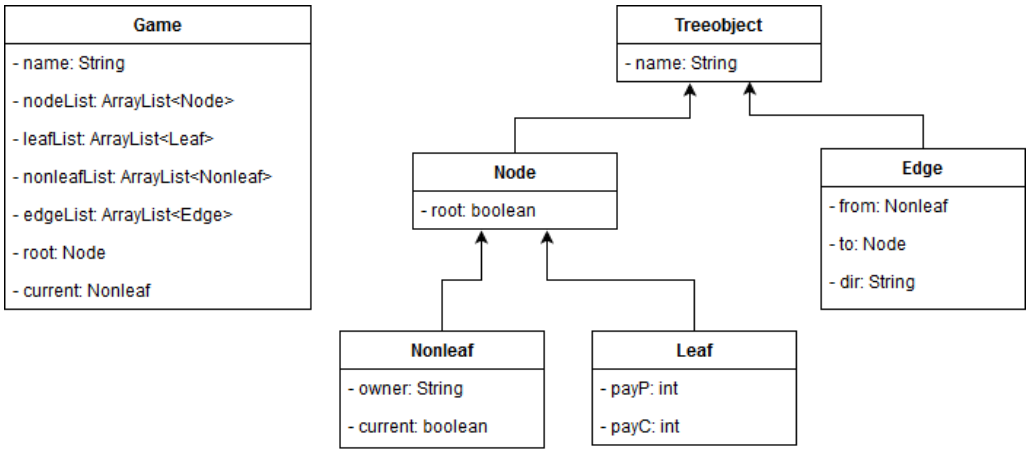

Figure 4.1: Class diagram for games and classes required to create games.

In the diagram in Figure 4.1, each rectangle is a class. The name of the class is in bold. It is followed by a list of instance variables of the class. These consist of the name of the instance variable, followed by a colon, followed by the type of the instance variable. The types String, boolean, and int (for integers) should be known to the reader. If not, it is recommended to read Chapter 1 of (Savitch, 2012), or another introduction to Java. All other types are explained in the present section. An arrow indicates from which class a class inherits. Classes inherit the instance variables from their superclasses. For example, the class Leaf inherits the instance variable *root* from its superclass Node, and the class Leaf inherits the instance variable *name* from its superclasses' superclass, Treeobject. The notation 'ArrayList<Node>' indicates that the instance variable is a list of nodes.

4.2.2 Representing strategies

In Section 2.2.1 on page 26, four of the five propositions used for strategies in Ghosh & Verbrugge (online first) are introduced. These are the following:

- **root**
- $\mathbf{turn}_i$
- $(u_i = q_i)$
- $(r \leqslant q)$

In Section 2.2.2 on page 29, it is stated that actions are also part of these propositions:

- $\Sigma \subseteq P$

We have a Java class for each of these propositions. Their class names are, respectively, Root, Turn, Utility, Comparison, and Action. All of these classes are subclasses of the class Logicobject. Each of them, except Root, has one or more instance variables. The Turn class has an instance variable denoting what the value of i is, that is, which player it is about. The Utility class has an instance variable denoting what the value of i is as well. Furthermore, the Utility class has an instance variable containing its numerical value, and an instance variable containing its name, which is required to distinguish equal payoffs at different leaf nodes.

The Comparison class has as instance variables the two utilities that should be compared. In this, we deviate from the logic. In the logic, r and q in a comparison $(r \leqslant q)$ are numerical values, not utilities of the type $(u_i = q_i)$. However, in a translated PRIMs model, a utility assignment such as $(u_i = q_i)$ would mean that a certain payoff at a certain leaf node is stored in working memory. A comparison such as $(r \leqslant q)$ means that two payoffs previously placed in working memory are used to start a declarative memory retrieval. If there are two utilities $(u_i = 1)$ and two comparisons, namely $(1 \leqslant 2)$ and $(2 \leqslant 1)$, which of these utilities refers to which comparison? It is only possible to know this if comparisons refer directly to utilities, which we do in our system.

The Action class represents the actions, which are also part of the propositions, $\Sigma \subseteq P$. Because the actions Σ are in bijection with the edges $\Rightarrow$ (for each action there is one edge and vice versa), the Action class has an instance variable denoting to which edge it belongs. When we create a game and its corresponding edges, we can use those edges to create the actions required for our strategy.

In Section 2.2.1 on page 26, three operators from Ghosh & Verbrugge (online first) are introduced: $\langle a^+ \rangle$, $\langle a^- \rangle$, and $\mathbb{B}_h^{(i,x)}$. For $\langle a^+ \rangle$ and $\langle a^- \rangle$, we use the class Step. It is called 'Step' because the operator denotes a step being taken in a game tree from one node to another. We use a single class for these two operators because their only difference is the direction of the step. Similar to the class Action, the class Step has an instance variable that denotes which edge it refers to. It also has a boolean instance variable denoting which of the two operators $\langle a^+ \rangle$ and $\langle a^- \rangle$ is used, or which direction the edge is traversed in. For $\mathbb{B}_h^{(i,x)}$, we use the class Belief. It has an instance variable that denotes to which player it refers, and an instance variable that denotes to which node it refers. In the logic, the belief operator also indicates to which game it applies, but we assume that it applies to the game of the strategy in which it resides.

The classes Step and Belief share the superclass Logicmodifier.

Using the classes we introduced, we can create more complex combinations of propositions and operators, such as $\langle d^+ \rangle \mathbb{B}^P f$ or $\langle b^- \rangle \langle a^+ \rangle (u_C = 2)$. For these logical propositions modified by operators such as $\langle a^+ \rangle$ and $\mathbb{B}_h^{(i,x)}$, we have the class Modlogic. The Modlogic class has as instance variables the proposition that is being modified, a list of steps that may precede it, and a list of beliefs that may precede it as well. We assume that a list of operators preceding a proposition is always a list of belief operators, followed by a list of steps. We do this to simplify the translation process and to disallow constructs such as $\langle a^+ \rangle \mathbb{B}_h^{(i,x)} \langle a^+ \rangle \mathbb{B}_h^{(i,x)}$. Furthermore, we assume that the belief operator is only used when the proposition is an action. This is simply because is intuitively not necessary to formulate beliefs about whose turn it is, where the root node is, what a specific payoff value is, or which payoff is smaller than which other payoff. Because the games we consider are games of complete information, the first three of these should be common knowledge. The last of these should be common knowledge because we assume that all players know which payoffs are smaller than or equal to other given payoffs. However, players may not know which strategy another player is using, or which action the other might play.

Using the Modlogic class, we create strategies. The class Strategy has four instance variables: its name, which is needed to generate PRIMs models, a list of Modlogic objects, which is the conjunction of propositions modified by operators in the strategy, the player the strategy belongs to, and the action the strategy prescribes if all of the items in the conjunction are true.

A class diagram containing the classes required to create strategies can be found in Figure 4.2.

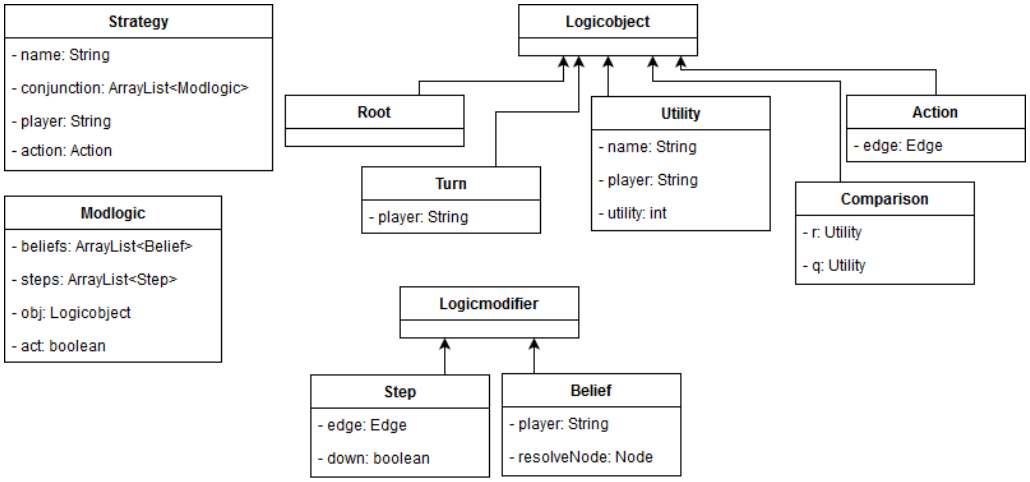

Figure 4.2: Class diagram for games and classes required to create strategies.

4.3 Translating logical formulae to PRIMs models

4.3.1 Task script

Our system needs to generate all code required to run a PRIMs model. This includes a task execution script such as the one seen in Section 3.2.3 on page 40, which includes a representation of the game tree using the function *screen*. Our system recursively generates this code using the game's representation as presented in Section 3.3.2 on page 42. It then generates the full task execution script which starts the trial, displays the game using *screen*, waits for the model to play an action, and ends the trial. Note that the task execution script in Section 3.3.2 updates the display as long as the game has not ended, so that the model could not only play its first move, but also any future moves. The task execution script generated by our system does not do this, because the strategies represented in the logic only specify one action at one node.

4.3.2 Declarative memory

Our system also needs to generate an initialization script which specifies which chunks are in declarative memory when the model starts performing a task (see Section 3.3.3 on page 44). These chunks consist of comparisons and beliefs.

Comparisons

Like the models in Ghosh & Verbrugge (online first) and our models in Section 3.3 on page 41, automatically generated models need to 'know' which numerical values are equal to or smaller than which other numerical values. We use the same kind of chunks as those in Ghosh & Verbrugge (online first) (shown in Section 3.2.2 on page 39) and the one we used in our myopic and own-payoff models (see Section 3.3.3 on page 44). These chunks have five slots, containing respectively the chunk's name, the first value, whether the first value is equal to or smaller than the second value, the second value, and the value 'comparison'. To create these chunks, our system traverses through the game tree, keeping track of all unique payoffs in it. It then creates a chunk for each combination of two of these payoffs, using 'bigger' in the third slot if the first value is bigger than the second value, and using 'not' in the third slot if the first value is not bigger than the second value.

Beliefs

In Ghosh & Verbrugge (online first), a formula is presented for a first-order theory of mind strategy in Game 1 for player P:

$$\tau_P^1 : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle f) \mapsto c]^P.$$

In this formula, $\mathbb{B}_{g1}^{(n2,P)} \langle d \rangle f$ indicates that player P should believe that after it plays d, player C will play f. However, the formula does not specify *how* this belief should be acquired.

In Stevens, Taatgen & Cnossen (2015), a bargaining game is played where an ACT-R model tests its beliefs by comparing its opponent's behaviour to chunks in its declarative memory. It could attribute an aggressive or a cooperative strategy to its opponent. Whenever its opponent played a move, it tried to retrieve a chunk from memory corresponding to this move. If the retrieved chunk corresponded to an aggressive strategy, it would attribute an aggressive strategy to its opponent, and vice versa for a cooperative strategy.

We use a similar approach. Our automatically generated models start with chunks corresponding to the own-payoff strategy, the EFR strategy, and the BI strategy in its declarative memory. We use the own-payoff strategy because its PRIMs models have a good fit in Ghosh & Verbrugge (online first). We use the EFR strategy because the actions it prescribes correspond somewhat to the human data of Ghosh et al. (2015). We use the BI strategy because it reaches a Nash equilibrium.

When generating a PRIMs model, our system calculates all EFR and BI strategies for the relevant game using the algorithms specified in Gradwohl & Heifetz (2011). It also calculates all own-payoff strategies for the relevant game. The algorithm to obtain all own-payoff strategies can be summarized as follows: 'If player i is at decision node s_x, and the highest payoff at any leaf node reachable from node s_x for player i is q, then an action a_x belongs to an own-payoff strategy of player i if there is a play $\ldots s_x a_x \ldots s_n$ where s_n is a leaf node where player i's payoff is q'.

Our system creates chunks corresponding to each strategy it has calculated. These chunks have five or more slots. The first slot contains its name, which is 'strat-fact-' followed by a number. The second is 'strat' to indicate that it is a strategy and to prevent erroneous retrievals. The third slot specifies which strategy it is, 'efr' for EFR, 'bi' for BI, and 'op' for own-payoff. The fourth slot specifies which player the strategy belongs to, player C or player P. The fifth slot, as well as any further slots, each contain a move, and specify a sequence of moves corresponding to the strategy. The fifth slot contains the first move in this sequence, the sixth slot contains the second move in this sequence, the seventh slot contains the third move in this sequence, et cetera.

4.3.3 Model initialization

Automatically generating a model initialization script from a strategy is fairly straightforward. The model initialization script generated for the myopic strategy for Game 1′ is as follows:

```
define task Myopic_11 {
        initial-goals: (root-one)
        goals: (playdir playother utility-one utility-two comparison-
            ↪ one)
        task-constants: (bigger not c p current notcurrent decision-
            ↪ node leaf root end comparison right down)
        imaginal-autoclear: nil
        default-activation: 1.0
        rt: -2.0
        ol: t
        batch-trace: t
        default-operator-assoc: 8.0
        default-operator-self-assoc: -2.0
}
```

For the name of the task, the name specified in the strategy in Java is used. The task's goals consist of *playdir*, *playother*, and one goal for each proposition in the strategy. There is one initial goal, which is the first proposition that will be tested. Every other goal is a non-initial goal. How the first goal is selected and how the goals are ordered can be found in the next subsection.

The task constants are the same as those in Section 3.3.3 on page 44. However, *end* is not included in the tasks constants, because our generated models do not require it. Furthermore, instead of adding *right* and *down* to each model definition, our model adds the names of the directions found in the corresponding game, which is an instance variable for the Edge class.

The PRIMs parameters of our automatically generated models are almost the same as those in Section 3.3.3 on page 44. There are two differences. First of all, *default-operator-self-assoc* is set to -2.0 instead of 0 or -1.0. This is because no operators in our automatically generated models should fire more than once. Furthermore, *default-operator-assoc* is set to 8.0 instead of its default value of 4.0. This is the association between an operator and the goal it is defined in. We increased it because each goal performs a small task, such as looking at the next leaf node or inspecting a payoff value, that should not be interrupted.

4.3.4 Goals and operators

Consider the logical formula for the myopic strategy for Game $1'$:

$$\mathcal{K}_P^{1'} : [(\langle c^+ \rangle (u_P = 2) \wedge \langle d^+ \rangle \langle e^+ \rangle (u_P = 1) \wedge (1 \leqslant 2) \wedge \mathbf{root}) \mapsto c]^P$$

A model using this formula should inspect two payoffs, make a comparison, and verify where the root of the tree is. If all of these things correspond to what the formula prescribes, the model should play c. Otherwise, it may play anything, because we do not yet use a conjunction of strategy formulae that exhausts all possibilities.

The translation system creates a goal for each proposition in the conjunction of the formula. In this example, these propositions are $(u_P = 2)$, $(u_P = 1)$, $(1 \leqslant 2)$, and **root**. It also creates two more goals, one for when the model plays the prescribed action, *playdir*, and one for when it may play anything, *playother*. Goals corresponding to propositions are named 'root-x', 'turn-x', 'utility-x', 'comparison-x', and 'action-x', where x is a number string such as 'one'. Operator names start with the name of the goal it is defined in, with the exception of operators within the goals *playdir* and *playother*.

The goal *playdir* has only one operator, *play*. The operator *play* plays the action prescribed by the strategy the model was generated with.

The goal *playother* has one operator for each outgoing edge from the currently active node. Each of these operators plays one of the actions possible at the currently active decision node, but only one can fire when the goal *playother* is in the goal buffer, because the task ends as soon as the model performs an action. Which of the operators fires depends on their activation noise, because they all have the same conditions.

The operators within the goals that correspond to propositions test whether that proposition is true. If it is true, the next proposition is tested, because the conjunction is only true if all of the propositions within it are true. This is done by placing a goal that corresponds to a proposition in the first slot of the goal buffer. When the operators within this goal test the corresponding proposition and the proposition turns out to be true, the first slot of the goal buffer is overwritten by the goal that corresponds to the next proposition that should be tested. The last goal corresponding to a proposition places *playdir* in the first slot of the goal buffer if the corresponding proposition is true. Therefore, if all propositions are true, the prescribed action is played.

Within a goal that corresponds to a proposition, if the proposition is false, then the conjunction within the corresponding strategy is false, and if the conjunction is false, the strategy prescribes that anything may be played. Therefore, if the operators within a goal that corresponds to a proposition find out that the proposition is false, the goal *playother* is placed in the first slot of the goal buffer, overwriting the current goal. Therefore, if one proposition is false, any action is played.

For goals corresponding to the proposition **root**, the proposition is true if 'root' is in *V4*, and false if it is not. For goals corresponding to a proposition $\mathbf{turn}_i$, the proposition is true if i ('c' or ''p') is in *V2*, and false if it is not. For goals corresponding to a proposition $(u_i = q_i)$, the proposition is true if q_i (for example, 'one', 'two', or 'three'), is in *V2*. Furthermore, if the proposition is true, the value in *V2* is also placed in a working memory slot for use with comparisons later. If q_i is not in *V2*, the proposition is false.

Each comparison $(r \leqslant q)$ refers to two utilities. These utilities have to be present in buffer slots accessible by the model at the same time so they can be copied to the retrieval buffer to make a retrieval request. Because they are usually in different locations in the visual display, they cannot both be in the visual buffer slots at the same time. Because of this, they have to be copied to working memory, so they can be copied to the retrieval buffer later. To ensure that utilities do not overwrite each other in working memory slots and to ensure that comparisons can be executed successfully, the translation system maps each utility to a working memory slot. When the system generates the code for a comparison, it knows which working memory slots the comparison's utilities are mapped to, so it knows which working memory slots have to be copied to the retrieval buffer to make a retrieval request. A goal corresponding to a comparison has three operators: '-start-retrieval', '-not-bigger', and '-bigger'. Note that these operator names are preceded by the name of their goal. The operator '-start-retrieval' places the values r and q from working memory into slots of the retrieval buffer. The operator '-not-bigger' fires when the chunk that has been retrieved states that r is not bigger than q. In this case, the proposition $(r \leqslant q)$ is true. The operator '-bigger' fires when the chunk that has been retrieved states that r is bigger than q. In this case, the proposition $(r \leqslant q)$ is false.

Some of the items in a conjunction of a strategy formula are an action, preceded by several belief operators, such as $\mathbb{B}_{g1}^{(n2,P)}\langle d\rangle f$. For each of these beliefs, the model has to test whether the belief is true. In Section 4.3.2 on page 58, we explain how automatically generated models have chunks corresponding to beliefs in declarative memory. To test whether a belief is true, a model simply has to retrieve such a chunk, corresponding to the situation described in the belief, from declarative memory, and verify whether the future situation the retrieved belief describes corresponds to the belief as prescribed by the strategy formula. For example, the proposition $\mathbb{B}_{g1}^{(n2,P)}\langle d\rangle f$ states that in Game 1 (see Figure 2.1 on page 18), at node *n2*, which is player P's first decision node, after a d move, player C will play *f*. Because this belief involves a player C move, the model retrieves a strategy chunk for player C's strategies. Because the belief describes a situation where player C has already played b, which is right, the model retrieves a strategy chunk where the first move of player C is right. It can then compare player C's second move in the chunk it retrieves to the move prescribed by the strategy, which is *f*, which is also right. If these are the same, the proposition is true. If it is not, the proposition is not true.

4.3.5 Sorting the propositions

Within logical formulae such as $a \wedge b \wedge c$ and $c \wedge a \wedge b$, the ordering of conjuncts does not change the meaning of the formula. The same holds for strategies such as $\mathcal{K}_P^{1'} : [(\langle c^+\rangle(u_P = 2) \wedge \langle d^+\rangle\langle e^+\rangle(u_P = 1) \wedge (1 \leqslant 2) \wedge \mathbf{root}) \mapsto c]^P$. However, PRIMs models can only fire one operator at a time, and since multiple operators need to fire to complete a goal that tests a proposition, it can only test these propositions one at a time.

Luckily, there is already some ordering in the propositions. A PRIMs model can only test whether **root**, $\mathbf{turn}_i$, and $(u_i = q_i)$ are true when it is focusing at the node to which these propositions apply. These nodes can be found by looking at the $\langle a^+\rangle$ and $\langle a^-\rangle$ operators that precede the corresponding proposition. For example, $\langle d^+\rangle\langle e^+\rangle(u_P = 1)$ has to be verified at the leaf node reached by performing a d and an e action at the currently active node. Furthermore, a comparison such as $(1 \leqslant 2)$ can only be verified once its corresponding utilities have been placed in working memory.

We believe that human participants start playing games of marble drop by looking at the root of a tree. An eye-tracking experiment (Figure 3.2 on page 46) confirms this. In this figure the participant's first fifteen fixations are depicted in black. Our PRIMs models also start with their

focus at the root of the tree, because a PRIMs model starts with its focus at the first item on the top-most hierarchical level of the visual display, which is the root of the tree in our models. When our PRIMs models start, they may test whichever propositions can be tested at the root of the tree.

We base our order of goals on this notion. For each proposition, our translation system computes where in the game tree this proposition can be verified. Comparisons can be verified as soon as the second utility has been inspected. Beliefs do not require the model to compare values in the visual buffer to anything, so they do not have a location in the game tree where they can be verified. Instead, beliefs are tested after all other propositions within our automatically generated models, because we assume that knowing the game's payoffs and their relations are required to do so.

The translation system then calculates the shortest path through each of the locations required to test all of the strategy's propositions. The goals corresponding to these propositions are sorted based on when they can be verified on this path.

For example, consider the formula $\mathcal{K}_P^{1'} : [(\langle c^+\rangle(u_P = 2) \wedge \langle d^+\rangle\langle e^+\rangle(u_P = 1) \wedge (1 \leqslant 2) \wedge \textbf{root}) \mapsto c]^P$. It has four propositions. In Figure 4.3, it can be seen where these propositions must be verified. In Table 4.1, it can be seen which propositions have to be verified at each location.

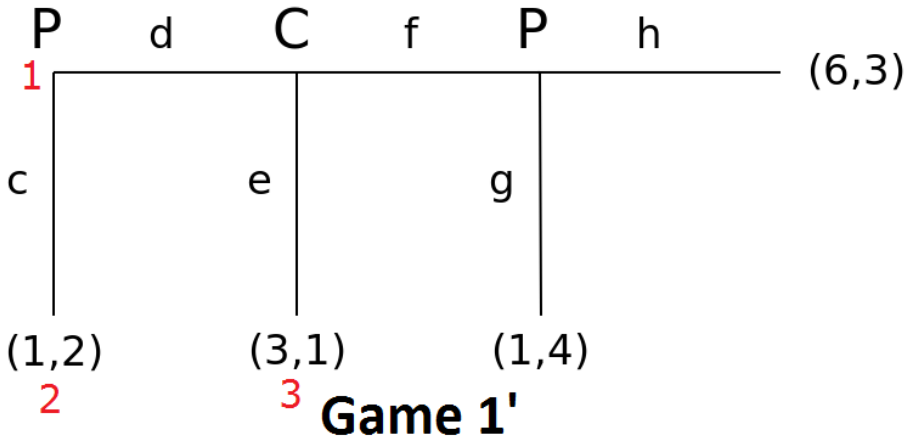

Figure 4.3: Game $1'$ of Ghosh et al. (2017), with locations where propositions must be verified in red. The corresponding propositions can be found in Table 4.1

Location	Proposition
1	**root**
2	$(\langle c^+\rangle(u_P = 2)$
3	$\langle d^+\rangle\langle e^+\rangle(u_P = 1)$ and $(1 \leqslant 2)$

Table 4.1: An overview of which propositions have to be verified at which location in Figure 4.3

The proposition **root** must be verified at location **1**. The proposition $(\langle c^+\rangle(u_P = 2)$ must be verified at location **2**. The propositions $\langle d^+\rangle\langle e^+\rangle(u_P = 1)$ and $(1 \leqslant 2)$ must be verified at location **3**, but the comparison can only be verified once both payoff values have been placed in working memory, so it has to be verified after $\langle d^+\rangle\langle e^+\rangle(u_P = 1)$. The shortest route through locations **1**, **2**, and **3**, starting at the root node, is $\langle c^+\rangle\langle c^-\rangle\langle d^+\rangle\langle e^+\rangle$. If we take this route, the propositions are verified in the following order: **root**, $\langle c^+\rangle(u_P = 2)$, $\langle d^+\rangle\langle e^+\rangle(u_P = 1)$, $(1 \leqslant 2)$.

This is how the goals are ordered. The first goal is 'root-one', which is placed in the list of initial goals. The next goals are 'utility-one', 'utility-two', and 'comparison-one', in that order, which are placed in the list of non-initial goals. When any of the propositions corresponding to

these goals are true, the next goal replaces this goal in the goal buffer. For 'comparison-one', the next goal is *playdir*.

Before any proposition corresponding to a goal can be verified, the focus of the PRIMs model has to be moved to the node where that proposition can be verified. To do this, each goal starts with a set of operators that move the model's focus to the node relevant for this goal. In the previous example, the goal 'utility-one' would start with an operator moving the focus along edge c.

4.4 Results

4.4.1 Example model

As an example of what kind of models the translation system creates, an automatically generated myopic model for Game $1'$ of Ghosh et al. (2017) can be found in Appendix D on page 100. The logical formula this model was generated from is

$$\mathcal{K}_P^{1'} : [(\langle c^+ \rangle (u_P = 2) \wedge \langle d^+ \rangle \langle e^+ \rangle (u_P = 1) \wedge (1 \leqslant 2) \wedge \mathbf{root}) \mapsto c]^P.$$

Table 4.2 lists, for each part of this formula, what the corresponding goal is and which focus actions are performed in this goal. In this table, the goals are sorted in the order that is explained in Section 4.3.5 on page 60.

Proposition	Goal	Focus actions
root	root-one	-
$\langle c^+ \rangle (u_P = 2)$	utility-one	$\langle c^+ \rangle$
$\langle d^+ \rangle \langle e^+ \rangle (u_P = 1)$	utility-two	$\langle c^- \rangle \langle d^+ \rangle \langle e^+ \rangle$
$(1 \leqslant 2)$	comparison-one	-
$\mapsto c$	playdir	-

Table 4.2: The goals of the automatically generated myopic model for Game $1'$ of Ghosh et al. (2017)

4.4.2 Exploratory statistics

To test our translation system, we recreate the models we described in Chapter 3 on page 37 using our translation system. These are the myopic and own-payoff models for Games $1'$ and $3'$ of Ghosh et al. (2017). The logical formulae for these models are as follows:

$\mathcal{K}_P^{1'} : [(\delta_{P,1'} \wedge \gamma_{P,1'} \wedge (1 \leqslant 2) \wedge \mathbf{root}) \mapsto c]^P$

$\mathcal{K}_P^{3'} : [(\delta_{P,3'} \wedge \gamma_{P,3'} \wedge (1 \leqslant 2) \wedge \mathbf{root}) \mapsto c]^P$

$\mathcal{X}_P^{1'} : [(\alpha_{P,1'} \wedge \beta_{P,1'} \wedge \gamma_{P,1'} \wedge \delta_{P,1'} \wedge (1 \leqslant 2) \wedge (2 \leqslant 4) \wedge (2 \leqslant 3) \wedge \mathbf{root}) \mapsto d]^P$

$\mathcal{X}_P^{3'} : [(\alpha_{P,3'} \wedge \beta_{P,3'} \wedge \gamma_{P,3'} \wedge \delta_{P,3'} \wedge (1 \leqslant 2) \wedge (2 \leqslant 4) \wedge \mathbf{root}) \mapsto d]^P$

We use these four formulae to automatically generate four models. We test these models using the same method as used in Ghosh et al. (2015) (as found in Section 3.4 on page 48): for each of these models, we let fifty virtual participants play fifty times each.

After running these models, we find that the proportions of down played for these models is the same as those found in Table 3.2 on page 48. The automatically generated myopic models always play down, and the automatically generated own-payoff models always play right. Reaction times for our automatically generated myopic and own-payoff models can be found in Figure 4.4.

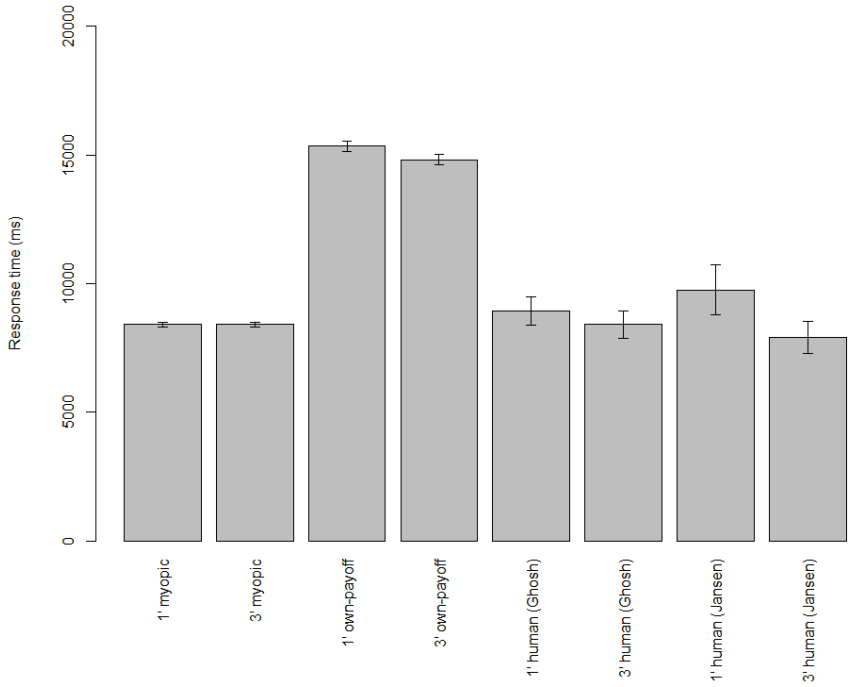

Figure 4.4: Reaction times for our automatically generated myopic and own-payoff models.

In Figure 4.4, it can be seen that, just like the own-payoff models we created in Chapter 3 on page 37, the automatically generated own-payoff models are too slow to model human reaction times. However, we can train the models on focus actions to speed them up. To do so, we let each virtual participant perform the training task several times, and then let them play the game a single time to ensure that any speed-up is caused by playing the training tasks and not by the game itself. The resulting reaction times can be found in Figure 4.5.

In Figure 4.5, it can be seen that our trained models are not much faster than the untrained models (see Figure 4.4). In fact, we selected 200 gaze training trials because this seemed to be the number where there was no speed-up from further training. Note that the virtual participants in the 'untrained' models actually played 50 trials, so they were trained with 25 trials on average. It appears that 25 trials of a game cause models to achieve reaction times of approximately 15 seconds, and performing 200 gaze training tasks also cause models to achieve reaction times of approximately 15 seconds. It appears that training only on focus actions cannot bring the automatically generated own-payoff model's reaction times down to human reaction times, but perhaps training on the task itself can. In Figure 4.6, reaction times for our automatically generated myopic and own-payoff models can be found after 55 model runs.

In Figure 4.6, it can be seen that the own-payoff model fits the data from Ghosh et al. (2015) and Ghosh et al. (2017) quite well after 55 model runs. However, it is still an issue why we should pick this number, as explained in Section 3.5 on page 50.

We have seen that the automatically generated models cannot fit the human data when only training on focus actions, but they fit the human data when training on playing full centipede games. We have also seen that the models we create in Chapter 3 on page 37 can fit the human data when only training on focus actions. This tells us that some new addition introduced in the automatically generated models is the cause of this slow-down in reaction times. There are, in fact, several new additions, as explained in Section 4.3.4 on page 59, which may all contribute to this slow-down.

To prevent overfitting, we may not be able to compare our automatically generated models to

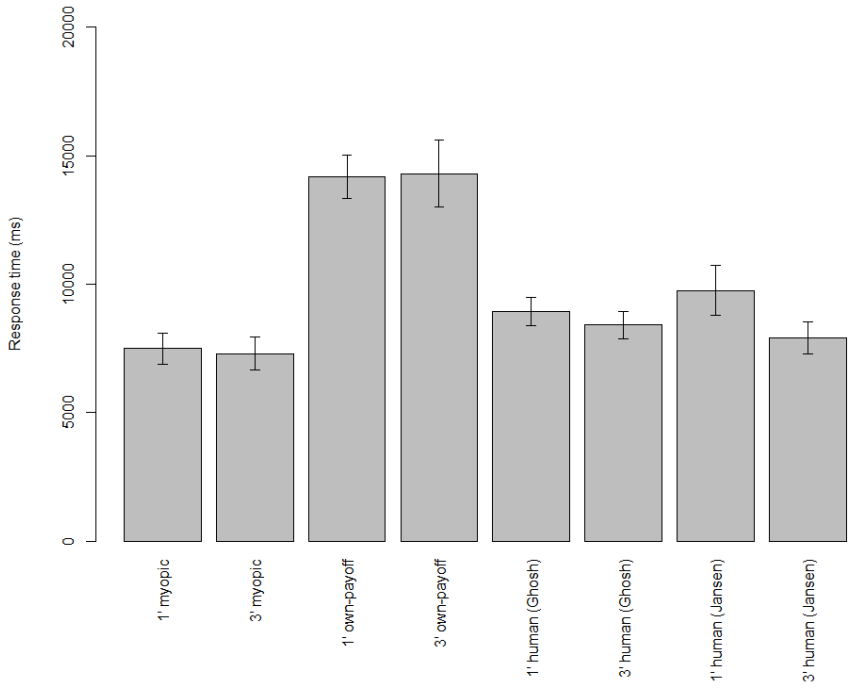

Figure 4.5: Reaction times for our automatically generated myopic and own-payoff models after 200 training trials.

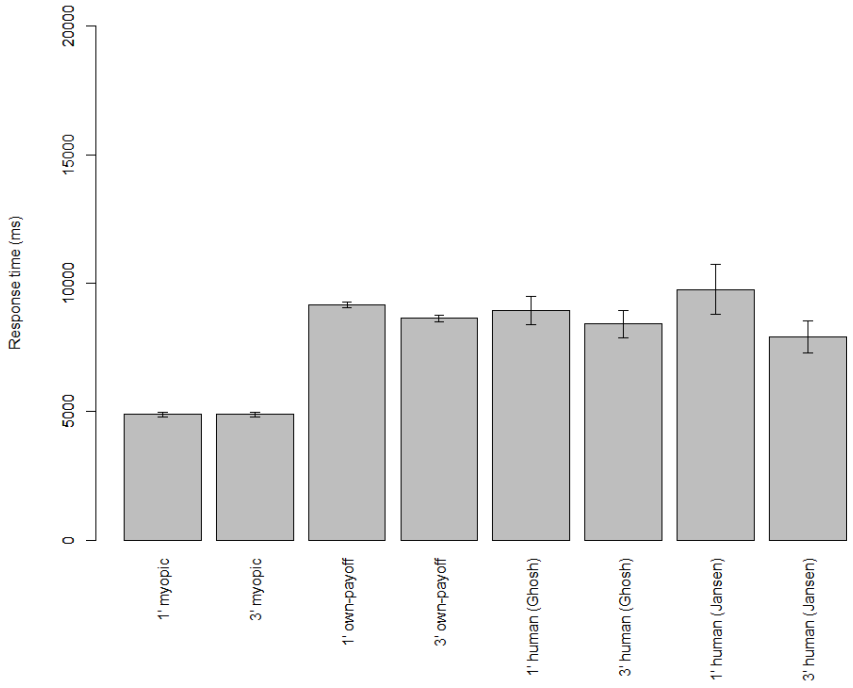

Figure 4.6: Reaction times for our automatically generated myopic and own-payoff models after 55 model runs.

human data. However, we can compare different strategies with each other. For example, in Figure 4.6, it can be seen that the own-payoff model is faster in Game $3'$ than in Game $1'$. Furthermore, the own-payoff models are about twice as slow as the myopic models, just as we've seen with our models in Chapter 3 on page 37.

We also automatically create PRIMs models using the formulae corresponding to Theory of Mind strategies as presented in Ghosh et al. (2015). These formulae describe strategies involving zero-order, first-order, and second-order Theory of Mind. We only use Games $1'$ and $3'$ for these models. Mean reaction times for these models can be found in Table 4.3. These reaction times are

	Game $1'$	Game $3'$
ToM 0	34799.41	35220.34
ToM 1	34717.34	35341.00
ToM 2	34723.19	35337.90

Table 4.3: Mean reaction times in milliseconds for automatically generated Theory of Mind models playing Games $1'$ and $3'$

very similar. This may be caused by each of these models' formulae sharing many propositions, such as the ones summarized with φ and ψ_P. Their main differences lie in the zero, one or two memory retrievals they have to perform to verify their beliefs.

Table 4.4 contains proportions of down played by the Theory of Mind models. In Table 4.4,

	Game $1'$	Game $3'$
ToM 0	0.4908	0.654
ToM 1	0.516	0.6492
ToM 2	0.5196	0.6472

Table 4.4: Proportions down played by automatically generated Theory of Mind models in Games $1'$ and $3'$

it can be seen that our automatically generated Theory of Mind models seem to play down more frequently in Game $3'$ than in Game $1'$. None of these proportions are 0 or 1, as is the case with our myopic and own-payoff models.

4.5 Exhaustive strategy formulae

In Section 4.3 on page 57, we describe our translation system which translates logical formulae into PRIMs models. This translation system translates one formula at a time. However, in Ghosh et al. (2014), it is demonstrated that the logic should be used to specify every possibility a strategy can encounter in a game, using a disjunction of multiple formulae. For example, consider the myopic strategy in Game $1'$ of Ghosh et al. (2017) (which can be found in Figure 2.2 on page 19). A player using the myopic strategy looks at his current and next payoff, plays down if his current payoff is larger, and plays right if his next payoff is larger. These two possibilities can be represented as follows:

$$\mathcal{K}_P^{1'} : [(\langle c^+ \rangle (u_P = 2) \wedge \langle d^+ \rangle \langle e^+ \rangle (u_P = 1) \wedge (1 \leqslant 2) \wedge \mathbf{root}) \mapsto c]^P$$

$$\mathcal{K}_P^{1'} : [(\langle c^+ \rangle (u_P = 2) \wedge \langle d^+ \rangle \langle e^+ \rangle (u_P = 1) \wedge (2 \leqslant 1) \wedge \mathbf{root}) \mapsto d]^P$$

A model can then try both of the strategies represented in the above strategy formulae. If the conjunction of one of these formulae is true, it should play the action prescribed by the formula. If the conjunction of one formula is not true, it should try the other formula. As long as you exhaust all possibilities a strategy can encounter, at least one formula must be true, so the model will never have to randomly select a strategy.

We expand our system to allow for constructs such as these. To represent them, we use a list of strategy objects in Java. Such a list of strategy objects can then be used to automatically generate

a PRIMs model. The model will try a strategy and, if it does not apply, move on to the next strategy. To do so, instead of placing the goal 'playother' into the goal buffer, it places the first goal of the next strategy into the goal buffer as soon as one of the conjuncts of the current strategy turns out to be false. Because checking the conjuncts of the next strategy starts at the root node of the tree, the model will first move its focus back to the root of the tree, before moving on to the next strategy. If the last strategy also fails, the model will play any enabled move. However, this should not happen if the strategy formulae exhaust all possibilities.

Like the conjunction of propositions within a strategy formula, a disjunction of strategy formulae themselves is also unordered. In this case, we simply create all permutations and run each of them an equal number of times when generating data, removing any order effects.

4.5.1 Testing BI and EFR

To test our exhaustive models, we will create models using the BI and EFR strategies, which are explained in Section 2.1 on page 17. We will use the strategies with Games 1 and 4 of Ghosh et al. (2017), which can be found in Figure 2.1 on page 18. We use these games because player C's payoff at the first leaf node differs between both games, which leads to different EFR strategies. Furthermore, player P's payoff at the rightmost leaf node also differs between these games, leading to different BI strategies in both games.

The formulae for BI in Game 1 are as follows:

$$-\eta_P^1 : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle e \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle \langle f \rangle g) \mapsto c]^P$$

The formulae for BI in Game 4 are as follows:

$$-\eta_P^1 : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle e \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle \langle f \rangle g) \mapsto c]^P$$

$$-\eta_P^1 : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle f \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle \langle f \rangle h) \mapsto d]^P$$

The formulae for EFR in Game 1 are as follows:

$$-\eta_P^1 : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle f \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle \langle f \rangle g) \mapsto d]^P$$

The formulae for EFR in Game 4 are as follows:

$$-\eta_P^1 : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle e \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle \langle f \rangle g) \mapsto c]^P$$

$$-\eta_P^1 : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle e \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle \langle f \rangle h) \mapsto c]^P$$

$$-\eta_P^1 : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle f \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle \langle f \rangle g) \mapsto d]^P$$

$$-\eta_P^1 : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle f \wedge \mathbb{B}_{g1}^{(n2,P)} \langle d \rangle \langle f \rangle h) \mapsto d]^P$$

These formulae have been created by Sujata Ghosh (personal communication), author of some of our references (Ghosh et al. (2014), Ghosh et al. (2015), Ghosh & Verbrugge (online first)). The corresponding document can be found in Appendix E on page 104.

We use these formulae to have our translation system automatically generate a set of exhaustive PRIMs models. We then use these models to obtain reaction times and decisions. The number of possible orders of formulae is the factorial of the number of formulae. Therefore there will be one possible order for the formulae for BI in Game 1, two possible orders for the formulae for BI in Game 4, one possible order for the formulae for EFR in Game 1, and 24 possible orders for the formulae for EFR in Game 1. We create a model for each of these orders. We run a model 48 times for each combination of a strategy and a game, where we run each possible order 48, divided by the total number of possible orders, times. For example, there are 24 possible orders for the formulae for EFR in Game 4, so we run each model corresponding to such an order two times.

The BI models only have BI beliefs in their declarative memory, and the EFR models only have EFR beliefs in their declarative memory.

	Game 1	Game 4
BI	0.438	0.458
EFR	0.646	0.532
Human (Ghosh et al. (2017))	0.397	0.273

Table 4.5: The proportion of human and virtual participants who played down in Game 1 and Game 4

The proportions of down played for the exhaustive BI and EFR models, as well as the human players in the experiment of Ghosh et al. (2017), can be found in Table 4.5. In Game 4, all moves are both BI and EFR moves, according to Table 2.3 on page 22 and Table 2.4 on page 22. Therefore, all beliefs in the BI and EFR formulae for Game 4 can be retrieved from declarative memory. Furthermore, there is an equal number of formulae prescribing c and d. Therefore it is not surprising that the proportions of BI and EFR models playing down in Game 4 are close to 0.5.

In Game 1, according to Table 2.3 on page 22 and Table 2.4 on page 22, C's only BI and EFR move at its first decision node is to play a, or down. However, since the models start at the second decision node, C has already played b. Therefore the model, when verifying a belief about C's actions, tries to retrieve a strategy that starts with C playing b. However, because the only BI and EFR moves for C are playing a at his first node, no such strategy is in the model's declarative memory. Therefore the model is not able to verify its beliefs about player C's actions, and it does not always play the action prescribed by a formula.

More humans seem to play down in Game 1 than in Game 4. This corresponds to the proportions of down played by the EFR models, suggesting that humans may use something similar to EFR more than something similar to BI. However, one must remain cautious with such conclusions, as there could be an alternative explanation for these proportions.

The reaction times for our automatically generated exhaustive BI and EFR models can be found in Figure 4.7.

These reaction times somewhat correspond to the number of formulae required to describe all possibilities for a strategy in the relevant game. The list of formulae for BI and EFR in Game 1 consists of one formula. Two formulae are required to describe the possibilities for BI in Game 4. Four formulae are required to describe the possibilities for EFR in Game 4. The reaction times do not seem to be a multiple of the number of formulae required to describe a strategy in a game, but they do appear to be a multiple of the number of formulae required plus an intercept.

To test this, we perform a simple linear regression using number of formulae to predict reaction times. A significant regression equation was found ($F(1, 189) = 432.6, p < 2.2 * 10^{-16}$), with an R^2 of 0.696. Predicted reaction time in milliseconds is equal to $10401 + 50453 \cdot$number of formulae.

A prediction of reaction times based on this model can be found in Figure 4.8.

It can be seen that the plots in Figure 4.8 and Figure 4.7 are very similar.

The fit of our linear regression model can be explained by the models created from our formulae. Each model tries formulae until it finds one that fits the current situation, which could explain the slope. The intercept could be explained by the fact that each model has to look at all payoffs and compare all of these payoffs, regardless of the number of formulae.

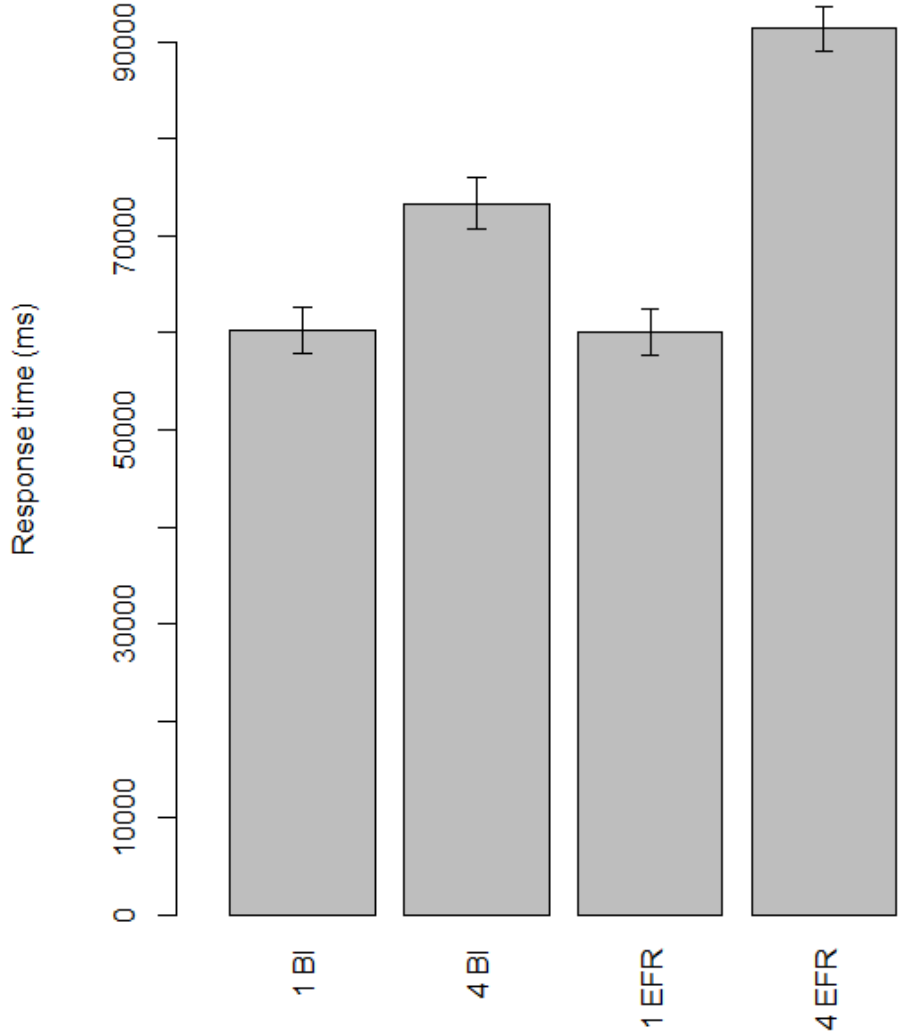

Figure 4.7: Reaction times for our automatically generated exhaustive BI and EFR models.

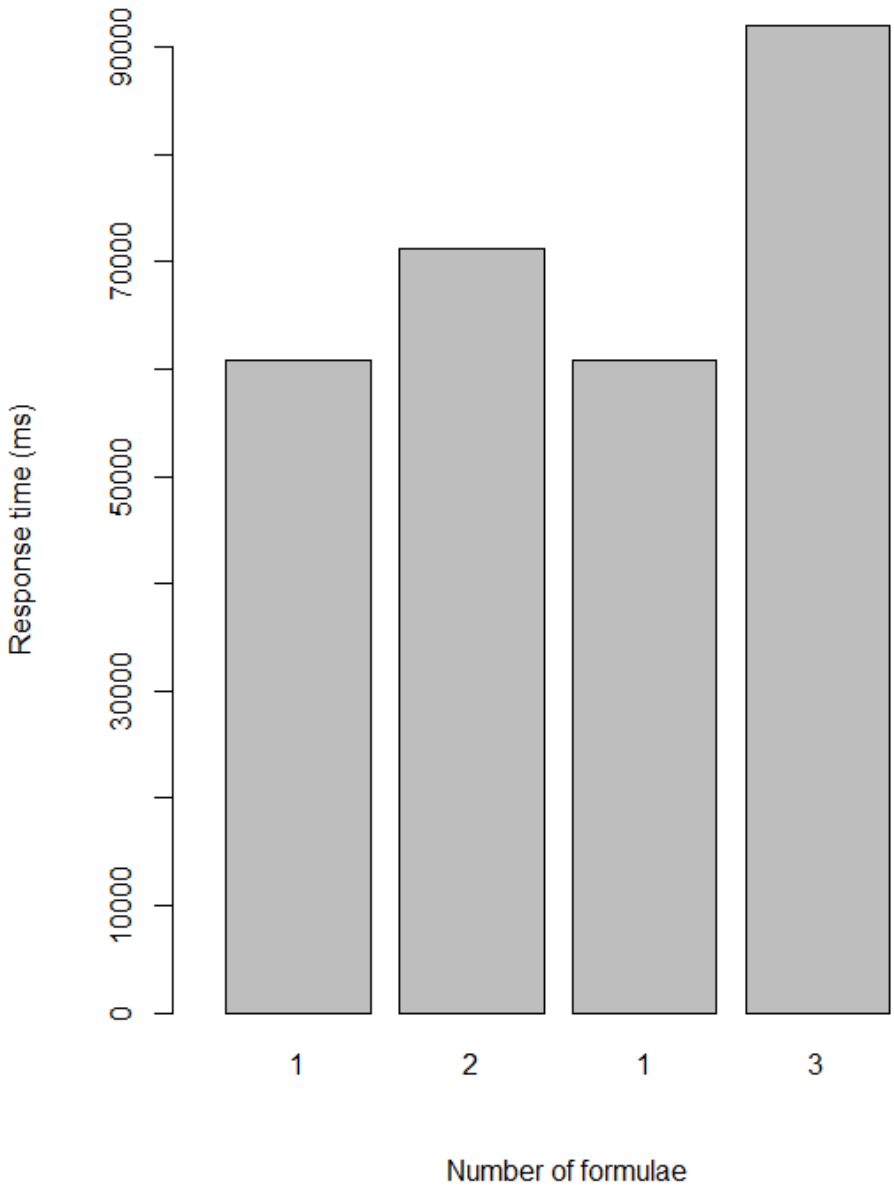

Figure 4.8: Reaction times for our automatically generated exhaustive BI and EFR models, as predicted by our linear regression model. 'BI 1' are the response times for our exhaustive BI model in Game 1, and 'EFR 4' are the response times for our exhaustive EFR model in Game 4. The remaining labels use the same notation.

Chapter 5

Discussion & Conclusion

In this thesis, we try to bring two fields closer together. The field of formal logic, and the field of cognitive modelling. We use a subset of dynamic perfect-information games, called *centipede games*, in our investigations. Centipede games are explained in Section 1.1 on page 11. Within the field of formal logic, there is a logic that can be used to express strategies in such centipede games, which is explained in Ghosh & Verbrugge (online first). Within the field of cognitive modelling, there is a relatively recent cognitive architecture called PRIMs (Taatgen, 2013b), which can be used to model the human mind as it performs experimental tasks. Our goal in this thesis is to create a general translation system which, when supplied with a strategy formula in the logic of Ghosh & Verbrugge (online first), creates a cognitive model in the PRIMs cognitive architecture. This PRIMs model should then be able to generate response times and decisions.

5.1 Chapter retrospect

5.1.1 Our myopic and own-payoff models

In Chapter 3 on page 37, we create PRIMs models that play the myopic and own-payoff strategies as presented in Ghosh & Verbrugge (online first). We find that the actions performed by these models do not resemble those of human participants, suggesting that human participants mostly use different strategies. Furthermore, we find that the reaction times of our models are too slow to resemble human reaction times. Because of Figure 3.2 on page 46, we believe that human participants do not use strategies that are even less complex than the myopic strategy, so we think that these slow reaction times are not caused by our models being too complex. Because training the models on focus actions can be used to fit the human reaction times, we think that untrained focus actions in PRIMs are too slow to model human gazes. An untrained PRIMs model is like a small child, in that it does not have the experience with looking around visual displays that adults have.

We have not only created the myopic and own-payoff models to verify the findings in Ghosh & Verbrugge (online first) and expand on their myopic and own-payoff models, we have also created them as a starting point for our general translation system, and to investigate focus actions in PRIMs. Furthermore, these models help us understand what the myopic and own-payoff strategies are, and how the logical formulae that represent them relate to them.

5.1.2 The general translation system

In Chapter 4 on page 52, we compare the formal logic presented in Ghosh & Verbrugge (online first) to our moypic and own-payoff models created in Chapter 3 on page 37. Taking into account the differences between them, we devise a way to create a general translation system. We describe how we represent games and logical formulae in Java, and describe how we automatically generate a PRIMs model using these Java representations. One of the main problems we encounter is that propositions in a conjunction, in formal logic, are unsorted while our PRIMs models require them

to be ordered. We solve this problem by ordering such propositions using the shortest path through the game tree, ordering the propositions using the order they appear in in this path.

We then automatically generate the myopic and own-payoff models, and find that our automatically generate model results are similar to those of our hand-made models.

Finally, we expand our system by including exhaustive strategy formulae, where multiple formulae are used to describe every possibility a strategy may encounter. We automatically generate PRIMs models that use the backward induction (BI) and extensive-form rationalizable (EFR) strategies from the strategies' respective logical formulae, and find that the response times of these models are a function of the number of formulae required to describe the corresponding strategy.

5.2 Findings

5.2.1 Our myopic and own-payoff models

In Section 3.3 on page 41, we discuss our creation of myopic and own-payoff models in PRIMs, capable of playing any centipede game. We present a method of representing centipede games hierarchically in PRIMs, which can be used in future work. We also add several focus actions to the PRIMs cognitive architecture, adding more possibilities to how our models look through the visual display.

In Section 3.4 on page 48, we present the results of our hand-made myopic and own-payoff models. We find that the proportions of *down* played are the same as those in Ghosh & Verbrugge (online first), and that the response times for the own-payoff models are approximately twice as long as those of the myopic models, similar to the response times in Ghosh & Verbrugge (online first). Furthermore, the proportions of *down* played and the relative response times adhere to what the strategies prescribe, and how many payoffs one has to look at according to the strategies. However, we also find that our myopic and own-payoff models respond much slower than the human participants, unlike the myopic and own-payoff models presented in Ghosh & Verbrugge (online first). Looking at the differences between our hand-made models and those in Ghosh & Verbrugge (online first), this suggests that focus actions in PRIMs are too slow to resemble human saccades. To test this, we train our models on focus actions, and find that their response times can fit those of the human participants after doing so, confirming that our models' focus actions are too slow, and that PRIMs models need to be trained on focus actions to generate realistic response times.

5.2.2 Our translation system

In Section 4.2.1 on page 54, we present our method of representing game trees in Java, which may be used in future work. In Section 4.2.2 on page 55, we present our method of representing strategies, represented in the logic of Ghosh & Verbrugge (online first), in Java. Unlike in Ghosh & Verbrugge (online first), our comparisons, such as $(1 \leqslant 2)$, refer to two specific utilities at specific leaves in the game tree, because otherwise our translation system would be unable to discern which utitilies should be compared. In Section 4.3 on page 57, we present our system which translates logical formulae, represented in Java, into PRIMs models. This system uses many useful Java methods that may be used in future work. One such method computes the BI and EFR strategies given a centipede game (or binary game), using the algorithm from Gradwohl & Heifetz (2011). Another method automatically generates the PRIMs representation for a game tree represented using our Java classes. We also propose a method that can be used to verify beliefs, by having the model try to retrieve chunks from declarative memory corresponding to the belief to be verified, similar to the method used in the models of Stevens et al. (2015). Lastly, we propose to sort propositions in a logical formula using the location they should be verified at in the game tree, as propositions in a logical conjunctions are unsorted, while PRIMs works through goals and operators in some order.

In Section 1.4 on page 15, we asked what the smallest elements of 'skill' are in our automatically generated models. In our system, these turn out to be the items within a conjunction of a strategy formula.

In Section 4.4.2 on page 62, we use our translation system to create myopic and own-payoff models in PRIMs based on the formulae in Ghosh & Verbrugge (online first). We find very similar results to our hand-made models presented in Section 3.3 on page 41, validating the automatically generated models created by our translation system.

In Section 4.5 on page 65, we extend our translation system by including exhaustive strategies, in which each possibility for a strategy in a game is listed using a logical formula. This system creates the corresponding PRIMs models by chaining the PRIMs models for each of these formulae together, moving its gaze back to the game tree's root node whenever necessary.

In Section 4.5.1 on page 66, we test the exhaustive PRIMs models using the BI and EFR strategies. Using a simple linear regression, we find that the response times of these models can be predicted using the number of formulae used to list all possibilities for the strategy.

5.3 Future work

In the present section, we discuss future directions of research based on our findings.

5.3.1 Behavioural research questions

The system we discuss in Chapter 4 on page 52 creates models which use working memory to store payoffs. These models can hold any number of payoffs in working memory, even though human participants may not. We use this assumption in our models because of the increase in model complexity when models can forget payoffs. To create more realistic models, a behavioural experiment could be performed to investigate whether human participants forget payoffs in a centipede game, and how they resolve this problem. To test this, one may use a method similar to the one used in Payne et al. (1993), where the payoffs in a centipede game are not visible to the participant unless they decide to 'uncover' them. A possible method would be to interrupt the player by covering a payoff and asking them what the payoff value was.

In Section 4.5 on page 65, we find that response times are a function of the number of possibilities a certain strategy has. This prediction can be experimentally tested, by letting human participants play a set of games designed to have a different number of possibilities for a set of commonly used strategies.

5.3.2 Problems in the formal logic

When creating our general translation system in Chapter 4, we encountered two problems while interpreting the logic to create PRIMs operators. The first of these is that values in comparisons such as $(1 \leqslant 2)$ do not refer to specific payoffs at specific leaves in the game tree, so interpretation is required to know which payoffs are being compared. Solving this problem may require changing the logic. One solution would be to use payoffs such as $\langle a^+ \rangle (U_C = 1)$ instead of numerical values within the comparison. Another would be to use variables instead of numerical values, such as $\langle a^+ \rangle (U_C = v_1)$ and $(v_1 \leqslant v_2)$. The latter allows for more general strategy formulae, whereas the former requires specific strategy formulae.

The second problem we encountered while interpreting the logic to create PRIMs operators occurred when translating beliefs such as $\mathbb{B}_{g1}^{(n2,P)} \langle d \rangle f$. This proposition with its belief operator states that at node *n2*, player P should believe that after a d move, f will be played. However, it does not state *how* this belief is obtained. Therefore some interpretation is required when creating a model that has to test this belief. Perhaps it would be better to use nested strategies, such as the ones in Ghosh et al. (2015), to formulate beliefs. Even though the notation $\mathbb{B}_{g1}^{(n2,P)} \langle d \rangle f$ is easier to read, it does force us to make assumptions about the nature of these beliefs when creating models. When using nested strategies for beliefs, no assumptions are required, because how the belief should be achieved is described in the nested strategy.

5.3.3 Cognitive modelling work

In Section 3.4 on page 48 and Section 3.5 on page 50, we found that our hand-made models are too slow to accurately predict response times. We believe this is because models in the PRIMs cognitive architecture are like children when gazing through a hierarchical display, in that they have never seen such a display before. This can be tested by conducting an experiment where children have to look through a hierarchical display, and comparing the results to those of PRIMs models. In future work, it will be useful to conduct a behavioural experiment to investigate how and how fast human participants look through hierarchical displays, such that its findings can be used to improve PRIMs in order to make focus actions more representative of human gazes, or to find a method of calculating the number of training trials that should yield realistic human response times, assuming that the remainder of the model is accurate. Because PRIMs arose from ACT-R, and ACT-R has a more advanced visual system (Anderson, Bothell, Byrne, Douglass, Lebiere & Qin, 2004), one could also use ACT-R's visual system as a starting point.

Another problem we encountered with the PRIMs visual system is that each focus action requires one operator to be performed, including all of its conditions and actions, which inflates response times. A solution may be to allow for multiple focus actions to be performed by a single operator, by performing them one by one based on which slot of the manual buffer they were placed in. For example, a single operator could place *focus-down* in both $AC1$, $AC2$, and $AC3$, and PRIMs would perform three *focus-down* actions. In the current version of PRIMs this is not possible, and only one focus action can be performed by an operator.

5.3.4 Bridging the gap

Although we have begun to bridge the gap between logic and cognitive modelling by creating a system that creates PRIMs models from logical formulae, we are not yet finished. Our system could be used to create more PRIMs models to be tested, and it can currently only translate a subset of the logic presented in Ghosh & Verbrugge (online first). We did not include disjunctions and negations in our translation system, leaving them as a subject for future work. We also assume that beliefs are only used when describing actions, and our operators are always ordered with beliefs first and steps last (such as $\mathbb{B}_{g1}^{(n2,P)}\langle d\rangle$), not allowing for different formats. In future work, one could try to extend our translation system or to create a new one which allows for all formulae possible in the logic of Ghosh & Verbrugge (online first).

Currently, our system translates formulae represented by Java classes, and these Java classes have to be created by writing the Java code required to do so by hand. Another important improvement could be the addition of a user interface which can be used to create games and formulae more easily.

Finally, we have found that PRIMs models become very large when automatically generating all possibilities a strategy prescribes. Some parts of these possibilities overlap, and some PRIMs operators are identical. Due to the high number of operators and due to activation noise, models become prone to unexpected behaviour as they get larger, and smaller models are preferred. Furthermore, models of such a large size are very difficult to intuitively understand.

Perhaps it could be useful to create a hybrid system, which uses PRIMs' chunks, production compilation, and activation formulae, but also allows for processes outside of operators. Such a hybrid system can move through conjuncts and strategy formulae without requiring a large number of goals and operators for each possibility. It can also keep track of which conjuncts within a strategy's formulae have already been checked, such that conjuncts present in multiple formulae do not have to be checked twice without having to create even more goals and operators. Furthermore, it can be used to greatly reduce the time required to gaze through the game tree. Another problem that can be solved by such a hybrid system is the fact that we need, for each goal that corresponds to a proposition, another goal that moves the focus back to the root of the tree if this proposition turns out to be false. Within a hybrid system, a general solution could be used for this problem.

Another question we encountered during this research is 'Is it possible to automatically generate logical formulae from models in the PRIMs cognitive architecture?' Research such as Gall &

Frühwirth (2015) and Gall & Frühwirth (2016) has shown that it is indeed possible to formalize models in the ACT-R cognitive architecture by hand. Due to the similarities between PRIMs and ACT-R we suspect that this should also be possible for the PRIMs cognitive architecture. Models in the PRIMs cognitive architecture use condition-action rules to operate, which are already quite similar to logical if-then statements. A system that automatically generates logical formulae based on cognitive models could be used to formalize existing models, such that their strategies can be intuitively understood, or compared to strategy formulae based on human behaviour.

5.4 Conclusion

The main goal of this thesis is to help understand human behaviour, in our case in centipede games, a subset of dynamic-perfect information games. On one hand, game theorists use formal logic to describe human behaviour. On the other hand, cognitive scientists use cognitive models to predict human behaviour. To help understand human behaviour, we created a general translation system which can use formalizations of strategies to automatically generate cognitive models of the human mind. The place of our work in this continuing body of research can be found in Figure 5.1, and is indicated with a red arrow. Here, each arrow is a line of research. In Figure

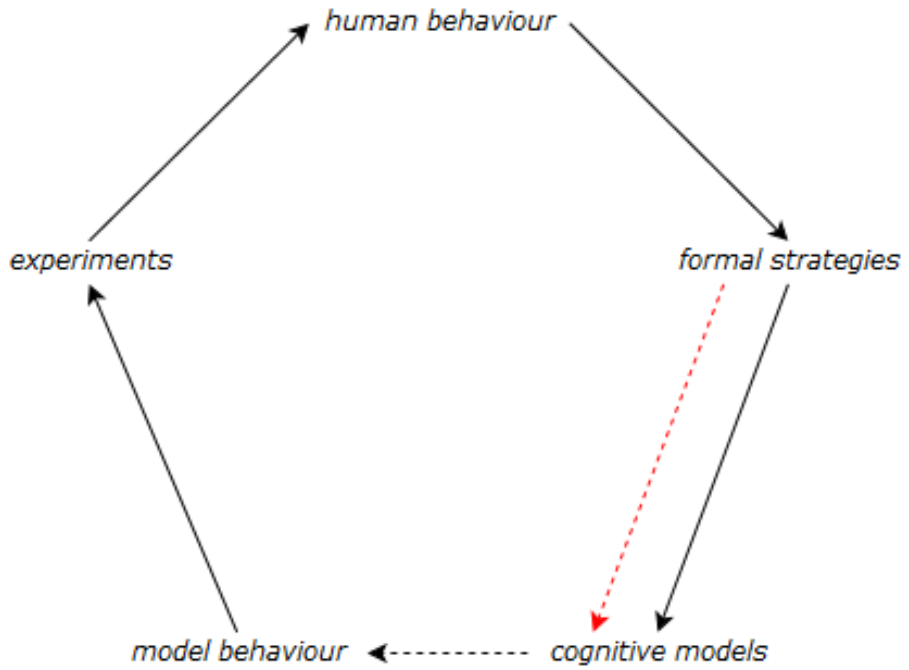

Figure 5.1: A schematic diagram of how our work fits into existing research. Dashed lines indicate automated processes.

5.1, human behaviour can be found at the top of the diagram, as all research involved aims to understand human behaviour. By observing human behaviour, game theorists use formal modelling to formalize strategies that are being used by human participants. These formal strategies can then be used by cognitive scientists to manually construct cognitive models of the human mind. These models can then automatically generate response times, decisions made, brain activity, and loci of attention. The behaviour of such a model (as well as the strategy formula the model is based on) can then be verified by constructing a behavioural experiment. From these behavioural experiments, data about human behaviour can be obtained, which closes the circle.

In this thesis we describe how we succeeded in creating a general translation system which creates cognitive models from formal strategies. This system can use any logical formula in the subset of the logic we are using (Ghosh & Verbrugge, online first) to automatically generate fully

functioning models in the PRIMs cognitive architecture (Taatgen, 2013b). This opens up a new line between formal strategies and cognitive modelling. Formerly, researchers had to manually create cognitive models that are based on strategy formulae. Our system shows that this process can be automated, such that only formal strategies have to be created manually before model behaviour can be obtained automatically.

The models our system creates are fully functioning PRIMs models. We verified our system by comparing these models to models we made by hand, and found similar response times and actions. Our system shows that not all PRIMs models have to be made by hand, and automatically generating models can save a lot of time, as well as ensure consistency between different models.It also shows how strategies in a formal logic can be used to predict response times and actions. As an example of what our system could be used for, we have successfully used our system to make predictions about human behaviour based on the backward induction and extensive-form rationalizable strategies. When looking at the literature, our system seems to be the first effort at automatically generating cognitive models in PRIMs based on logical formulae, and we are eager to see more research in this area in the future.

Appendices

A: Original myopic and own-payoff models

This section contains the full myopic and own-payoff models created for Ghosh & Verbrugge (online first). The myopic model is as follows:

```
define goal find {
    operator find−location−1{
        V1<>nil
        WM1=nil
        ==>
        V1−>WM1
    }

    operator find−location−2 {
        V2<>nil
        WM1<>nil
        WM3=nil
        ==>
        V2−>WM3
        compare−>G1
    }
}

define goal compare {
    operator check−largest−payoff−retrieve {
        WM1<>nil
        WM3<>nil
        ==>
        WM1−>RT1
        WM3−>RT3
    }

    operator check−largest−payoff−bigger {
        RT1<>nil
        RT2=GC1
        RT3<>nil
        ==>
        RT2−>WM2
        play−>AC1
        left−>AC2
        done−>G1
    }

    operator check−largest−payoff−not−bigger {
```

```
        RT1<>nil
        RT2<>GC1
        RT3<>nil
        ==>
        RT2−>WM2
        play−>ac1
        right−>ac2
        done−>G1
    }
}

define goal next {
    operator last−comparison {
        V1=done
        WM2<>nil
        ==>
        respond−>ac1
        anything−>ac2
    }

    operator new−comparison {
        V1<>done
        WM2<>nil
        ==>
        nil−>WM1
        nil−>WM2
        nil−>WM3
        find−>G1
    }

}
```

The own-payoff model is as follows:

```
define goal find {
    operator find−location−1{
        V1<>nil
        WM1=nil
        ==>
        V1−>WM1
    }

    operator find−location−2 {
        V2<>nil
        WM1<>nil
        WM3=nil
        ==>
        V2−>WM3
        compare−>G1
    }
}

define goal compare {
    operator check−largest−payoff−retrieve {
        WM1<>nil
```

```
        WM3<>nil
        ==>
        WM1->RT1
        WM3->RT3
    }

    operator check-largest-payoff-bigger {
        RT1<>nil
        RT2=GC1
        RT3<>nil
        ==>
        RT2->WM2
        focus->AC1
        next->AC2
        next->G1
    }

    operator check-largest-payoff-not-bigger {
        RT1<>nil
        RT2<>GC1
        RT3<>nil
        ==>
        RT2->WM2
        play->AC1
        right->AC2
    }
}

define goal next {
    operator last-comparison {
        V1=done
        WM2<>nil
        ==>
        play->ac1
        left->ac2
        done->G1
    }

    operator new-comparison {
        V1<>done
        WM2<>nil
        ==>
        nil->WM1
        nil->WM2
        nil->WM3
        find->G1
    }

}
```

B: Our myopic and own-payof models

This section contains the full myopic and own-payoff models we created in PRIMs, including comments. The myopic model is as follows:

```
//Myopic Forward Reasoning model by Jordi Top (s2402319)
//Plays centipede games by comparing its own current payoff to its next payoff.
//Games included are the following:

//Game 1: (4,1) (1,2) (3,1) (1,4) (6,3) C−P−C−P
//Game 2: (2,1) (1,2) (3,1) (1,4) (6,3) C−P−C−P
//Game 3: (4,1) (1,2) (3,1) (1,4) (6,4) C−P−C−P
//Game 4: (2,1) (1,2) (3,1) (1,4) (6,4) C−P−C−P
//Game 1': (1,2) (3,1) (1,4) (6,3) P−C−P
//Game 3': (1,2) (3,1) (1,4) (6,4) P−C−P

//a,c,e,g = down, game ends. h = right, game ends
//b, d, f = right, game continues.

//Representation of game 1:
//screen(
//["node","C","notcurrent",
//["node","P","current",
//["node","C","notcurrent",
//["node","P","notcurrent",["leaf",6,3],["leaf",1,4]],
//["leaf",3,1]],
//["leaf",1,2]],
//["leaf",4,1]])


define task MyopicForwardReasoningJordi {
        initial−goals: (findcurrent) //Start by finding where the token is
        goals: (findvals compare preparenext)
        task−constants: (bigger not c p current notcurrent node leaf root end down right
            ↪ comparison)
        imaginal−autoclear: nil
        default−activation: 1.0 //Minimum activation of pre−existing chunk
        rt: −2.0 //Retrieval threshold
        ol: t //Optimized learning
        batch−trace: t
        default−operator−self−assoc: 0 //Default −1.0, self−association of operator
        default−operator−assoc: 4 //Default 4.0, association between operator and goal
}

//Find where the token currently is
define goal findcurrent{

        //If you are looking at the current node, move to goal "findvals"
        operator found−current−node {
                V1 = node
                WM1 = nil
                V3 = current
        ==>
                findvals −> G1
        }
```

```
        //If you are not looking at the current node, focus down (at the next node)
        operator not−found−current−node {
                V1 = node
                WM1 = nil
                V3 = notcurrent
        ==>
                focusdown −> AC1
        }
}

//Find payoffs at the current and next node
define goal findvals{

        //If you are at the current node, focus at its first leaf
        operator init−read−current {
                V1 = node
                V3 = current
                WM1 = nil
                WM3 = nil
        ==>
                focus−down−last −> AC1
        }

        //Store your own payoff at the current node in WM1
        operator read−current−leaf {
                V1 = leaf
                WM1 = nil
                WM3 = nil
        ==>
                V3 −> WM1
                focus−up−stay −> AC1
        }

        operator move−next−node {
                V1 = node
                V3 = current
                WM1 <> nil
                WM3 = nil
        ==>
                focus−down −> AC1
        }

        //Move to the next leaf
        operator find−next−node {
                V1 = node
                V3 <> current
                WM1 <> nil
                WM3 = nil
        ==>
                focus−down−last −> AC1
        }

        //Store your own payoff for the next leaf in WM3 and move to goal 'compare'
        operator find−next−leaf {
```

```
                V1 = leaf
                WM1 <> nil
                WM3 = nil
        ==>
                V3 -> WM3
                focus-up-stay -> AC1
                compare -> G1
        }
}

//Compare two payoffs
define goal compare {
        //Retrieve a 'bigger'-fact with your own current and next payoffs in slot values RT1
            ↪ and RT3
        operator start-retrieval {
                WM1 <> nil
                WM3 <> nil
                RT1 = nil
        ==>
                WM1 -> RT1
                WM3 -> RT3
                comparison -> RT4
        }

        //If we retrieve that the first is bigger, play down, and prepare start with goal '
            ↪ preparenext'
        operator first-bigger {
                RT1 <> nil
                RT2 = bigger
                RT3 <> nil
                RT4 = comparison
        ==>
                play -> AC1
                down -> AC2
                done -> WM2
                preparenext -> G1
        }

        //If the second is bigger, play right, and prepare start with goal 'preparenext'
        operator second-bigger {
                RT1 <> nil
                RT2 = not
                RT3 <> nil
                RT4 = comparison
        ==>
                play -> AC1
                right -> AC2
                done -> WM2
                preparenext -> G1
        }
}

//Prepare for a next turn
define goal preparenext {
```

```
        //Clear working memory and start at 'findvals' again
        operator reset-game {
                WM2 = done
        ==>
                nil -> WM1
                nil -> WM2
                nil -> WM3
                findvals -> G1
        }
}

//Store for all combinations of 0 through 9 whether the first is larger in memory
define init-script {
    for i in 1 to 6 {
        for j in 1 to 6 {
            if (i > j) {
                name = "larger-" + i
                name = name + "-"
                name = name + j
                add-dm(name, i, "bigger", j, "comparison")
            } else {
                name = "notlarger-" + i
                name = name + "-"
                name = name + j
                add-dm(name, i, "not", j, "comparison")
            }
        }
    }
}

define script {

        params = batch-parameters() //Allows reading parameters from a batch file
        if(params == "NA"){ //If those are not present,
                params = [1,"b","f"] //Use game 1 where player C will play "b" and "f" (see
                    ↪ above for all games). Change to 1,2,3,4,11,13,a,b,e or f, to change the
                    ↪ game.
        }
        if((params[0] == 11) || (params[0] == 31)){ //If we're using a pruned game, set player
            ↪ C's first move to "C" (to prevent the "a" if-clause from firing)
                params[1] = "b"
        }
        trial-start()
        if(params[1] == "a"){ //If C plays "a", end the game
                issue-reward(1)
                trial-end()
        }
        if(params[1] == "b"){ //If C plays "b", or if the pruned games are used...
                if(params[0] == 1){ //Load game 1
                        screen(
                        ["node","c","notcurrent","root","notend",
                        ["node","p","current","notroot","notend",
                        ["node","c","notcurrent","notroot","notend",
                        ["node","p","notcurrent","notroot","end",["leaf",6,3],["leaf",1,4]],
                        ["leaf",3,1]],
```

```
                ["leaf",1,2]],
                ["leaf",4,1]])
        }
        if(params[0] == 2){ //Load game 2
                screen(
                ["node","c","notcurrent","root","notend",
                ["node","p","current","notroot","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","notcurrent","notroot","end",["leaf",6,3],["leaf",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],
                ["leaf",2,1]])
        }
        if(params[0] == 3){ //Load game 3
                screen(
                ["node","c","notcurrent","root","notend",
                ["node","p","current","notroot","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","notcurrent","notroot","end",["leaf",6,4],["leaf",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],
                ["leaf",4,1]])
        }
        if(params[0] == 4){ //Load game 4
                screen(
                ["node","c","notcurrent","root","notend",
                ["node","p","current","notroot","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","notcurrent","notroot","end",["leaf",6,4],["leaf",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],
                ["leaf",2,1]])
        }
        if(params[0] == 11){ //Load game 1'
                screen(
                ["node","p","current","root","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","notcurrent","notroot","end",["leaf",6,3],["leaf",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],)
        }
        if(params[0] == 31){ //Load game 3'
                screen(
                ["node","p","current","root","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","notcurrent","notroot","end",["leaf",6,4],["leaf",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],)
        }
}

set-data-file-field(0, "firstmove")
run-until-action("play") //Display the game until the player plays an action
ac = last-action()
if(ac[0]=="play"){
```

```
if(ac[1]==”down”){ //If the player moves down, end the game
        issue−reward(2)
        trial−end()
}
if(ac[1]==”right”){
        if(params[2]==”e”){ //If the player moves right and the computer
            ↪ moves down afterwards, end the game
            issue−reward(1)
            trial−end()
        }
        if(params[2]==”f”){ //If the player moves right and the computer
            ↪ moves right, change the location of the token
            if(params[0] == 1){
                screen(
                [”node”,”c”,”notcurrent”,”root”,”notend”,
                [”node”,”p”,”notcurrent”,”notroot”,”notend”,
                [”node”,”c”,”notcurrent”,”notroot”,”notend”,
                [”node”,”p”,”current”,”notroot”,”end”,[”leaf”,6,3],[”leaf
                    ↪ ”,1,4]],
                [”leaf”,3,1]],
                [”leaf”,1,2]],
                [”leaf”,4,1]])
            }
            if(params[0] == 2){
                screen(
                [”node”,”c”,”notcurrent”,”root”,”notend”,
                [”node”,”p”,”notcurrent”,”notroot”,”notend”,
                [”node”,”c”,”notcurrent”,”notroot”,”notend”,
                [”node”,”p”,”current”,”notroot”,”end”,[”leaf”,6,3],[”leaf
                    ↪ ”,1,4]],
                [”leaf”,3,1]],
                [”leaf”,1,2]],
                [”leaf”,2,1]])
            }
            if(params[0] == 3){
                screen(
                [”node”,”c”,”notcurrent”,”root”,”notend”,
                [”node”,”p”,”notcurrent”,”notroot”,”notend”,
                [”node”,”c”,”notcurrent”,”notroot”,”notend”,
                [”node”,”p”,”current”,”notroot”,”end”,[”leaf”,6,4],[”leaf
                    ↪ ”,1,4]],
                [”leaf”,3,1]],
                [”leaf”,1,2]],
                [”leaf”,4,1]])
            }
            if(params[0] == 4){
                screen(
                [”node”,”c”,”notcurrent”,”root”,”notend”,
                [”node”,”p”,”notcurrent”,”notroot”,”notend”,
                [”node”,”c”,”notcurrent”,”notroot”,”notend”,
                [”node”,”p”,”current”,”notroot”,”end”,[”leaf”,6,4],[”leaf
                    ↪ ”,1,4]],
                [”leaf”,3,1]],
                [”leaf”,1,2]],
                [”leaf”,2,1]])
```

```
                                }
                                if(params[0] == 11){
                                        screen(
                                        [”node”,”p”,”notcurrent”,”root”,”notend”,
                                        [”node”,”c”,”notcurrent”,”notroot”,”notend”,
                                        [”node”,”p”,”current”,”notroot”,”end”,[”leaf”,6,3],[”leaf
                                            ↪ ”,1,4]],
                                        [”leaf”,3,1]],
                                        [”leaf”,1,2]],)
                                }
                                if(params[0] == 31){
                                        screen(
                                        [”node”,”p”,”notcurrent”,”root”,”notend”,
                                        [”node”,”c”,”notcurrent”,”notroot”,”notend”,
                                        [”node”,”p”,”current”,”notroot”,”end”,[”leaf”,6,4],[”leaf
                                            ↪ ”,1,4]],
                                        [”leaf”,3,1]],
                                        [”leaf”,1,2]],)
                                }


                        }
                }
        }

        set−data−file−field(0, ”secondmove”)
        run−until−action(”play”) //Play until the player has taken his second turn
        ac = last−action() //Issue a reward based on the player’s last action and the game
            ↪ being played
        if(ac[0]==”play”){
                if(ac[1]==”down”){
                        issue−reward(4)
                        trial−end()
                }
                if(ac[1]==”right”){
                        if((params[0] == 1) || (params[0] == 2) || (params[0] == 11)){
                                issue−reward(3)
                        }
                        if((params[0] == 3) || (params[0] == 4) || (params[0] == 31)){
                                issue−reward(4)
                        }
                        trial−end()
                }
        }
}
```

The own-payoff model is as follows:

```
//Forward Reasoning model by Jordi Top (s2402319)
//Plays centipede games by comparing its own current payoff to all of its next payoffs.
//Games included are the following:

//Game 1: (4,1) (1,2) (3,1) (1,4) (6,3) C−P−C−P
//Game 2: (2,1) (1,2) (3,1) (1,4) (6,3) C−P−C−P
//Game 3: (4,1) (1,2) (3,1) (1,4) (6,4) C−P−C−P
```

```
//Game 4: (2,1) (1,2) (3,1) (1,4) (6,4) C−P−C−P
//Game 1': (1,2) (3,1) (1,4) (6,3) P−C−P
//Game 3': (1,2) (3,1) (1,4) (6,4) P−C−P

//a,c,e,g = down, game ends. h = right, game ends
//b, d, f = right, game continues.

//Representation of game 1:
//screen(
//["node","C","notcurrent",
//["node","P","current",
//["node","C","notcurrent",
//["node","P","notcurrent",["leaf",6,3],["leaf",1,4]],
//["leaf",3,1]],
//["leaf",1,2]],
//["leaf",4,1]])

define task ForwardReasoningJordi {
        initial−goals: (findcurrent) //Start by finding where the token is
        goals: (findvals compare preparenext)
        task−constants: (bigger not c p current notcurrent node leaf root end down right
            ↪ comparison)
        imaginal−autoclear: nil
        default−activation: 1.0 //Minimum activation of pre−existing chunk
        rt: −2.0 //Retrieval threshold
        ol: t //Optimized learning
        batch−trace: t
        default−operator−self−assoc: 0 //Default −1.0, self−association of operator
}

//Find where the token currently is
define goal findcurrent{

        //If you are looking at the current node, move to goal "findvals"
        operator found−current−node {
                V1 = node
                WM1 = nil
                V3 = current
        ==>
                findvals −> G1
        }

        //If you are not looking at the current node, focus down (at the next node)
        operator not−found−current−node {
                V1 = node
                WM1 = nil
                V3 = notcurrent
        ==>
                focusdown −> AC1
        }
}

//Find payoffs at the current and next node(s)
define goal findvals{
```

```
//If you are at the current node, focus at its first leaf
operator init−read−current {
        V1 = node
        WM1 = nil
        WM3 = nil
==>
        focus−down−last −> AC1
}

//Store your own payoff at the current node in WM1
operator read−current−leaf {
        V1 = leaf
        WM1 = nil
        WM3 = nil
==>
        V3 −> WM1
        focus−up−stay −> AC1
}

operator move−next−node {
        V1 = node
        V5 <> end
        WM1 <> nil
        WM2 = nil
        WM3 = nil
==>
        focus−down −> AC1
        notcurrent −> WM2
}

//Move to the next leaf
operator find−next−node {
        V1 = node
        WM1 <> nil
        WM2 = notcurrent
        WM3 = nil
==>
        focus−down−last −> AC1
}

//Store your own payoff for the next leaf in WM3 and move to goal 'compare'
operator find−next−leaf {
        V1 = leaf
        WM1 <> nil
        WM2 = notcurrent
        WM3 = nil
==>
        V3 −> WM3
        focus−up−stay −> AC1
        compare −> G1
}

operator move−next−node−end−of−tree {
        V1 = node
        V5 = end
```

```
                WM1 <> nil
                WM2 = nil
                WM3 = nil
        ==>
                focus-down -> AC1
                end -> WM2
        }

        operator find-next-leaf-end-of-tree {
                V1 = leaf
                WM1 <> nil
                WM2 = end
                WM3 = nil
        ==>
                V3 -> WM3
                focus-up-stay -> AC1
                compare -> G1
        }
}

//Compare two payoffs
define goal compare {

        //Retrieve a 'bigger'-fact with your own current and next payoffs in slot values RT1
            ↪ and RT3
        operator start-retrieval {
                WM1 <> nil
                WM3 <> nil
                RT1 = nil
        ==>
                WM1 -> RT1
                WM3 -> RT3
                comparison -> RT4
        }

        //If we retrieve that the first is bigger, don't play yet, because there may still be bigger
            ↪ payoffs later.
        operator first-bigger {
                RT1 <> nil
                RT2 = bigger
                RT3 <> nil
                RT4 = comparison
                WM2 <> end
        ==>
                nil -> WM2
                nil -> WM3
                findvals -> G1
        }

        operator first-bigger-end-of-tree {
                RT1 <> nil
                RT2 = bigger
                RT3 <> nil
                RT4 = comparison
                WM2 = end
```

```
        ==>
                play -> AC1
                down -> AC2
                done -> WM2
                preparenext -> G1
        }

        //If the second is bigger, play right, and prepare start with goal 'preparenext'
        operator second-bigger {
                RT1 <> nil
                RT2 = not
                RT3 <> nil
                RT4 = comparison
        ==>
                play -> AC1
                right -> AC2
                done -> WM2
                preparenext -> G1
        }
}

//Prepare for a next turn
define goal preparenext {

        //Clear working memory and start at 'findvals' again
        operator reset-game {
                WM2 = done
        ==>
                nil -> WM1
                nil -> WM2
                nil -> WM3
                findcurrent -> G1
        }
}

//Store for all combinations of 0 through 9 whether the first is larger in memory
define init-script {
    for i in 1 to 6 {
        for j in 1 to 6 {
            if (i > j) {
                name = "larger-" + i
                name = name + "-"
                name = name + j
                add-dm(name, i, "bigger", j, "comparison")
            } else {
                name = "notlarger-" + i
                name = name + "-"
                name = name + j
                add-dm(name, i, "not", j, "comparison")
            }
        }
    }
}

define script {
```

```
params = batch−parameters() //Allows reading parameters from a batch file
if(params == "NA"){ //If those are not present,
        params = [1,"b","f"] //Use game 1 where player C will play "b" and "f" (see
            ↪ above for all games). Change to 1,2,3,4,11,13,a,b,e or f, to change the
            ↪ game.
}
if((params[0] == 11) || (params[0] == 31)){ //If we're using a pruned game, set player
    ↪ C's first move to "C" (to prevent the "a" if−clause from firing)
        params[1] = "b"
}
trial−start()
if(params[1] == "a"){ //If C plays "a", end the game
        issue−reward(1)
        trial−end()
}
if(params[1] == "b"){ //If C plays "b", or if the pruned games are used...
        if(params[0] == 1){ //Load game 1
                screen(
                ["node","c","notcurrent","root","notend",
                ["node","p","current","notroot","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","notcurrent","notroot","end",["leaf",6,3],["leaf",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],
                ["leaf",4,1]])
        }
        if(params[0] == 2){ //Load game 2
                screen(
                ["node","c","notcurrent","root","notend",
                ["node","p","current","notroot","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","notcurrent","notroot","end",["leaf",6,3],["leaf",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],
                ["leaf",2,1]])
        }
        if(params[0] == 3){ //Load game 3
                screen(
                ["node","c","notcurrent","root","notend",
                ["node","p","current","notroot","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","notcurrent","notroot","end",["leaf",6,4],["leaf",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],
                ["leaf",4,1]])
        }
        if(params[0] == 4){ //Load game 4
                screen(
                ["node","c","notcurrent","root","notend",
                ["node","p","current","notroot","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","notcurrent","notroot","end",["leaf",6,4],["leaf",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],
```

```
                ["leaf",2,1]])
        }
        if(params[0] == 11){ //Load game 1'
                screen(
                ["node","p","current","root","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","notcurrent","notroot","end",["leaf",6,3],["leaf",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],)
        }
        if(params[0] == 31){ //Load game 3'
                screen(
                ["node","p","current","root","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","notcurrent","notroot","end",["leaf",6,4],["leaf",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],)
        }
}

set-data-file-field(0, "firstmove")
run-until-action("play") //Display the game until the player plays an action
ac = last-action()
if(ac[0]=="play"){
        if(ac[1]=="down"){ //If the player moves down, end the game
                issue-reward(2)
                trial-end()
        }
        if(ac[1]=="right"){
                if(params[2]=="e"){ //If the player moves right and the computer
                    ↪ moves down afterwards, end the game
                    issue-reward(1)
                    trial-end()
                }
                if(params[2]=="f"){ //If the player moves right and the computer
                    ↪ moves right, change the location of the token
                    if(params[0] == 1){
                            screen(
                            ["node","c","notcurrent","root","notend",
                            ["node","p","notcurrent","notroot","notend",
                            ["node","c","notcurrent","notroot","notend",
                            ["node","p","current","notroot","end",["leaf",6,3],["leaf
                                ↪ ",1,4]],
                            ["leaf",3,1]],
                            ["leaf",1,2]],
                            ["leaf",4,1]])
                    }
                    if(params[0] == 2){
                            screen(
                            ["node","c","notcurrent","root","notend",
                            ["node","p","notcurrent","notroot","notend",
                            ["node","c","notcurrent","notroot","notend",
                            ["node","p","current","notroot","end",["leaf",6,3],["leaf
                                ↪ ",1,4]],
                            ["leaf",3,1]],
```

```
                ["leaf",1,2]],
                ["leaf",2,1]])
        }
        if(params[0] == 3){
                screen(
                ["node","c","notcurrent","root","notend",
                ["node","p","notcurrent","notroot","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","current","notroot","end",["leaf",6,4],["leaf
                    ↪ ",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],
                ["leaf",4,1]])
        }
        if(params[0] == 4){
                screen(
                ["node","c","notcurrent","root","notend",
                ["node","p","notcurrent","notroot","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","current","notroot","end",["leaf",6,4],["leaf
                    ↪ ",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],
                ["leaf",2,1]])
        }
        if(params[0] == 11){
                screen(
                ["node","p","notcurrent","root","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","current","notroot","end",["leaf",6,3],["leaf
                    ↪ ",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],)
        }
        if(params[0] == 31){
                screen(
                ["node","p","notcurrent","root","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","current","notroot","end",["leaf",6,4],["leaf
                    ↪ ",1,4]],
                ["leaf",3,1]],
                ["leaf",1,2]],)
        }


                }
        }
}

set-data-file-field(0, "secondmove")
run-until-action("play") //Play until the player has taken his second turn
ac = last-action() //Issue a reward based on the player's last action and the game
    ↪ being played
if(ac[0]=="play"){
        if(ac[1]=="down"){
```

```
                        issue-reward(4)
                        trial-end()
                }
                if(ac[1]=="right"){
                        if((params[0] == 1) || (params[0] == 2) || (params[0] == 11)){
                                issue-reward(3)
                        }
                        if((params[0] == 3) || (params[0] == 4) || (params[0] == 31)){
                                issue-reward(4)
                        }
                        trial-end()
                }
        }
}
```

C: Training models for our myopic and own-payoff models

This section contains the models used to train our myopic and own-payoff models in performing focus actions. The training model for the myopic model is as follows:

```
//Gaze training task by Jordi Top (s2402319)
//Gazes through centipede games.

//Representation of game:
//screen(
//["node","C","notcurrent",
//["node","P","current",
//["node","C","notcurrent",
//["node","P","notcurrent",["leaf",0,0],["leaf",0,0]],
//["leaf",0,0]],
//["leaf",0,0]],
//["leaf",0,0]])

define task GazeTrainTaskMyopic {
        initial-goals: (findcurrent) //Start by finding where the token
            ↪ is
        goals: (findvals compare preparenext)
        task-constants: (bigger not c p current notcurrent node leaf
            ↪ root end down right comparison)
        imaginal-autoclear: nil
        default-activation: 1.0 //Minimum activation of pre-existing
            ↪ chunk
        rt: -2.0 //Retrieval threshold
        ol: t //Optimized learning
        batch-trace: t
        default-operator-self-assoc: 0 //Default -1.0, self-association
            ↪ of operator
        default-operator-assoc: 4 //Default 4.0, association between
            ↪ operator and goal
}

//Find where the token currently is
define goal findcurrent{
```

```
        //If you are looking at the current node, move to goal "
           ↪ findvals"
        operator found-current-node {
                V1 = node
                WM1 = nil
                V3 = current
        ==>
                findvals -> G1
        }

        //If you are not looking at the current node, focus down (at
           ↪ the next node)
        operator not-found-current-node {
                V1 = node
                WM1 = nil
                V3 = notcurrent
        ==>
                focusdown -> AC1
        }
}

//Find payoffs at the current and next node
define goal findvals{

        //If you are at the current node, focus at its first leaf
        operator init-read-current {
                V1 = node
                V3 = current
                WM1 = nil
                WM3 = nil
        ==>
                focus-down-last -> AC1
        }

        //Store your own payoff at the current node in WM1
        operator read-current-leaf {
                V1 = leaf
                WM1 = nil
                WM3 = nil
        ==>
                V3 -> WM1
                focus-up-stay -> AC1
        }

        operator move-next-node {
                V1 = node
                V3 = current
                WM1 <> nil
                WM3 = nil
        ==>
                focus-down -> AC1
        }

        //Move to the next leaf
        operator find-next-node {
```

```
                V1 = node
                V3 <> current
                WM1 <> nil
                WM3 = nil
        ==>
                focus-down-last -> AC1
        }

        //Store your own payoff for the next leaf in WM3 and move to
            ↪ goal 'compare'
        operator find-next-leaf {
                V1 = leaf
                WM1 <> nil
                WM3 = nil
        ==>
                V3 -> WM3
                focus-up-stay -> AC1
                compare -> G1
        }
}

//Compare two payoffs
define goal compare {

        operator play-right {
                G1 = compare
        ==>
                play -> AC1
                right -> AC2
                done -> WM2
                preparenext -> G1
        }

}

//Prepare for a next turn
define goal preparenext {

        //Clear working memory and start at 'findvals' again
        operator reset-game {
                WM2 = done
        ==>
                nil -> WM1
                nil -> WM2
                nil -> WM3
                findvals -> G1
        }
}

define init-script {

}

define script {
```

```
        params = batch-parameters()
        trial-start()

        screen(
                ["node","c","notcurrent","root","notend",
                ["node","p","current","notroot","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","notcurrent","notroot","end",["leaf
                    ↪ ",0,0],["leaf",0,0]],
                ["leaf",0,0]],
                ["leaf",0,0]],
                ["leaf",0,0]])


        run-until-action("play") //Display the game until the player
            ↪ continues

        screen(
                ["node","c","notcurrent","root","notend",
                ["node","p","notcurrent","notroot","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","current","notroot","end",["leaf",0,0],["
                    ↪ leaf",0,0]],
                ["leaf",0,0]],
                ["leaf",0,0]],
                ["leaf",0,0]])

        run-until-action("play") //Play until the player continues

        issue-reward()
        trial-end()
}
```

The training model for the own-payoff model is as follows:

```
//Gaze training task by Jordi Top (s2402319)
//Gazes through centipede games.

//Representation of game:
//screen(
//["node","C","notcurrent",
//["node","P","current",
//["node","C","notcurrent",
//["node","P","notcurrent",["leaf",0,0],["leaf",0,0]],
//["leaf",0,0]],
//["leaf",0,0]],
//["leaf",0,0]])

define task GazeTrainTask {
        initial-goals: (findcurrent) //Start by finding where the token
            ↪  is
        goals: (findvals compare preparenext)
        task-constants: (bigger not c p current notcurrent node leaf
            ↪ root end down right comparison)
        imaginal-autoclear: nil
```

```
        default-activation: 1.0 //Minimum activation of pre-existing
            ↪ chunk
        rt: -2.0 //Retrieval threshold
        ol: t //Optimized learning
        batch-trace: t
        default-operator-self-assoc: 0 //Default -1.0, self-association
            ↪  of operator
}

//Find where the token currently is
define goal findcurrent{

        //If you are looking at the current node, move to goal "
            ↪ findvals"
        operator found-current-node {
                V1 = node
                WM1 = nil
                V3 = current
        ==>
                findvals -> G1
        }

        //If you are not looking at the current node, focus down (at
            ↪ the next node)
        operator not-found-current-node {
                V1 = node
                WM1 = nil
                V3 = notcurrent
        ==>
                focusdown -> AC1
        }
}

//Find payoffs at the current and next node(s)
define goal findvals{

        //If you are at the current node, focus at its first leaf
        operator init-read-current {
                V1 = node
                WM1 = nil
                WM3 = nil
        ==>
                focus-down-last -> AC1
        }

        //Store your own payoff at the current node in WM1
        operator read-current-leaf {
                V1 = leaf
                WM1 = nil
                WM3 = nil
        ==>
                V3 -> WM1
                focus-up-stay -> AC1
        }
```

```
operator move-next-node {
        V1 = node
        V5 <> end
        WM1 <> nil
        WM2 = nil
        WM3 = nil
==>
        focus-down -> AC1
        notcurrent -> WM2
}

//Move to the next leaf
operator find-next-node {
        V1 = node
        WM1 <> nil
        WM2 = notcurrent
        WM3 = nil
==>
        focus-down-last -> AC1
}

//Store your own payoff for the next leaf in WM3 and move to
   ↪ goal 'compare'
operator find-next-leaf {
        V1 = leaf
        WM1 <> nil
        WM2 = notcurrent
        WM3 = nil
==>
        V3 -> WM3
        focus-up-stay -> AC1
        compare -> G1
}

operator move-next-node-end-of-tree {
        V1 = node
        V5 = end
        WM1 <> nil
        WM2 = nil
        WM3 = nil
==>
        focus-down -> AC1
        end -> WM2
}

operator find-next-leaf-end-of-tree {
        V1 = leaf
        WM1 <> nil
        WM2 = end
        WM3 = nil
==>
        V3 -> WM3
        focus-up-stay -> AC1
        compare -> G1
}
```

```
}

//Compare two payoffs
define goal compare {

        operator dont-play {
                G1 = compare
                WM2 <> end
        ==>
                nil -> WM2
                nil -> WM3
                findvals -> G1
        }

        operator play-right {
                G1 = compare
                WM2 = end
        ==>
                play -> AC1
                right -> AC2
                done -> WM2
                preparenext -> G1
        }
}

//Prepare for a next turn
define goal preparenext {

        //Clear working memory and start at 'findvals' again
        operator reset-game {
                WM2 = done
        ==>
                nil -> WM1
                nil -> WM2
                nil -> WM3
                findcurrent -> G1
        }
}

define init-script {

}

define script {

        params = batch-parameters()
        trial-start()

        screen(
                ["node","c","notcurrent","root","notend",
                ["node","p","current","notroot","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","notcurrent","notroot","end",["leaf
                    ↪ ",0,0],["leaf",0,0]],
                ["leaf",0,0]],
```

```
                ["leaf",0,0]],
                ["leaf",0,0]])


        run-until-action("play") //Display the game until the player
            ↪ continues

        screen(
                ["node","c","notcurrent","root","notend",
                ["node","p","notcurrent","notroot","notend",
                ["node","c","notcurrent","notroot","notend",
                ["node","p","current","notroot","end",["leaf",0,0],["
                    ↪ leaf",0,0]],
                ["leaf",0,0]],
                ["leaf",0,0]],
                ["leaf",0,0]])

        run-until-action("play") //Play until the player continues

        issue-reward()
        trial-end()
}
```

D: Automatically generated own-payoff model

This section contains the own-payoff model for Game 1′ of Ghosh et al. (2017) that has automatically been generated by our translation system.

```
//Automatically generated model
//Created using a Java system created by Jordi Top
//PRIMs model created from a logical formula represented in the logic
    ↪ presented in
//"Studying strategies and types of players: Experiments, logics and
    ↪ cognitive models" by Sujata Ghosh and Rineke Verbrugge in
    ↪ Synthese (in press)
//Strategy: Myopic_11:[root /\ <c>(U(p) = 2) /\ <d><e>(U(p) = 1) /\
    ↪ (1<=2) |--> c](p)

define task Myopic_11 {
        initial-goals: (root-one)
        goals: (playdir playother utility-one utility-two comparison-
            ↪ one)
        task-constants: (bigger not c p current notcurrent decision-
            ↪ node leaf root comparison right down)
        imaginal-autoclear: nil
        default-activation: 1.0
        rt: -2.0
        ol: t
        batch-trace: t
        default-operator-assoc: 8.0
        default-operator-self-assoc: -2.0
}
```

```
define goal root-one {

        operator root-one-check-yes {
                WM1 = nil
                V4 = root
        ==>
                nil -> WM1
                utility-one -> G1
        }

        operator root-one-check-no {
                WM1 = nil
                V4 <> root
        ==>
                nil -> WM1
                playother -> G1
        }
}

define goal utility-one {

        operator utility-one-mov-one {
                WM1 = nil
        ==>
                one -> WM1
                focus-down-last -> AC1
        }

        operator utility-one-check-yes {
                WM1 = one
                V3 = two
        ==>
                nil -> WM1
                V3 -> WM2
                utility-two -> G1
        }

        operator utility-one-check-no {
                WM1 = one
                V3 <> two
        ==>
                nil -> WM1
                playother -> G1
        }
}

define goal utility-two {

        operator utility-two-mov-one {
                WM1 = nil
        ==>
                one -> WM1
                focus-up-stay -> AC1
        }
```

```
        operator utility-two-mov-two {
                WM1 = one
        ==>
                two -> WM1
                focus-down -> AC1
        }

        operator utility-two-mov-three {
                WM1 = two
        ==>
                three -> WM1
                focus-down-last -> AC1
        }

        operator utility-two-check-yes {
                WM1 = three
                V3 = one
        ==>
                nil -> WM1
                V3 -> WM3
                comparison-one -> G1
        }

        operator utility-two-check-no {
                WM1 = three
                V3 <> one
        ==>
                nil -> WM1
                playother -> G1
        }
}

define goal comparison-one {

        operator comparison-one-start-retrieval {
                WM1 = nil
                RT1 = nil
                WM3 <> nil
                WM2 <> nil
        ==>
                WM3 -> RT1
                WM2 -> RT3
                comparison -> RT4
        }

        operator comparison-one-not-bigger {
                RT2 = not
                RT4 = comparison
        ==>
                playdir -> G1
        }

        operator comparison-one-bigger {
                RT2 = bigger
                RT4 = comparison
```

```
        ==>
                playother -> G1
        }
}



define goal playdir {

        operator play {
                G1 = playdir
        ==>
                play -> AC1
                down -> AC2
        }

}

define goal playother {

        operator play-right {
                G1 = playother
        ==>
                play -> AC1
                right -> AC2
        }

        operator play-down {
                G1 = playother
        ==>
                play -> AC1
                down -> AC2
        }

}

define init-script {
        add-dm("strat-fact-1","strat","efr","c","down")
        add-dm("strat-fact-2","strat","efr","p","down","right")
        add-dm("strat-fact-3","strat","efr","p","down","down")
        add-dm("strat-fact-4","strat","bi","c","down")
        add-dm("strat-fact-5","strat","bi","p","down","right")
        add-dm("strat-fact-6","strat","bi","p","down","down")
        add-dm("strat-fact-7","strat","op","c","right")
        add-dm("strat-fact-8","strat","op","p","right","down")

        add-dm("notlarger-one-one","one","not","one","comparison")
        add-dm("notlarger-one-two","one","not","two","comparison")
        add-dm("notlarger-one-three","one","not","three","comparison")
        add-dm("notlarger-one-four","one","not","four","comparison")
        add-dm("notlarger-one-six","one","not","six","comparison")
        add-dm("larger-two-one","two","bigger","one","comparison")
        add-dm("notlarger-two-two","two","not","two","comparison")
        add-dm("notlarger-two-three","two","not","three","comparison")
        add-dm("notlarger-two-four","two","not","four","comparison")
```

```
        add-dm("notlarger-two-six","two","not","six","comparison")
        add-dm("larger-three-one","three","bigger","one","comparison")
        add-dm("larger-three-two","three","bigger","two","comparison")
        add-dm("notlarger-three-three","three","not","three","
            ↪ comparison")
        add-dm("notlarger-three-four","three","not","four","comparison
            ↪ ")
        add-dm("notlarger-three-six","three","not","six","comparison")
        add-dm("larger-four-one","four","bigger","one","comparison")
        add-dm("larger-four-two","four","bigger","two","comparison")
        add-dm("larger-four-three","four","bigger","three","comparison
            ↪ ")
        add-dm("notlarger-four-four","four","not","four","comparison")
        add-dm("notlarger-four-six","four","not","six","comparison")
        add-dm("larger-six-one","six","bigger","one","comparison")
        add-dm("larger-six-two","six","bigger","two","comparison")
        add-dm("larger-six-three","six","bigger","three","comparison")
        add-dm("larger-six-four","six","bigger","four","comparison")
        add-dm("notlarger-six-six","six","not","six","comparison")

}

define script {
        trial-start()
        screen(["decision-node","p","current","root",["decision-node","
            ↪ c","notcurrent",["decision-node","p","notcurrent",["leaf
            ↪ ","six","three"],["leaf","one","four"]],["leaf","three
            ↪ ","one"]],["leaf","one","two"]])
        run-until-action("play")
        issue-reward()
        trial-end()
}
```

E: BI and EFR formulae

The following is a document on BI and EFR formulae for a subset of centipede games, created by Sujata Ghosh (personal communication).

Backward induction formulas

July 31, 2017

Consider any game similar in structure to the games in Figure 1, the payoffs might differ. Let us assume that actions are part of the observables, that is, $\Sigma \subseteq P$. The semantics for the actions can be defined appropriately. Let $n_1, \ldots, n_4$ denote the four decision nodes of Game 1 of Figure 1, with C playing at n_1 and n_3, and P playing at the remaining two nodes n_2 and n_4. We have four belief operators for this game, namely two per player. We abbreviate some formulas that describe the payoff structure of the game:

$\alpha := \langle d \rangle\!\rangle \langle f \rangle\!\rangle \langle h \rangle((u_C = p_C) \wedge (u_P = p_P))$
(from the current node, a d move followed by an f move followed by an h move lead to the payoff (p_C, p_P))

$\beta := \langle d \rangle\!\rangle \langle f \rangle\!\rangle \langle g \rangle((u_C = q_C) \wedge (u_P = q_P))$
(from the current node, a d move followed by an f move followed by a g move lead to the payoff (q_C, q_P))

$\gamma := \langle d \rangle\!\rangle \langle e \rangle((u_C = r_C) \wedge (u_P = r_P))$
(from the current node, a d move followed by an e move lead to the payoff (r_C, r_P))

$\delta := \langle c \rangle((u_C = s_C) \wedge (u_P = s_P))$
(from the current node, a c move leads to the payoff (s_C, s_P))

$\chi := \langle b^- \rangle\!\rangle \langle a \rangle((u_C = t_C) \wedge (u_P = t_P))$
(the current node can be accessed from another node by a b move from where an a move leads to the payoff (t_C, t_P))

Now we can define the conjunction of these five descriptions:

$$\varphi := \alpha \wedge \beta \wedge \gamma \wedge \delta \wedge \chi$$

Let ψ_i denote the conjunction of all the order relations of the rational payoffs for player i ($\in \{P, C\}$) given in any game similar to the games in Figure 1. Evidently, for different games with different payoff structures both φ and ψ_i's will differ.

Backward induction reasoning at the node n_2 can be formulated as follows depending on φ and ψ_i's:

- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle e \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle\langle f \rangle g) \mapsto c]^P$
- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle e \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle\langle f \rangle h) \mapsto c]^P$
- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle e \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle\langle f \rangle g) \mapsto d]^P$
- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle e \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle\langle f \rangle h) \mapsto d]^P$
- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle f \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle\langle f \rangle g) \mapsto c]^P$
- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle f \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle\langle f \rangle h) \mapsto c]^P$
- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle f \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle\langle f \rangle g) \mapsto d]^P$
- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \langle b^- \rangle \mathbf{root} \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle f \wedge \mathbb{B}^{n_2,P}_{g1}\langle d \rangle\langle f \rangle h) \mapsto d]^P$

For a game similar in structure given in Figure 2, assuming the nodes to be n_1, n_2 and n_3, BI reasoning can be formulated in a similar way:

- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \mathbf{root} \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle e \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle\langle f \rangle g) \mapsto c]^P$
- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \mathbf{root} \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle e \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle\langle f \rangle h) \mapsto c]^P$
- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \mathbf{root} \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle e \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle\langle f \rangle g) \mapsto d]^P$
- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \mathbf{root} \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle e \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle\langle f \rangle h) \mapsto d]^P$
- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \mathbf{root} \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle f \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle\langle f \rangle g) \mapsto c]^P$
- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \mathbf{root} \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle f \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle\langle f \rangle h) \mapsto c]^P$
- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \mathbf{root} \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle f \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle\langle f \rangle g) \mapsto d]^P$
- $\eta^1_P : [(\varphi \wedge \psi_P \wedge \psi_C \wedge \mathbf{root} \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle f \wedge \mathbb{B}^{n_1,P}_{g1}\langle d \rangle\langle f \rangle h) \mapsto d]^P$

Note that φ and ψ_i's need to be changed according the structure and pay-offs in these games.

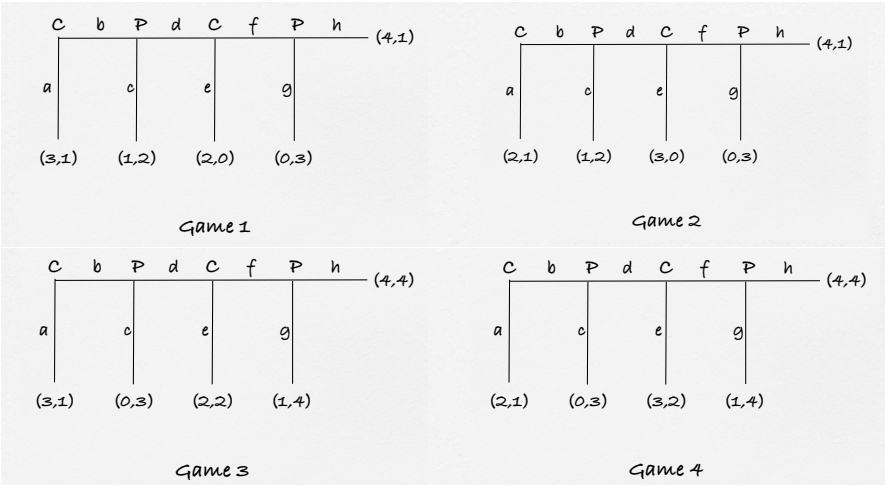

Figure 1: Main games

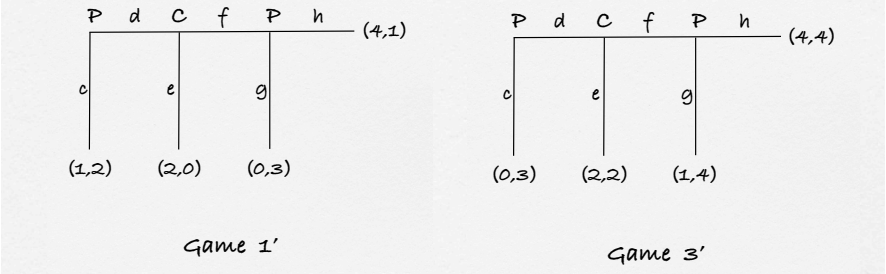

Figure 2: Truncated games

F: LaTeX symbol list

This section contains a list of uncommon LaTeX symbols used in this thesis and how to reproduce them in LaTeX, in case any reader wishes to do so. If the symbol is an accent or a modification of another symbol, we will use the letter a as the to-be-modified symbol. Code enclosed in dollar signs ($) only work in math mode.

Section of first appearance	Symbol	LaTeX code
1.2	a'	`$a'$`
2.1	Φ	`$\Phi$`
2.2	$\{$	`\{`
”	$\}$	`\}`
”	$\bar{\imath}$	`$\textit{\=\i}$`
”	$\overline{a}$	`$\overline{a}$`
”	Σ	`$\Sigma$`
”	ρ	`$\rho$`
”	λ	`$\lambda$`
”	$\emptyset$	`$\emptyset$`
”	$\mathbb{T}$	`$\mathbb{T}$`
”	$\Rightarrow$	`$\Rightarrow$`
”	$\times$	`$\times$`
”	$\rightarrow$	`$\rightarrow$`
”	$\in$	`$\in$`
”	$\vec{a}$	`$\vec{a}$`
”	$\mid$	`$\mid$`
”	$\overset{a}{\Rightarrow}$	`$\overset{a}{\Rightarrow}$`
”	$\varnothing$	`$\varnothing$`
”	$\widehat{\lambda}$	`$\widehat{\lambda}$`
”	μ	`$\mu$`
”	τ	`$\tau$`
”	$\neq$	`$\neq$`
”	$\restriction$	`$\restriction$` (see note 1)
”	Ω	`$\Omega$`
”	$\geqslant$	`$\geqslant$`
”	σ	`$\sigma$`
”	$\rightharpoonup$	`$\rightharpoonup$`
”	$\mathfrak{D}$	`$\mathfrak{D}$`
”	π	`$\pi$`
”	$\mathbb{G}$	`$\mathbb{G}$`
”	$\ldots$	`$\ldots$`
”	$\leqslant$	`$\leqslant$`
”	$\cup$	`$\cup$`
”	$\bigcup$	`$\bigcup$`
2.2.1	$\neg$	`$\neg$`
”	ψ	`$\psi$`
”	$\vee$	`$\vee$`
”	$\langle$	`$\langle$`
”	$\rangle$	`$\rangle$`
”	$\mathbb{B}$	`$\mathbb{B}$`
”	$\mapsto$	`$\mapsto$`
”	$\cdot$	`$\cdot$`
”	$\longrightarrow$	`$\longrightarrow$`
”	$\mathcal{U}$	`$\mathcal{U}$`

”	$\mathbb{Q}$	`$\mathbb{Q}$`
”	$\models$	`$\models$`
”	η	`$\eta$`
”	$\llbracket$	`$\llbracket$`
”	$\rrbracket$	`$\rrbracket$`
”	Υ	`$\Upsilon$`
”	$\cap$	`$\cap$`
2.2.2	α	`$\alpha$`
”	β	`$\beta$`
”	γ	`$\gamma$`
”	δ	`$\delta$`
”	χ	`$\chi$`
”	φ	`$\varphi$`
”	$\mathcal{K}$	`$\mathcal{K}$`
”	$\mathcal{X}$	`$\mathcal{X}$`

Table 5.1: LaTeX symbols used in this thesis

1. The command *\restrict* has been created for Ghosh & Verbrugge (online first) using the following code:

```
\newcommand{\restrict}{\:\raisebox{0.2ex}{$|$}\kern-1.8pt\raisebox{-.1ex}{$\
    ↪ scriptscriptstyle /$}\:}
```

Bibliography

Anderson, J. R. (2007). *How can the human mind occur in the physical universe?* New York: Oxford University Press.

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological Review*, *111*(4), 1036–1060.

Anderson, J. R. & Schooler, L. J. (1991). Reflections of the environment in memory. *Psychological Science*, *2*(6).

Arslan, B., Verbrugge, R., & Taatgen, N. (in press). Cognitive control explains the mutual transfer between dimensional change card sorting and first-order false belief understanding: A computational modeling study on transfer of skills. *Biologically Inspired Cognitive Architectures*, -.

Arslan, B., Wierda, S., Taatgen, N., & Verbrugge, R. (2015). The role of simple and complex working memory strategies in the development of first-order false belief reasoning: A computational model of transfer of skills. In *Proceedings of the 13th International Conference on Cognitive Modeling*, (pp. 100–105).

Gall, D. & Frühwirth, T. (2015). A refined operational semantics for ACT-R: Investigating the relations between different ACT-R formalizations. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, (pp. 114–124)., New York, NY, USA. ACM.

Gall, D. & Frühwirth, T. (2016). *Translation of Cognitive Models from ACT-R to Constraint Handling Rules*, (pp. 223–237). Cham: Springer International Publishing.

Ghosh, S., Heifetz, A., & Verbrugge, L. (2015). Do players reason by forward induction in dynamic perfect information games? In Ramanujam, R. (Ed.), *Proceedings of the 15th Conference on Theoretical Aspects of Rationality and Knowledge (TARK XV)*, (pp. 121–130)., Pittsburgh.

Ghosh, S., Heifetz, A., Verbrugge, R., & de Weerd, H. (2017). What drives people's choices in turn-taking games, if not game-theoretic rationality? In Lang, J. (Ed.), *Proceedings Sixteenth Conference on Theoretical Aspects of Rationality and Knowledge, Liverpool, UK, 24-26 July 2017*, volume 251 of *Electronic Proceedings in Theoretical Computer Science*, (pp. 265–284). Open Publishing Association.

Ghosh, S., Meijering, B., & Verbrugge, R. (2014). Strategic reasoning: Building cognitive models from logical formulas. *Journal of Logic, Language and Information*, *23*(1), 1–29.

Ghosh, S. & Verbrugge, R. (online first). Studying strategies and types of players: Experiments, logics and cognitive models. *Synthese*, -.

Gradwohl, R. & Heifetz, A. (2011). Rationality and equilibrium in perfect-information games. In *Working paper*, Northwestern university.

Horn, A. (1951). On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, *16*(1), 14–21.

Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In J. M. Claude E Shannon (Ed.), *Automata Studies* (pp. 3–42). Princeton University Press.

Meijering, B., van Rijn, H., Taatgen, N. A., & Verbrugge, L. C. (2012). What eye movements can tell about theory of mind in a strategic game. *PLoS ONE*, *7*(9), e45961.

Payne, J. W., Bettman, J. R., & Johnson, E. J. (1993). *The adaptive decision maker*. Cambridge University Press.

Savitch, W. (2012). *Absolute Java* (5 ed.). Pearson.

Stevens, C., Taatgen, N., & Cnossen, F. (2015). Agent models of the game of nines. *The METALOGUE Consortium*, *1*.

Taatgen, N., Katidioti, I., Borst, J., & van Vugt, M. (2015). A model of distraction using new architectural mechanisms to manage multiple goals. In Taatgen, N., van Vugt, M., Borst, J., & Mehlhorn, K. (Eds.), *Proceedings of the 13th International Conference on Cognitive Modeling*, (pp. 264–269)., Groningen, The Netherlands. University of Groningen.

Taatgen, N. A. (2013a). Diminishing return in transfer: A prim model of the frensch (1991) arithmetic experiment. In *Proceedings of the 12th international conference on cognitive modeling*, (pp. 29–34)., Ottawa. Cartelon University.

Taatgen, N. A. (2013b). The nature and transfer of cognitive skills. *Psychological Review*, *120*(3), 439–471.

Taatgen, N. A. (2016). Prims tutorial. Downloaded from http://www.ai.rug.nl/ niels/actransfer-/PRIMs.zip, accessed January 9th, 2017.

Wierda, S. & Arslan, B. (2014). Modeling theory of mind in actransfer. In *Proceedings of the Second Workshop Reasoning About Other Minds: Logical and Cognitive Perspectives*, Groningen, Netherlands.