Learning by Imitation using Inductive Logic Programming

by

Alan Chi-Lun Wai

A thesis submitted to the Faculty of Graduate Studies and Postdoctoral Affairs
in partial fulfilment of the requirements for the degree of
Master of Applied Science in Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering (OCIECE)

Department of System and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6
January 2011

Your file Votre référence
ISBN: 978-0-494-79558-3
Our file Notre référence
ISBN: 978-0-494-79558-3

Abstract

The transfer of knowledge from an expert to a software agent can be a tedious task. Instead, the agent can learn by observing the expert. This thesis examines, with RoboCup soccer data, how inductive logic programming can be used to learn rules in first-order logic that describe an agent's underlying behavior. Experimental results have shown that a discriminatory induction algorithm generates rules with better quality than those generated by a descriptive induction algorithm. We also show that the rules learned can be directly applied in a real-time environment with excellent response time. We conclude the thesis by comparing the results with previous CBR research work, and provide suggestions for future work.

Acknowledgments

First, I would like to thank Professor Babak Esfandiari for his on-going support and guidance through every stage of this thesis. His excellent insight and feedback has made this challenging journey enjoyable and fruitful. I would also like to thank Michael Floyd for providing various assistance.

Lastly, I would like to thank my Lord and Saviour Jesus Christ, for providing comforts during my difficult moments, opening doors when the road was dark, and helping me to finally make this thesis a success.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Designing and implementing software agents is often a difficult task that requires a significant amount of development time and software programming expertise [LET06]. In addition to the technical skills required to develop a software agent there is also the need to transfer domain knowledge to the agent, often times from a human expert. This requires modeling the expert knowledge in a manner that is interpretable by a software agent. Even after a software agent has been created, it is likely only able to perform a specific set of tasks in a given domain. Adding the ability to perform new tasks or deploying the agent in novel domains may require providing the agent with additional or modified expert knowledge.

In order to remove the technical skills required to model expert knowledge and train a software agent, and thereby making the software agent useful to people without such technical skills, the ability for an agent to learn through observation could be employed. More specifically, the agent could observe an expert (a human or another agent) perform a task and then attempt to imitate the behavior of the expert when faced with a similar task. This type of approach moves the burden of modeling the expert knowledge from the expert to the agent.

1.1 Motivation

After observing how a teacher behaves in specific situations, the agent can attempt to behave similarly when presented with similar situations. The imitating agent may not have any knowledge of the reasoning process used by the teacher and may only be able to observe the action the teacher performs, a, in response to sensory inputs, $x_1, \ldots, x_n$. The teacher could then be modeled by a function as follows:

$$a = f(x_1, \ldots, x_n)$$

By remembering the actions the expert performed given the sensory input, the agent could then perform similar actions when encountering similar sensory input. This type of reasoning, by solving current problems (sensory input) using knowledge from previous problem instances, is known as case-based reasoning (CBR) [AP94]. This type of reasoning has already been explored [LE05,FEL08,FE08] within the RoboCup domain. CBR exploits the assumption that similar problems have similar solutions, so it lends itself well to imitation where an agent and teacher should behave similarly given similar sensory input. Several limitations/difficulties with CBR were encountered as follows:

- In RoboCup, a case needs to describe multiple and diverse objects, as well as relationships between them. Coming up with a metric to calculate the distance between such descriptions can be very difficult.
- CBR is unable to encode useful background knowledge as part of its learning. Such background knowledge can include spatial properties of the objects, laws of physics, etc.
- The reasoning ability of CBR is related to the number of cases that can be remembered. The system must contain enough cases so that a wide variety

of input problems can be solved As it was found in [FE08], the number of stored cases can be quite big (tens of thousands), which can be problematic if such approach is implemented in a real-life robot where resources are scarce.

- It is not very easy for a human to understand the underlying behavior of the imitated agent simply by looking at the cases themselves.

In light of these limitations with CBR, we are motivated to investigate other approaches for retrieving the appropriate action given the sensory input. We have decided to focus on inductive learning techniques, and Inductive Logic Programming (ILP) in particular. ILP is the combination of inductive machine learning and logic programming, and can be considered a classification technique. The main goal of ILP is to provide tools and techniques to induce hypotheses from observations and to synthesize new knowledge from experience [MLR94]. Furthermore, ILP uses first-order logic as knowledge representation, which is more expressive than propositional logic used by classical machine learning techniques such as the Top-Down-Induction-of-Decision-Tree (TDIDT) family [Qui86]. First-order logic (FOL), or predicate calculus, provides access to the components of an individual assertion within a proposition declared in propositional logic [Lug01]. It allows expressions to contain variables, which allows ones to create general assertions about classes of entities rather than using specific instances as in propositional logic. In addition, a scene in a spatial domain inherently consists of relationships and interactions between objects on the field, not simply attributes of individual objects. FOL is an appropriate language for describing relationships, unlike attribute-based learning algorithms. The clauses generated with ILP algorithms are in first-order logic themselves, making them very human-readable. We summarize the benefits of ILP below:

- Expressive representation using first-order logic

- Allows relational predicates learning
- Use of variables to create general assertions
- Background knowledge integrated in the search process
- Rules generated provide high generalization over examples, allowing better efficiency for real-time performance
- Rules generated are human-readable

Background knowledge allows miscellaneous domain-specific information and tasks such as data pre-processing, defining relevant attributes/objects to use, or confining the search space, to be included in the ILP algorithms as part of the search process. This eliminates many pre-processing steps that are usually required (as in CBR) to convert raw data into searchable data, or parameters/switches that are hard-coded in the algorithm itself. More importantly, background knowledge allows one to explicitly encode some of the underlying relations between predicates involved in the rules to be learned. For example, symmetric and transitive properties of directional relations (ie. left/right/front/back) can be encoded to indicate spatial facts, such as if object A is to the left of object B, then object B is to the right of object A. Such information simplifies learning by reducing the rules needed to be learned, and potentially allows new knowledge to be inferred.

1.2 Objectives

The goal of this research is to determine whether ILP can be used to imitate the behavior of RoboCup agents and overcome the limitations of CBR presented above. Note that it is not our goal to improve on the performance of existing RoboCup champions, since the notion of improvement would presuppose that our agent

would have a hard-coded notion of goal or utility. In imitation, the agent does not know whether it is successful at the task it is undertaking. The advantage of imitation, however, is that the end user is not expected to have any expertise in coding the task or utility, and can also re-program the agent by demonstration for any other task if she wishes. This leads to the following research questions (in the context of the RoboCup Simulation League):

1. Can ILP be used to effectively learn the behavior of RoboCup agents?
    - Which ILP algorithms are most suitable for this type of learning?
    - Does the use of background knowledge improve learning?
2. Can the ILP approach perform more efficiently than the CBR approach in real-time?

There are many ILP algorithms available, as we explain in Section 2. This thesis will limit itself to two families, namely discriminatory and descriptive induction.

1.3 Contribution

The primary contributions of this thesis in attempting to answer the research questions posed in the previous section are as follows:

- Comparison of two families of ILP algorithms in the RoboCup simulated soccer domain.
    1. Evaluation and Comparison of different search methods using the ALEPH software (discriminative ILP);
    2. Evaluation of the CLASSIC'CL software, and implementation of various improvements (characteristic ILP);

- We found that the discriminative ILP family provides better results than the characteristic ILP family;
- The use of background knowledge does not provide significant performance improvement for simple agents such as Krislet and Sprinter, but shows improvement with a modified Krislet (Krislet2) where intrinsic distance between objects has a role in behavior;
- The ILP approach can provide excellent real-time performance;
- We also contributed an open-source framework that integrates a Prolog inference engine with a RoboCup agent.

The ILP approach is shown experimentally to provide promising results comparable to that of CBR. While it provides excellent real-time performance, it suffers from limitations due to region discretizations. Also, similar to our CBR framework, the ILP imitation system currently is not able to identify multiple states of behavior and only considers visual stimuli. Therefore the imitative agents still have difficulty imitating multi-state agents or agents that rely heavily on inter-agent communication.

1.4 Organization

Following the introduction, Chapter 2 provides an overview of inductive logic programming and describes the RoboCup domain. Chapter 3 provides an overview of the current state of research in imitative learning, learning techniques with ILP, and spatial ontology. Chapter 4 outlines the methodological approach followed in this thesis, as well as the proposed unit of analysis, method of data collection and method of data analysis.

The design and implementation of our ILP imitation system is presented in Chapter 5. Chapter 6.2 and 6.3 presents the experimental results and analysis for two ILP algorithms of different families. Chapter 6.4 describes how the agents behave in real time using the FOL rules generated, and chapter 6.5 compares the results qualitatively and quantitatively to that of CBR. Finally, Chapter 7 provides conclusions and outlines future areas of work.

Chapter 2

Background

The following sections will provide a summary of two important topics relevant to this thesis: the RoboCup Simulation League and Inductive Logic Programming (ILP). We assume readers have a general understanding of first-order logic (FOL).

2.1 RoboCup Simulation League

RoboCup [Rob08] is a collection of competitions that aim to foster artificial intelligence and robotics research. The project initially started as a robotic soccer competition but has since expanded to include disaster recovery (RoboCup Rescue) and human-machine interaction (RoboCup@Home) competitions. RoboCup has become an increasingly popular platform for the application and testing of artificial intelligence, multi-agent systems and robotics research and is the focus of numerous conferences and workshops every year. Various algorithms and techniques can be compared using one of the RoboCup competitions as a common platform.

The RoboCup soccer competition itself is further divided based on the type of robots used. There are divisions for small-size, medium-size, four-legged and humanoid robots as well as a simulation division called the RoboCup Simulation

League, which provides an excellent virtual platform for research without building physical robots. The League is based on a client-server architecture with each player represented as a separate client. All players connect to the soccer server [CFH+02] which maintains the state of the game (score, time, and location of objects), enforces the rules of the game and handles communication with each player. At discrete time intervals, the server sends a message to each player informing them of their current sensory inputs (Figure 1) and the players respond with the action they wish to perform (Table 1). In this thesis, only the first three actions (ie. Kick, Dash, Turn) will be used. A graphical view of a simulated game can be seen in Figure 2.

```
      Time = simulation cycle of the soccerserver
   ObjInfo = (ObjName Distance Direction DistChange DirChange BodyFacingDir HeadFacingDir )
             (ObjName Distance Direction DistChange DirChange
           | (ObjName Distance Direction)
           | (ObjName Direction)
   ObjName = (p [ Teamname | Uniform Number [goalie]])
             (b)
             (g [l|r])
             (f c)
             (f [l|c|r] [t|b])
             (f p [l|r] [t|c|b])
             (f g [l|r] [t|b])
             (f [l|r|t|b] 0)
           | (f [t|b] [l|r] (10|20|30 40|50])
             (f [l|r] [t|b] 10|20|30})
           | (l [l|r t|b])
             (B)
             (F)
             (G)
           | (P)
   Distance = positive real number
  Direction = -180 ~180 degrees
 DistChange = real number
  DirChange = real number
HeadFaceDir = -180 ~180 degrees
BodyFaceDir = -180 ~180 degrees
   Teamname = string
Uniform Number = 1 ~11
p/P: player; b/B: ball; g/G: goal; f/F: flag; l: line
```

Figure 1: RoboCup Visual Sensor Protocol [CFH+02]

RoboCup Simulation League provides a dynamic, non-deterministic environment where players have an incomplete world model. Other players (and the ball) are able to move and can influence the environment, so there is no guarantee that

Object	Parameters	Notes
Kick	power (-100 to 100), direction (-90 to 90)	once per cycle
Dash	power (-100 to 100)	once per cycle
Turn	moment (-180 to 180)	once per cycle
Catch	direction (-90 to 90)	once per cycle, only by goalie
Turn Neck	angle (-90 to 90)	can be performed in the same cycle as other actions
Move	X (-54 to 54) and Y (-32 to 32) position	only performed before kickoffs

Table 1: RoboCup soccer simulation actions and their parameters

a player's desired action will succeed. In addition, players only have a limited view of the soccer field at a given time, and they cannot be certain of the position/direction of objects (due to noise in the position data provided by the server, or if they are very far). These factors create interesting challenges that must be taken into account when designing how the players will reason and behave

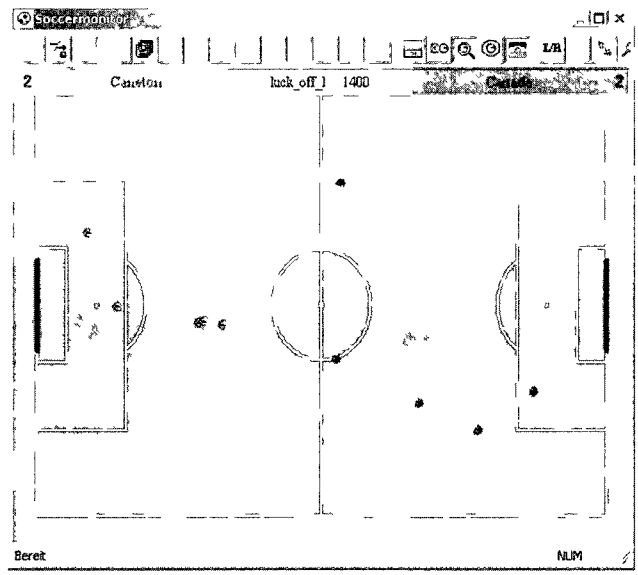

Figure 2: A visualization of a RoboCup Simulation League game

2.2 Inductive Logic Programming (ILP)

2.2.1 Introduction

Inductive Logic Programming, or ILP, is the combination of inductive machine learning and logic programming. In Section 1.1, we described some benefits of ILP, including expressive representation using clausal logic, extensive use of background knowledge, multi-relational learning, etc. Before we can formally define inductive logic programming, we shall review some first-order logic (FOL) concepts below. Readers can refer to [McA92] for more details on FOL.

- First-order alphabet: consists of variables, predicate symbols and function symbols (including constants)
- Term: a variable is a term, and a function symbol immediately followed by a bracketed n-tuple of terms is a term.
- Atom: a predicate symbol immediately followed by a bracketed n-tuple of terms.
- Literals: if L is an atom, both L and its negation ¬L are literals.
- Clause: a formula of the form $A_1,\ldots,A_m \leftarrow B_1,\ldots,B_n$, where A_i and B_i are positive literals. It can be read as "If B_1 and ... and B_n, then A_1 or ... or A_m". All variables in the clause are universally quantified. The left side of "$\leftarrow$" is the head of the clause; the right side is the body.
- Clause theory: a conjunction of clauses
- Horn clause: A clause where m = 0 or 1 (ie. only 0 or 1 literal in the head)
- Definite clause: A clause where m = 1 (ie. exactly 1 literal in the head)

- Fact: a definite clause with an empty body.
- Ground fact: a fact with all variables instantiated as constants.
- Example: a ground fact with its truth value in the intended interpretation: positive examples are true; negative examples are false
- Query: viewed as a horn clause having conjunction of atoms in the body, but with no head. It is usually written in the form:

  $$?\text{-}\ A_1, \ldots, A_n, \text{ where } A_i \text{ are atoms}$$

  A query is considered the first-order logic equivalent of an itemset [DT98] (See Section 2.2.5).

According to [RL93], ILP can be defined as follows. Given the input:

1. a set of *examples* E
2. *background knowledge* B
3. a *language bias* LB that defines the hypothesis search space
4. a notion of *explanation* (a semantics)

Find a *hypothesis* H (theory) within the constraints of the language bias LB that explains the examples E with respect to the background theory B.

Positive and negative examples are denoted as E_+ and E_- respectively, and $E = E_+ \bigcup E_-$. The background theory B and hypothesis H are represented by sets of clauses The entire set of permitted clauses that may be included in hypothesis H is called the hypothesis search space, since H is hidden somewhere in the list of such clauses. In other words, learning is equivalent to searching for a correct theory. The language bias LB places constraints on the clauses allowed in the search space,

such as restriction to Horn clauses only, clauses without function symbols, number of literals allowed, number of clauses allowed in the theory, etc. Obviously, there is a trade-off between the size of the search space and the quality of the hypothesis H being discovered.

Searching of the hypothesis space can be done through classical search methods such as depth-first, breadth-first, best-first search, beam search, etc. Through the use of a refinement operator (Section 2.2.3), the search can be done from the most general clauses in the hypothesis space to the most specific ones (general-to-specific), or from the most specific clauses to the most general (specific-to-general), as allowed by the language bias. Generality order is used to structure a set of clauses, to define what it means for a clause to be more general than another. The most often used generality order in ILP is the Subsumption Order (Section 2.2.2), which imposes a partial order on the set of clauses. The different notions of explanations and semantics derive from the two main settings of ILP, which are listed in Table 2 and detailed in Section 2.2.5.

Setting	Also Known As	Descriptions
Normal	Discriminatory induction, strong/predictive ILP	Focus on separating positive and negative observations, learning of classification and prediction rules
Non-monotonic	characteristic induction, weak/descriptive ILP	Focus on characterizing (finding the explanations of) one or more observations, learning of clausal theories

Table 2: Major Settings for ILP

There are other ILP settings that aims to integrate the two main ILP settings [DD97]. In this thesis, we shall only focus on the normal and non-monotonic settings.

2.2.2 Subsumption Order

As mentioned earlier, ILP is basically a search problem. There is a space of candidate solutions to be searched (ie. the set of hypotheses defined by the language bias) based on an acceptance criterion (eg. consistency, validity). One can solve ILP as an enumeration algorithm by performing a naive generate and test. However, as with any other search problems, this kind of algorithm is too computationally expensive. In ILP, the search space is typically structured by means of dual notions of specialization and generalization. Finding a correct theory means to adjust the theory repeatedly by means of specialization and generalization to fit the examples through the use of refinement operators. θ-subsumption is a weak form of generalization that is decidable, hence it is most often employed within the ILP paradigm. The θ-subsumption order on clauses is defined as follows:

Definition [NCW97]: Let c_1 and c_2 be clauses. A clause c_1 *θ-subsumes* c_2, denoted as $c_1 \geq c_2$, if and only if there exists a substitution θ such that $c_1\ \theta \subseteq c_2$ (ie. every literal in $c_1\ \theta$, whose variables in c_1 are replaced by constants in θ, is also a literal in c_2). Clause c_1 is at least as general as clause c_2 under θ-subsumption.

For example, let c_1 be the clause:

$$c_1 = P(X, Y) \leftarrow Q(X)$$

Let c_2 be the clause:

$$c_2 = P(X, Y) \leftarrow Q(a)$$

Then c_1 θ-subsumes c_2 under the substitution θ = X/a

Determining whether a clause subsumes another is decidable (ie. can be calculated in finite time) The subsumption relation $\geq$ on the set of clauses G is reflexive and transitive. Thus the pair $< G, \geq >$ is called a partial-ordered set Two clauses

c and d are *θ-subsumption equivalent* if c θ-subsumes d and d θ-subsumes c. A clause is *reduced* if it is not θ-subsumption equivalent to any proper subset of itself. $< G, \geq >$ forms a lattice on the set of reduced clauses, which structures the hypothesis space by providing a generality ordering for the hypotheses. $< G, \geq >$ is called a lattice because for every two clauses c and d in G, a least upper bound (ie. least general generalization) and a greatest lower bound (greatest specialization) exists, under the subsumption relation. The same is also true if G is a set of Horn clauses. Such lattice structure has two important properties for pruning the search space:

- When generalizing c_2 to c_1, all examples covered by c_2 will also be covered by any of its generalizations as well (eg. c_1). This is because if $B \bigcup c_2 \models e$ is true, then $B \bigcup c_1 \models e$ is also true, where e is an example.

- When specializing c_1 to c_2, an example not covered by c_1 will not be covered by any of its specializations either (eg. c_2). This is because if $B \bigcup c_1 \not\models e$ is true, then $B \bigcup c_2 \not\models e$ is also true, where e is an example.

2.2.3 Refinement Operator

Now that we know how the hypothesis space is structured, we need to know how the lattice structure is being traversed.

A bottom-up search (specific-to-general) of the hypothesis space can be performed by starting from the most specific clauses (as constrained by the language bias) and repeatedly applying generalization to the clauses until they start to cover negative examples, while ensuring they cover at least one positive example. When the clause starts to cover negative example(s), the first lattice property above allows all the generalizations to be pruned, since all of them will be inconsistent as well. The generalization step is known as the upward refinement operator, which computes only the least general generalizations of a clause.

A top-down search (general-to-specific) of the hypothesis space can be performed by starting from the most general clauses and repeatedly applying specialization to the clauses until they no longer cover negative examples, while covering at least one positive example. If a clause does not cover a positive example, the second property allows all the specializations of the clause be pruned, since none of them will cover a positive example as well. This generalization step is known as the downward refinement operator, which computes only the most general specializations of a clause. For efficiency reasons, heuristics are very often used to prune parts of the search space, which means ideal theories may never be found.

2.2.4 Language Bias

Almost all ILP algorithms require the user to specify a language bias, which guides the algorithm only to search for patterns interesting to the user that are within an otherwise huge, and often infinite, search space. The bias restricts what predicates to be searched, types of variables allowed (eg. player, ball, flag, object, etc.), and *modes* of the predicate parameters. A popular language specification called WRMODE has the form of *rmode*(*n*: *Atom*), where:

- n = *maximum* # of occurrences *Atom* that can appear in a candidate clause
- *Atom* is in the form of $pred(arg_1, \quad , arg_m)$
- arg_i = {+,-,#}*type* (see below)

Each argument arg_i has a type, which can be in one of these modes:

- Input (+) the argument is strictly an input variable, meaning the variable exists in a previous atom
- Output (-) the argument is strictly an output variable, meaning the variable cannot appear before this atom
- Constant (#): the argument is a constant
- In/Out (+-). some ILP algorithms also allow the argument to be both an input and output variable

```
rmode(1,key(-int))
rmode(1,action_turn(+int#dir)).
rmode(2,objectSeen(+int,-obj))
rmode(2,objectSeen(+int,#obj))
rmode(1,not_objectSeen(+int#obj))
rmode(2,dirQual(+int,+obj,#dir)).
```

Figure 3: WRMODE language bias example

```
not_objectSeen(A,B) ← key(A), not(objectSeen(A,B)).

dirQual(C,self,X,farleft) ← key(C), objectSeen(C,X), dirQuan(C,self,X,D), D ≤ -23.0
dirQual(C,self,X,farright) ← key(C), objectSeen(C,X), dirQuan(C,self,X,D), D ≥ 23.0 .
```

Figure 4: Background Knowledge example

Figure 3 gives an example of the WRMODE language bias, with background knowledge in Figure 4 that needs to go along with it. The search space lattice grows exponentially when number of predicates, arguments, and instantiable objects increase. It is therefore crucial to specify the language bias wisely to keep the search space manageable, but not to a point where accuracy of the model is significantly sacrificed.

The first argument of each predicate is a variable that takes an integer as an input, which indexes each predicate across different scenes. Algorithms such as WARMR use the *key* predicate to allow filtering of the database into smaller set

of relevant examples, so that only those entries in the database are scanned for learning. In this language bias, we are only including the *turn* action for learning, which carries the *direction* parameter with it. Since the parameter's mode is "#", the argument will be instantiated as a constant with either *farleft* or *farright* through the conversion using the *dirQual*/4 predicate definitions in the background knowledge. This is one of the benefits of using background knowledge.

The two separate declarations for the predicate *objectSeen* demonstrates that multiple modes can be defined for a single predicate. The first definition with "-obj" can instantiate the object parameter using variable only, while the second definition with "#obj" must instantiate it with a constant. The constant to be instantiated during a search iteration is chosen among all the ground facts found in the knowledge base (ie. *p*1, *gr*, *ball*1, *p*2). The predicate *not_objectSeen* is defined in the background knowledge from *objectSeen* using negation as failure. This means that for any scene where *objectSeen*(A,B) does not succeed (ie. not true), *not_objectSeen*(A,B) succeeds.

In this example, the predicates *key*, *action_turn* and *not_objectSeen* can only appear at most once in a generated query (n = 1), while *objectSeen* (both forms) and *dirQual* can appear in a query up to two times (n = 2).

A query is mode conform if an ordering of atoms exists such that every input variable occurs in one of the previous atoms within the query, and no output variable does. A query is type conform if arguments that share a variable name have identical types, or at least one of them is untyped. The following queries are mode and type conform:

```
?- key(A), objectSeen(A,B), dirQual(A,B,farleft)

?- key(A), action_turn(A,farright), objectSeen(A,B), objectSeen(A,C), objectSeen(A,gr)
```

The following queries are *not* mode and type conform:

```
?- key(A), dirQual(A,B,farleft), objectSeen(A,C)

?- key(A), objectSeen(A,B), dirQual(A,A,farright)
```

The first query is not mode conform because the second argument (B) of *dirQual* is an input variable, and it does not appear in any previous atoms. The second query is not type conform because the second argument of *dirQual* is of an *obj* type, but it takes in an *int* instead (type of the argument in *key*).

2.2.5 Normal and Non-monotonic settings

Introduction

The normal setting of ILP restricts background theory B and hypotheses H to sets of definite clauses, and employs the normal semantics: $B \bigcup H \models E_+$ (*completeness*) and $B \bigcup H \not\models E_-$ (*consistency*). It states that for all positive examples e in E_+, hypothesis H is *complete* iff the union of the background knowledge B and the hypothesis H logically entails e (ie. $B \bigcup H$ is more general than e). And for all negative examples e in E_-, hypothesis H is *consistent* iff the union of background knowledge B and hypothesis H does not logically entail e. The aim is to find H that is both *complete* and *consistent* with respect to E_+ and E_-. Then we say H is *correct*. In the non-monotonic setting, for all positive examples E, all clauses c in H must respect the semantics: $B \bigcup E \models H$ (ie. $B \bigcup E$ is more general than H). All negative examples are derived implicitly through the closed world assumption (ie. all examples not stated are considered false).

In the normal setting, hypothesis clauses are found to cover (entail) all positive examples (while not implying any negative examples), but it is possible that they also cover other examples as well (that are not part of the negative examples). Therefore, we say that the normal setting can realize an *inductive leap* since new facts can be induced from the generated hypothesis. In this sense, the normal

setting identifies a hypothesis that can discriminate between positive and negative examples with a minimal set of hypothesis clauses (ie. *discriminatory induction*). In the non-monotonic setting, all clauses found in hypothesis H must be true in the combined theory B and examples given. Thus H is said to be deduced from the observed examples and theory. It does not induce new facts which cannot be identified as true in $B \bigcup E$. In this sense, the non-monotonic setting looks for all valid characteristics in a set of unclassified observations (ie. *characteristic induction*), which can be useful for reverse-engineering. Because of this, clauses derived in the non-monotonic setting are more certain and conservative than those derived in the normal setting. The hypotheses produced by characteristic induction are usually a superset of those produced by discriminatory induction [DD97]. The following table summarizes the differences between the two settings.

	Normal Setting	**Non-monotonic setting**
Induction characteristic	predictive/discriminatory	descriptive/characterization
Property of hypothesis	complete and consistent with data	looks for all interesting or valid regularities
Training Data	require positive and negative examples	can use unclassified (positive only) data
Generated Rules	usually definite clauses	can be any clausal forms
Application	concept-learning	data-mining
Example	Given B, E_+, E_- • *son*(X,Y) ← *male*(X), *parent*(Y,X)	Given B, E_+ only • ← *son*(X,Y), *father*(X,Y) • *male*(X) ← *son*(X,Y) • *father*(X,Y), *mother*(X,Y) ← *parent*(X,Y)

Table 3: Difference between normal and non-monotonic setting

Besides the two main settings, ILP algorithms can also be categorized as *supervised* vs. *un − supervised* (ie. whether examples need to be pre-classified as positive and negative). Table 4 lists a few famous ILP systems.

ILP System	ILP Setting	Supervised/Unsupervised
Progol	Normal	Sup/Unsup
FOIL	Normal	Sup
TILDE	Normal	Sup
WARMR	Non-monotonic	Unsup
CLAUDIEN	Non-monotonic	Unsup

Table 4: Comparisons of a few classical ILP Systems

The next few sections describe these systems in more details, which show how they actually perform and allows us to choose the relevant system(s) to deploy.

ILP systems in Normal setting

For all illustrations presented in the following sections, readers should refer to the demonstration example listed in Appendix A It shows eight scenes (represented in FOL) of a Krislet [Lan] Robocup agent playing soccer. Krislet is a simple agent that finds the ball, runs toward it and kicks it to the goal. Section 4.2.1 describes the behavior of the Krislet agent in more details.

Progol Progol [Mug95, MF01] attempts to find the minimal amount of clauses that best explain pre-classified observations. It uses a covering algorithm which repeatedly finds clauses to cover existing examples, and then removes examples that have already been covered by the hypotheses found so far, until all given examples are covered. It calculates a compression value f_s for each candidate clause s that measures how well it explains the examples given. In general, Progol calculates f_s for each s as follows:

p_s = # of positive examples correctly covered by s

n_s = # of negative examples *incorrectly* covered by s

c_s = length of clause s - 1

$f_s = p_s - (n_s + c_s)$

The compression equation is quite self-explanatory: covering positive examples means the clause has good compression (ie. compressing many examples into a single clause), while n_s and c_s are regarded as the cost of the clause. Preference is given to shorter clauses that deduce very few or no negative examples.

Progol requires examples to be classified into positive and negative examples in order for it to properly compute the compression f_s. This implies that Progol can only learn one class at a time. For example, to learn the different types of actions in RoboCup, user will need to specify one particular action type as positive examples, and all other types as negative. The process is then repeated for each action type in order to learn all of them.

Progol can also operate in a mode where learning can be performed using positive data only (hence can also be considered as unsupervised learning). Since our RoboCup actions are readily classifiable based on the action type, we will not consider this mode of operation.

Progol applies an A*-like algorithm to heuristically find the clause with maximal compression, starting from the empty clause, to a goal node (ie. the most-specific clause) within the constraints of a language bias, which is shown in Figure 5 as an example.

modeh/modeb have a very similar semantic as *rmode* as described in Section 2.2.4, except that *modeh* specifies that the predicate *action_turn*/2 must appear only in the head of a clause, whereas *modeb* specifies that the corresponding predicates must appear only in the body. For each given positive example, a most-specific clause is generated by Progol, according to the language bias, to bound the bottom of the hypothesis lattice to improve search efficiency, unlike many other ILP algorithms

```
modeh(1,action_turn(+int,-dir))

modeb(2,objectSeen(+int,-obj))
modeb(2,objectSeen(+int,#obj))
modeb(1,not_objectSeen(+int,#obj))
modeb(1,dirQual(+int,+obj,-dir))
modeb(1,dirQual(+int,+obj,#dir))
```

Figure 5: Example of Language Bias in WRMODE for Progol

where the bottom of the lattice is unbound. As the name suggests, this clause is the most specific clause which can possibly be generated under the language bias that can cover the given example. Table 5 shows the most-specific clauses for discretized versions of 3 scenes given from Appendix A. In these cases, all the *turn* actions are classified as positive and the rest (*dash* and *kick*) are classified as negative. Progol generates the most specific clause using a process called mode-directed inverse entailment, and interested readers can refer to [Mug95] for more details. The FOL scene descriptions are repeated here for convenience. The most specific clause for scene 8 shows that a large amount of search space can be pruned out: none of the predicates except *not_objectSeen* can be used for refinements for this example, since any candidate clause must be bounded by the empty clause and the most-specific clause.

For each given example, Progol traverses the search space using the refinement operator, adding/changing a new literal one at a time within the language constraints, keeping clauses with the best compressions at each refinement level along the way. All generated refinements must subsume the most specific clause under θ-subsumption (ie. at least as general as the most specific clause). Using the lattice structure properties described in Section 2.2.2, Progol can calculate if none of the remaining candidate clauses nor their refinements will produce a better compression than the current best. When that happens, the best clause has been determined and the search can terminate. Then, all examples covered by this clause are eliminated.

Scene	Scene Description	Most-Specific Clause
1	objectSeen(1,p1) dirQual(1,self,p1,right). distQual(1,self,p1,veryfar) actionQuan(1,turn,farright).	action_turn(A,B) ← objectSeen(A,C), objectSeen(A,p1), not_objectSeen(A,ball1), dirQual(A,C,right).
3	objectSeen(3,p2). dirQuan(3,self,p2,left) distQuan(3,self,p2,close) actionQuan(3,turn,farright)	action_turn(A,B) ← objectSeen(A,C), objectSeen(A,p2), not_objectSeen(A,ball1), dirQual(A,C,left).
8	actionQuan(8,turn,farright)	action_turn(A,B) ← not_objectSeen(A,ball1).

Table 5: Most-Specific Clause Examples

This process repeats until all given examples have been covered.

Starting from scene #1 in Table 5, Progol will generate these candidate clauses:

1. *action_turn*(A,B) ← *objectSeen*(A,C)
2. *action_turn*(A,B) ← *objectSeen*(A,p1)
3. *action_turn*(A,B) ← *not_objectSeen*(A,ball1)
4. *action_turn*(A,B) ← ...

Based on all the examples listed in Appendix A, the compression value calculated for each of these clauses is 0, -3, and 2, respectively. When Progol determines that it will not obtain a better compression value than 2, it will pick the 3^{rd} clause as the best clause. Progol will then remove scene #1, #3, and #8 from the list of examples, since they are all covered by this newly generated clause. The process then repeats using the next available example (scene #2), until all examples are covered.

FOIL FOIL [QM90] is one of the earliest first-order logic inductive learners. It evolved from the attribute-value learning algorithm ID3 [Qui86], to build classification rules expressed in a restricted form of first-order logic, namely function-free Horn clauses. Given a set of examples that represent a target relation, FOIL finds first-order clauses that represent the general definition of the relation. Similar to Progol, FOIL operates on a set of positive and negative examples.

Like Progol, FOIL operates as a covering algorithm, and only finds minimally sufficient clauses that can distinguish positive examples from negative ones. However, unlike Progol which exhaustively searches for all subsets of clauses (refinements) bounded by the most specific clause, FOIL does not employ any bottom bound, nor does it search exhaustively. Instead, FOIL grows a clause by adding a literal one at a time in a greedy manner to the body of the clause. During each iteration, FOIL uses an information-based heuristics as in ID3 to find the best literal that gives the maximum gain, and moves on to the next iteration (ie. adding the next best literal) until the resulting clause does not cover any negative examples. Basically, a literal produces a positive gain if the concentration of positive examples that satisfy the new literal is higher compared to the concentration before the literal is added, and the literal with the highest gain is chosen. The gain heuristic is also useful for significant pruning of the search space. However, due to its greediness, FOIL suffers from local maximum problem, and cannot backtrack to generate clauses that could have produced better compression.

Unlike Progol, FOIL only allows variables in the predicates. This restricts the variations of rules that can be generated, limiting its usefulness. Another weakness is that FOIL does not support background knowledge where clauses are declared using arbitrary Prolog programs, although some variations of FOIL attempt to correct this [MC]. Thus, it is not possible, for example, to specify background clauses that map predicates like *actionQn* into *action_turn*, etc. This

implies that the training models need to be pre-processed separately beforehand for such conversions.

Using the same example as before, Table 6 shows how FOIL determines the best literal to pick at each iteration. Readers can refer to [QM90] or [Qui86] for details on calculating information gain, which is not the focus here. Assuming that step 2.2 produces the best gain, the literal *not_objectSeen(A,C)* will be chosen for the new clause (since it does not cover any negative tuples). All positive examples that satisfy this clause are eliminated and the process continues until all examples have been covered.

Step	Clause	p(c)	n(c)	Gain
1	action_turn(A,B)	5	3	-
2 1	action_turn(A,B) ← objectSeen(A,C)	4	3	-0 52
2 2	action_turn(A,B) ← not_objectSeen(A,C)	1	0	0 68
2 3	action_turn(A,B) ←			

Table 6: Choosing the best literal in FOIL

TILDE TILDE (Top-down Induction of Logical Decision Tree) [BDR98] is an algorithm that combines ILP with logical decision trees. Logical decision trees are a first-order logic upgrade of the classical decision trees used by propositional learners. TILDE builds a binary decision tree where each node of the tree represents a literal or conjunction of literals, also known as a test. Each node has two subtrees that represent two outcomes: the left side being the test has succeeded and the right side failed. A test succeeds if a query Q succeeds in $e \bigcup B$, where e is an example and B is background knowledge. Each leaf of the decision tree represents the resulting class of the path from root to the leaf. Since each test node is a conjunction (one or more) of literals, the path that leads to the node from root derives a clause. Hence the resulting tree is a logical decision tree, which can be used directly for classification of unseen examples. This means that the examples

must be pre-classified into their appropriate classes (eg. RoboCup actions) before training. Figure 6 shows how a logical decision tree generated by TILDE may look like using the example in Appendix A.

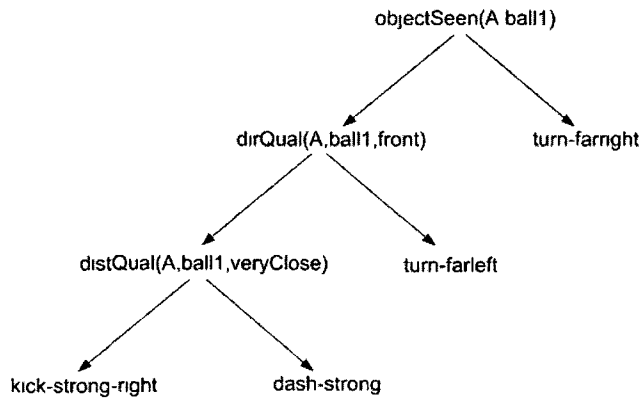

Figure 6: TILDE Example

Like other decision tree learners, TILDE employs a greedy divide-and-conquer strategy to build a binary induction tree by recursively testing and applying subdivision criteria on a training examples E, taking language bias and background knowledge into account. TILDE is similar to FOIL in the way that it uses a heuristic function, the gain ratio, to determine the literal that can best split the examples E into two partitions: the left side where the clause built along the path from the root so far succeeds, and the right side where it fails. It uses a refinement operator which again traverses the hypothesis lattice under θ-subsumption. TILDE uses the gain ratio computation employed in the popular C4.5 algorithm by Quinlan [Qui93]. The tree shown in figure 6 is generated by hand for illustration purposes only, and is not based on computation of gain ratios. However, it does classify the examples properly, although the order of the literals in the tree may be different if accurate gain ratios are computed. Readers are referred to [Qui93] and [BDR98] for details on gain ratio.

ILP systems in Non-monotonic setting

We shall first review the influential Apriori algorithm [AS94] developed by Agarwal and Srikant, which is the basis for many existing non-monotonic ILP systems such as WARMR and CLAUDIEN.

The Apriori algorithm Rules such as one that indicates 95% of customers that purchase toothbrushes also purchase toothpastes are valuable information for cross-marketing. The goal of the Apriori algorithm is to discover all relevant association rules, given a database of purchase transactions. We define a few definitions below that are also relevant to the ILP paradigm. To illustrate with an example, let database $\mathcal{D}$ = {(3,5), (1,2,3), (1,3,4), (1,2,4), (1,2,5), (4)}, where each number simply represents an item, and a tuple represents a transaction.

- item: an entity of interest (eg. 1,2,3,4,5 are items in $\mathcal{D}$);
- Itemset: a set (or a conjunction) of items (eg. (3,5), (1,2,3), etc. are itemsets);
- k-itemset: an itemset containing k items. A 1-itemset contains a single item. (eg. (3,5) is a 2-itemset; (1,2,5) is a 3-itemset);
- Transaction: An atomic entity within a database containing a set of items along with some unique attributes identifying the entity (eg. (1,2,3) is a transaction with ID=2; (4) is a transaction with ID=6);
- Association Rule: an implication of the form $X \Rightarrow Y$, where X and Y are itemsets, and $X \bigcap Y = \emptyset$(eg. (1,2) $\Rightarrow$ (4); (2,5) $\Rightarrow$ (1));
- Support-Count: # of occurrences (ie. frequency) of an itemset of interest within a database of transactions (eg. support-count((1,2),$\mathcal{D}$) = 3)

- Support: Support-Count / Total # of transactions (eg. support((1,2),$\mathcal{D}$)) = 3/6 = 50%);

- Confidence: The association rule $X \Rightarrow Y$ has confidence c = support($X \bigcup Y$) / support(X) (eg. confidence for rule (1) $\Rightarrow$ (2) = support(1 $\bigcup$ 2,$\mathcal{D}$) / support(1,$\mathcal{D}$) = (3/6) / (4/6) = 75%);

- Minimum support: user-specified threshold for minimum support that an itemset must have for it to be accepted;

- Minimum confidence: user-specified threshold for minimum confidence that an association rule must have for it to be accepted;

- Frequent itemset: support of an itemset larger than minimum support (eg. using the example above, itemset (1,2) is frequent if minimum support $\leq$ 3);

- Strong rule: An association rule with both support and confidence greater than the minimum support and confidence, respectively. By definition, infrequent itemsets cannot produce a strong association rule (eg. (1) $\Longrightarrow$ (2) is a strong rule if min support $\leq$ 3 and min confidence $\leq$ 75%)

Given a database of transactions, Apriori (Algorithm 1) finds all frequent itemsets $\mathcal{F}$ and generates strong rules using $\mathcal{F}$. The algorithm starts by generating candidate 1-itemsets C_1 (CAN-INIT), by simply counting the number of occurrences of each item within the database. The lists $\mathcal{F}$ and $\mathcal{I}$ are used to keep track of frequent and infrequent itemsets respectively. While there exist candidates for the k-itemset, the frequency (support count) of each candidate in C_k is calculated (Algorithm 2). Itemsets that do not meet minimum support are added to the infrequent list, else they are copied to L_k (frequent itemsets for level k) and also added to $\mathcal{F}$. Then, the candidates for the next level (ie. (k+1)-itemset) are generated

Algorithm 1 The Apriori Algorithm

Input: Apriori: Database $\mathcal{D}$; threshold *min_supp*
Output: Apriori: All frequent itemsets $\mathcal{F}$ in $\mathcal{D}$

1. C_k := candidate itemset or query of size k
2: L_k := frequent itemset or query of size k
3. size k := 1
4· Init set of candidate of size k=1: C_k = CAN-INIT($\mathcal{D}$)
5 Init set of global infrequent itemsets/queries $\mathcal{I}$:= 0
6: Init set of global frequent itemsets/queries $\mathcal{F}$:= 0
7· **while** C_k not empty **do**
8· freq(k) = CAN-EVAL($\mathcal{D}$, C_k)
9. $\mathcal{I}$ += all candidates in C_k with frequency < min_support
10 L_k := C_k / $\mathcal{I}$
11 $\mathcal{F}$:= $\mathcal{F} \bigcup L_k$
12 C_{k+1} := CAN-GEN(L_k, $\mathcal{F}$, $\mathcal{I}$)
13 increment k
14 **end while**
15 **return** $\mathcal{F}$

(Algorithm 3), using the frequent itemsets L_k for the current level. The process repeats until there are no more candidates left.

Algorithm 2 CAN-EVAL

Input: Database $\mathcal{D}$; Candidate k-itemset: C_k
Output: Frequencies of all candidates in C_k

1. **for** each candidate c in C_k **do**
2 initialize freq(c) := 0
3: **end for**
4· **for** each transaction t in database $\mathcal{D}$ **do**
5 **for** each candidate c in C_k **do**
6. freq(c) += 1 if c is contained in t
7. **end for**
8 **end for**
9 **return** freq(c): for all c in C_k

The other half of the algorithm generates strong association rules based on the frequent itemsets $\mathcal{F}$ found. Algorithm 4 shows the pseudo-code for association rule generation. For each frequent itemset l in $\mathcal{F}$, every non-empty subset s of l

Algorithm 3 CAN-GEN

Input: infrequent itemsets $\mathcal{I}$; global frequent itemsets $\mathcal{F}$, frequent k-itemsets L_k
Output: Candidate (k+1)-itemset C_{k+1}

1: *Joinstep* : C_{k+1} is generated by joining L_k with itself
2. *Prunestep* :
3: **for** each candidate c in C_{k+1} **do**
4: **if** any subsets of c is in $\mathcal{I}$ (ie. infrequent) **then**
5: remove c from C_{k+1} (Apriori property)
6: **end if**
7: **end for**
8 **return** C_{k+1}

is generated. Line 4 and 5 indicates the association rule is strong is if s, being the body of the rule and l - s, being the head of the rule, has a confidence greater than the minimum.

Algorithm 4 Association Rules Generation

Input: Global frequent list $\mathcal{F}$, Minimum Confidence min_conf.
Output: Strong Association Rules $\mathcal{R}$

1: $\mathcal{R} := 0$
2. **for** each frequent itemset l in $\mathcal{F}$ **do**
3. **for** every non-empty subset s of l **do**
4. **if** (support_count(l) / support_count(s)) $\geq$ min_conf **then**
5: $\mathcal{R} := \{\mathcal{R}, s \rightarrow (l\text{-}s)\}$
6· **end if**
7· **end for**
8. **end for**
9· **return** $\mathcal{R}$

The efficiency of the algorithm is due to the Apriori property which states that all subsets of a frequent itemset must also be frequent. This allows efficient pruning of the search space by comparing each candidate to the infrequent list. Any candidate that contains any subsets which is infrequent is pruned, and thus will not be part of the candidate set for the next level This property is also borrowed by ILP algorithms for pruning the search space.

Background knowledge cannot be used with the Apriori algorithm, nor a language bias to guide the search space. The algorithm simply searches through the entire combinatorial space available with the given items.

Illustration Appendix A is used to illustrate the Apriori algorithm. Figure 7 shows the steps taken to generate each candidate set, and ultimately the entire frequent list $\mathcal{F}$. Minimum support and confidence are set to 25% and 75%, respectively (ie. support-count = 8 examples $\times$25% = 2). The figure also shows the association rules generated from a frequent 3-itemset.

In Apriori terms, a scene can be represented as a transaction, and objects and action are represented as items. A limitation is that negation of an item is not allowed. C_1 simply contains all 1-itemsets with their support-count. L_1, L_2 and L_3 are created by eliminating the infrequent itemsets from the corresponding C_k. C_{k+1} is generated from L_k by joining all itemsets to itself, with redundant ones removed. The Apriori property is illustrated when generating C_3. For example, the pre-pruned version of C_3 contains {p1,gr,dash}, but the subset {gr,dash} has a support-count of only 1 (as seen from C_2). Therefore, the itemset is removed in the prune step of the CAN-GEN procedure. Such pruning makes the size of the candidate sets much more manageable. The algorithm terminates when C_4 becomes empty, since the joined step creates {p1,gr,ball1,dash} which is infrequent.

In the association rule generation step, the algorithm attempts to generate rules for all frequent itemsets. The example takes the itemset {p2,ball1,turn} and generates 6 rules out of it, of which only two meet the minimum confidence of 75%. Notice that there is no limit as to the number of items that can appear in either the head or body of an association rule: the algorithm simply exhaustively tests all combinations that it can generate.

With Apriori introduced, we can move on to study two non-monotonic ILP

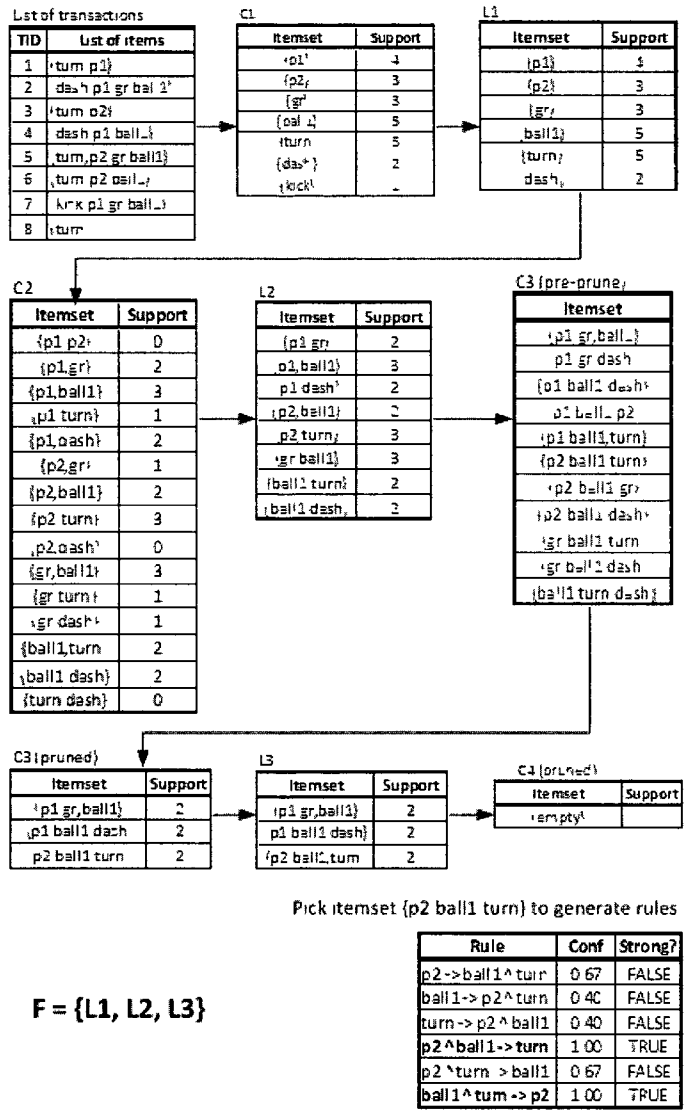

Figure 7: Apriori Example

systems: WARMR and CLAUDIEN.

WARMR WARMR ([Deh99], [DT98], [DR97], [DT99]) is aimed at mining Association Rules with Multiple Relations (ARMR). WARMR is considered a first-order logic upgrade of the Apriori algorithm. Instead of looking for itemsets, WARMR searches for frequent queries that meet minimum support. Strong association rules that meet minimum confidence are generated based on these frequent queries. WARMR works with queries that are represented in DATALOG [AHV95], a subset of the Prolog language restricted to function-free definite clauses. Figure 8 shows the hypothesis space generated using the language bias in Figure 3, again illustrating the example in Appendix A with minimum support and confidence set to 25% and 75%, respectively.

WARMR can be explained using the same algorithm as Apriori. In the candidate evaluation phase (Algorithm 2), instead of counting candidate items in database transactions, the frequency of each candidate query is incremented each time that binding the query (ie. instantiates the query variables with a set of constants) succeeds with respect to the database. Unlike Apriori, WARMR allows the user to specify background knowledge (as shown in figure 4) to provide additional information or constraints to the database for the testing of query success.

Like Apriori, in the candidate generation phase WARMR uses level-wise search to exhaustively traverse through the lattice in the general-to-specific order to look for frequent queries. Each refined query must be mode and type conform, and any specialization of an infrequent query under θ-subsumption is pruned like the Apriori property. Here are some additional observations regarding Figure 8:

- All queries with zero frequency are pruned (minimum support = 2).
- Query #6 is pruned because it is subsumed by #9.

#	Lattice	Support
1	?- key(A)	
2	+- key(A), action_turn(A,farright)	3
3	| +- key(A), action_turn(A,farright), objectSeen(A,C)	2
4	| | +- key(A), action_turn(A,farright), objectSeen(A,B), objectSeen(A,C)	2
5	| | | +- ..	
6	| | +- key(A), action_turn(A,farright), objectSeen(A,C), objectSeen(A,p1)	pruned
7	| | +- .	
8	| | +- key(A), action_turn(A,farright), objectSeen(A,C), dirQual(A,C,B)	2
9	| +- key(A), action_turn(A,farright), objectSeen(A,p1)	1
10	| +- key(A), action_turn(A,farright), objectSeen(A,p2)	1
11	| +- key(A), action_turn(A,farright), objectSeen(A,ball1)	0
12	| +- key(A), action_turn(A,farright), objectSeen(A,gr)	0
13	| +- key(A), action_turn(A,farright), not_objectSeen(A,p1)	2
14	| +- key(A), action_turn(A,farright), not_objectSeen(A,p2)	2
15	| +- key(A), action_turn(A,farright), not_objectSeen(A,ball1)	3
16	| | +- **key(A), action_turn(A,farright), not_objectSeen(A,ball1), objectSeen(A,C)**	2
17	| | +- key(A), action_turn(A,farright), not_objectSeen(A,ball1), objectSeen(A,p1)	1
18	| | +- key(A), action_turn(A,farright), not_objectSeen(A,ball1), objectSeen(A,p2)	1
19	| | +- key(A), action_turn(A,farright), not_objectSeen(A,ball1), objectSeen(A,ball1)	0
20	| | +- key(A), action_turn(A,farright), not_objectSeen(A,ball1), objectSeen(A,gr)	0
21	| | +- key(A), action_turn(A,farright), not_objectSeen(A,ball1), dirQual(A,C,B)	illegal
22	| +- key(A), action_turn(A,farright), not_objectSeen(A,gr)	3
23	| +- key(A), action_turn(A,farright), dirQual(A,C,B)	illegal
24	+- key(A), action_turn(A,farleft)	2
25	| +-	
26	+- key(A), objectSeen(A,B)	7
27	| +- key(A), objectSeen(A,B), action_turn(A,C)	pruned
28	| +- key(A), objectSeen(A,B), objectSeen(A,C)	5
29	| +- key(A), objectSeen(A,B), objectSeen(A,p1)	4
30	| +- .	
31	| +- key(A), objectSeen(A,B), not_objectSeen(A,ball1)	2
32	+- key(A), objectSeen(A,p1)	4
33	+- .	
34	+- key(A), not_objectSeen(A,ball1)	3
35	+- key(A), not_objectSeen(A,gr)	5
36	| +- key(A), not_objectSeen(A,gr), action_turn(A,farright)	
37	| +- key(A), not_objectSeen(A,gr), objectSeen(A,C)	
38	| +- key(A), not_objectSeen(A,gr), objectSeen(A,p1)	
39	| +- .	
40	| +- key(A), not_objectSeen(A,gr), not_objectSeen(A,p1)	
41	| +-	
42	| +- key(A), not_objectSeen(A,gr), dirQual(A,B,C)	illegal
43	+- key(A), dirQual(A,B,C)	illegal
44	| +-	

Figure 8: WARMR Example

- For query #19, it would be useful to specify background theory so that such contradiction should not be generated.
- Illegal queries will not even be generated by the refinement operator in the CAN-GEN procedure. Query #21 and #23 are illegal because the variable C is an input variable, but it has not been instantiated by any previous atom.
- Query #27 is redundant (because of #3) and is therefore pruned
- Queries #36 to #41 (as well as many unshown queries) are also redundant (due to queries visited before), and so they will be pruned in CAN-GEN.

Using the frequent queries found, strong association rules are generated in the last phase, just like in Apriori. All combinations of atoms within a query are tested for association rule generation. In figure 9, we take the query {key(A), action_turn(A,B), not_objectSeen(A,ball1), objectSeen(A,C)} and generate 6 rules with it.

Rule	Conf	Strong?
action -> not_obj, obj	0.67	FALSE
not_obj -> action, obj	0.67	FALSE
obj -> not_obj, action	0.29	FALSE
action, not_obj -> obj	0.67	FALSE
action, obj -> not_obj	1.00	TRUE
not_obj , obj -> action	1.00	TRUE

Figure 9: WARMR Generated Association Rules

Out of the 6 rules, 2 rules meet the minimum confidence of 75%. The rule "not_objectSeen(A,ball1), objectSeen(A,C) → action_turn(A,farright)" contains an essential piece of behavior of Krislet: the agent keeps turning far-right to look for the ball if it does not see it. Query #15 will also generate a strong association rule "not_objectSeen(A,ball1) → action_turn(A,farright)" that has 100% confidence. This

brings out an interesting point. WARMR (and Apriori) keeps looking for frequent queries/itemsets, even if a previous query or itemset more *general* than itself has already been found. This means both of these algorithms do not aim to look for only maximally general hypotheses, but *all* that meet the minimum requirement.

CARMR Many frequent queries found by WARMR may cover the same examples, meaning that they are equivalent/redundant. CARMR [DRR04] provides an optimization over WARMR that introduces various types of condensed representations in order to avoid generation of such redundant clauses. It aims at finding only one representative for each equivalence class of frequent patterns instead of finding all frequent patterns. Also, a concept called *δ-freeness* was introduced. Basically, δ is the number of exceptions a clause might make on the data that still get accepted as a rule (eg. $\delta = 0$ means 100% accuracy). Conceptually this is equivalent to the user-specified minimum confidence. CARMR generates strong-rule during its search for frequent queries, unlike WARMR which generates them only after all frequent queries are found. This makes CARMR more efficient since weak rules (which still meet minimum support) are not being further refined.

CLAUDIEN Like WARMR, CLAUDIEN [DD97] (CLAusal DIscovery ENgine) finds a set of first-order clauses that represent interesting regular patterns within the constraints of a language bias similar to WRMODE, given a set of unclassified observations. An example is shown in Figure 10.

```
modeh(1,action_turn(+int,+dir)).

modeb(2,objectSeen(+int,-obj))
modeb(2,objectSeen(+int,#obj)).
modeb(1,not_objectSeen(+int,#obj))
modeb(2,dirQual(+int,+obj,-dir))
```

Figure 10: WRMODE language bias example for CLAUDIEN

Since CLAUDIEN searches for clauses instead of queries, the *action_turn* predicate is now declared as the head and the rest declared as body, because we want the final sets of clauses to look like "$situation \Longrightarrow action$". This simple difference leads to a significant reduction in the hypothesis space, since now the predicate *action_turn* is confined in the head only, which is not the case for WARMR. This can be seen in Figure 11 which shows the induced hypothesis space by the above language bias, with minimum support and confidence are set to 2 and 100% respectively. The positive coverage p(c) and negative coverage n(c) indicate the # of observations where clause c is true and false, respectively.

#	Lattice	p(c)	n(c)
1	{true}		
2	+- action_turn(A,B)	5	3
3	+- action_turn(A,B) <- objectSeen(A,C)	4	3
4	| + action_turn(A,B) < objectSeen(A,C), objectSeen(A,D)	4	3
5	| | +		
6	| +- action_turn(A,B) <- objectSeen(A,C), objectSeen(A,p1)	1	3
7	| +-		
8	| +- action_turn(A,B) <- objectSeen(A,C), dirQual(A,C,B)	2	5
9	+- action_turn(A,B) <- objectSeen(A,p1)	1	3
10	**+- action_turn(A,B) <- objectSeen(A,p2)**	3	0
11	+ action_turn(A,B) < objectSeen(A,ball1)	2	3
12	| + action_turn(A,B) < objectSeen(A,ball1), objectSeen(A,C)	2	3
13	| +- action_turn(A,B) < objectSeen(A,ball1), objectSeen(A,p1)	0	3
14	| **+- action_turn(A,B) <- objectSeen(A,ball1), objectSeen(A,p2)**	1	0
15	| +- action_turn(A,B) <- objectSeen(A,ball1), objectSeen(A,ball1)	pruned	
16	| +- action_turn(A,B) <- objectSeen(A,ball1), objectSeen(A,gr)	1	2
17	| +- action_turn(A,B) <- objectSeen(A,ball1), dirQual(A,C,B)	illegal	
18	+- action_turn(A,B) <- objectSeen(A,gr)	1	2
19	**+- action_turn(A,B) <- not_objectSeen(A,p1)**	4	0
20	+- action_turn(A,B) <- not_objectSeen(A,p2)	2	3
21	**+ action_turn(A,B) < not_objectSeen(A,ball1)**	3	0
22	+- action_turn(A,B) <- not_objectSeen(A,gr)	4	1
23	+- action_turn(A,B) <- dirQual(A,C,B)	illegal	

Figure 11: CLAUDIEN Example

Here are some more observations regarding the example:

- All clauses with positive coverage under 2 are pruned.

- All clauses with 0 negative coverage (100% confidence) and positive coverage > 2 are added to H, and are no longer refined (eg. clauses #10, #19, #21).
- Clause #15 is pruned since it is redundant with "action_turn(A,B) ← objectSeen(A,ball1)".
- Illegal clauses will not even be generated by the refinement operator. Clause #17 and #23 are illegal because the variable C is an input variable, which must be instantiated by a previous atom.

Table 7 lists the similarities and differences between WARMR/CARMR and CLAUDIEN.

Feature	WARMR	CARMR	CLAUDIEN
Generality Order	θ-subsumption	same as WARMR	same as WARMR
Query/Clause Mining	Query (then generate clauses)	Clauses	Clauses
Acceptable Criteria	Meets min support/confidence	same as WARMR	same as WARMR
Noisy Data Handling	relax minimum confidence below 100%	same as WARMR	same as WARMR
Search Strategy	level-wise search	same as WARMR	can be depth-first, best-first, breadth-first

Table 7: Comparison between WARMR/CARMR and CLAUDIEN

2.2.6 Summary

In this Chapter, we presented some popular ILP algorithms from two major ILP families. There are other well-known ones such as CIGOL [MB88], which is a bottom-up, interactive, incremental learner, and GOLEM [MF90], a bottom-up, non-interactive, batch learner. In this thesis we focus only on top-down algorithms that searches from most general clauses to most specific.

From the discussion in this section, we realize that the two major ILP settings offer different ways of discovering rules that match the underlying pattern. Therefore, it will be interesting to compare the two different approaches in the RoboCup context. In Section 3.4, we shall revisit this and look at the choice of suitable ILP algorithm(s) in existing literature. Next, we shall look at existing literature on classical imitation techniques, spatial ontologies and spatial learning techniques using ILP.

Chapter 3

State of the Art

3.1 Introduction

It has been recognized that it is not an easy to mine data that contains a spatial dimension [ML01]. In this section, we shall study some existing techniques and tools used to imitate behavior, with the specific focus on spatial domains. Section 3.2 discusses some traditional methods like decision trees [THT02b, THT02a, RV00], reinforcement learning [KLS07, CRCB08], neural networks [MKKP05] that have been popular among researchers and case-based reasoning [RVM+06, LE05, WGL98, FE08].

Section 3.3 discusses spatial relation systems and spatial ontologies in first-order logic, leading to section 3.4 which discusses the various ILP learning techniques used. Discussions in these two sections are based heavily on four existing works, which deal with spatial pattern learning for the purpose of prediction or verification. Menaka and Santos [RK06, SNM08] attempted to learn an underlying protocol of object/block placement on a tabletop using ILP, while Driessens [DJJ+98] focused on verifying a RoboCup agent's high-level features using rules generated. Lattner [Lat07, LH05] attempts to discover rules that describe high-level behavior in RoboCup games. Section 3.3.3 focuses on the spatial ontology that these works

used, while section 3.4.2 focuses on the techniques and tools they used to perform the actual learning.

3.2 Existing RoboCup Imitation Techniques

3.2.1 Classical Machine Learning

Induction of decision trees, such as the popular ID3 [Qui86] or its upgrade C4.5 [Qui93], which the ILP algorithm TILDE is based on, has been widely-used for generating predictions based on a given set of input stimuli and outcome. [THT02b] and [THT02a] examined classification of the decisions in RoboCup using hierarchical multiple C4.5 decision trees. The idea is that since kick, dash, dribble and shoot classes each have their own subclasses, namely, strengths and shoot directions, and so the authors propose that creating a hierarchical decision tree will increase the ability in generalization, and misclassification in the action subclass will be a less catastrophic mistake once the tree at the top layer correctly classifies the parent action class. This approach has shown improvement over a single decision tree. In [RV00], a feature set is collected from raw data of the simulation consisting of locations of players and the ball over time. The raw data is broken into windows of fixed size, where different features are extracted based on a pre-defined high-level behavior of a Robocup agent (eg. a pass or shot). One feature set is collected for each team, and the goal is to identify which team is currently playing given a series of high-level behavior observed. A similarity metric is developed, like those used in the CBR setting, to compare the current feature set observed with the stored feature sets for all teams. Classification is done by training with C4.5 decision tree and with a standard nearest-neighbor approach, with the former clearly outperforms the latter. Another work [VW03] uses decision

trees to learn propositional rules, where each proposition represents specific properties or attributes such as distance from ball to goal; number of defenders within the penalty area; distance to the next team-mate; and so on. Other works involve using reinforcement learning to learn half-field offense [KLS07] and defense by a goalkeeper and defender [CRCB08].

3.2.2 Case-Based Reasoning (CBR)

In case-based reasoning, a situation per unit time is represented as a case. In the context of RoboCup, the situation can be either a simple observation during that time unit as in [LE05] and [RVM+06], or a high-level set game play such as attacking moves or pass blocking in [KNS03]. Each case is also associated with a solution provided for the situation observed during that time unit. This solution can be an action or a sequence of actions performed by the RoboCup agent. The idea of CBR is that a collection of cases is stored in a knowledge base as past experiences, and then later current problems can be solved by retrieving similar cases from the knowledge base. Many have used CBR for imitating behavior of RoboCup agents, or to complement other techniques to improve results.

Wendler et al. [WGL98] employ CBR to learn high-level behavior by determining a preferred position of a player given a situation. Scenes of a game are stored as cases in a Case Retrieval Net (CRN), and similarity propagation within the CRN is used to determine the similarity of cases against the current situation. Marling et al. [MTG+03] use CBR for selecting team formations and recognizing game states. Their approach requires a complete world view which is given only to the coach of each Robocup team. Ros et al. [RVM+06] use CBR to imitate the behavior of Sony AIBO robots in the Robocup Four-Legged League. In this particular league, a robot captain is responsible for retrieving a case and informs the rest of the players

which actions they should perform. Each case is represented by features such as positions of the robot itself, the ball, teammates and opponents, as well as the timing of the match and the goal difference of the two teams. The most similar cases are retrieved, and the one that costs the least to adapt to the current situation to the case will be selected. Karol et al. [KNS03] use CBR to imitate the AIBO robots by categorizing situations into conceptual spaces, in which objects are categorized based on how similar they are to a prototype. High-level strategies are learned based on a case base that is manually picked by experts to include prototypical situations and important situations within a soccer match.

Steffens [Ste04] attempts to eliminate the comparison of irrelevant features by adapting similarity functions to the ball-owner dynamically in real-time. The assumption is that the set of features which is relevant or not is directly related to the role of the agent owning the ball (ie. forward, defense, etc.). The technique uses a goal dependency network to determine features that are related to goals and subgoals of the player.

While the previous techniques may be effective, they require extensive expert knowledge in areas such as picking the relevant features for a case, finding the right sets of cases for learning, determining specific parameters to be used in similarity functions, etc. In some cases, such as the Robocup Four-Legged league and the Small Size League, the centralized control model might not be realistic in a multi-agent system where agents act autonomously and do not possess a complete world view.

Autonomous Behavior Framework with CBR

Research performed in our lab by Michael Floyd et al. ([FEL08], [FE08], [LE05], [FDE08]) yielded a CBR framework for imitating RoboCup players. Unlike the previous approaches, this attempts to provide

complete autonomy in all aspects of learning and eliminates the requirement of a complete world model, expert knowledge or any domain-dependent assumptions.

In the current implementation, each case only considers information given in the current cycle and does not carry memory of the past. During a game, cases are retrieved and compared to the current situation using a distance calculation. The one being used requires objects to be paired up across the current scene and each retrieved case. It then sorts the objects based on their ego-centric distance to the agent and greedily matches them pair-wise. Therefore, a short distance indicates the current scene is a close match to the retrieved case. This method does not provide optimal matching but nevertheless provides quick results to meet the real-time RoboCup constraints. In addition, different types of objects may carry different significance or relevance. Each type of object is given a weight which contributes to the final similarity measure. The weights are calculated automatically using a genetic algorithm [FDE08], which aims to maximize the global f-measure. k-nearest-neighbor search is used to select the k most similar cases, and majority vote is used to pick the best case. The agent then executes the action by sending the solution from this case to the RoboCup server. Improvements are made to the case base in [FE08] by using pre-processing techniques such as feature selection, case clustering and prototyping. Multiple experiments have shown promising results using these techniques, which shall be compared with our ILP learning results in Section 6.5.

The framework is not able to identify multiple-states of behavior and only considers visual stimuli. Therefore the imitative agents have difficulty imitating multi-state agents or agents that rely heavily on inter-agent communication.

3.2.3 Discussion

We saw in this section that decision trees such as C4.5 have been used extensively in the field of RoboCup to generate various kinds of if-then-else propositional rules. Their effectiveness is well-known, and hence it is not a surprise that ILP algorithms such as FOIL and TILDE, etc., are derived from decision tree algorithms. However, decision trees are more suited to problems of attribute-value nature in which no substantial relationship between the values of the different attributes needs to be represented. This is also true for neural nets in general. First-order rules learned by ILP systems have much more representational power than propositional rules that are derived from decision trees. Such expressiveness is needed in complex settings like RoboCup where multiple predicates are needed to represent relationships between two attributes (eg. distance/orientation between two visible players), or where a variable is needed to represent a general value for an attribute (eg. any player that belongs to the opponent team). Regarding reinforcement learning, most often an expert is required to pick a particular feature to learn (eg. a defensive or an offensive action), and rewards and penalties must be assigned accordingly. Hence it does not meet our goal of aiming toward a complete autonomous approach without a priori knowledge of the application. Case-based reasoning allows past experience to be stored and retrieved at a later time for use when similar situation arises. The framework in [FEL08] has shown that it is possible, though with performance limitations, to automate the process of learning RoboCup actions given the current situation. First-order rules learned by ILP systems should be more "compact" in representing the underlying behavior, hence real-time performance should not be a concern. Another advantage of ILP over CBR is the fact that the rules learned using ILP are human-readable, hence easy to understand and troubleshoot, whereas the comparison between current situation and the case base in CBR is through some

mathematical similarity metric, which is more difficult to comprehend.

Now that we have studied some existing imitation techniques and laid out some benefits of using ILP, we can turn our focus to learning with ILP in the RoboCup domain. But before we dive into it, we need to look at how space is represented, as well as spatial ontologies that are currently used in the literature. Such knowledge is preliminary to using ILP for learning in the RoboCup domain, which is spatial in nature.

3.3 Spatial Ontology

3.3.1 Introduction

Since we have decided to use first-order logic as our representation language in the RoboCup domain, it is important to clearly define the predicates that we shall employ. Hence, an ontology is necessary. Because this domain is spatial in nature, most of the ontology involves spatial predicates, with some non-spatial ones as well. The temporal aspects were deemed out of our scope (ie. our agent is memoryless, reactive to situation happening only at the current moment).

Batsakis et al [BSar] provided a good summary of spatial relations categorized into three different types: direction, distance, and topological relations. This is shown in figure 12.

The topological relations are used to describe relations between two non-point objects. The system shown in figure 12 is referred to as the RCC-8 relations [WZ00]. The acronyms stand for disconnected (DC), externally connected (EC), partially overlapping (PO), tangential proper part (TPP), tangential proper part inverse (TPPi), non-tangential proper part (NTPP), and non-tangential proper part inverse (NTPPi). The direction relations shown are the eight cardinal directions [Fra96]

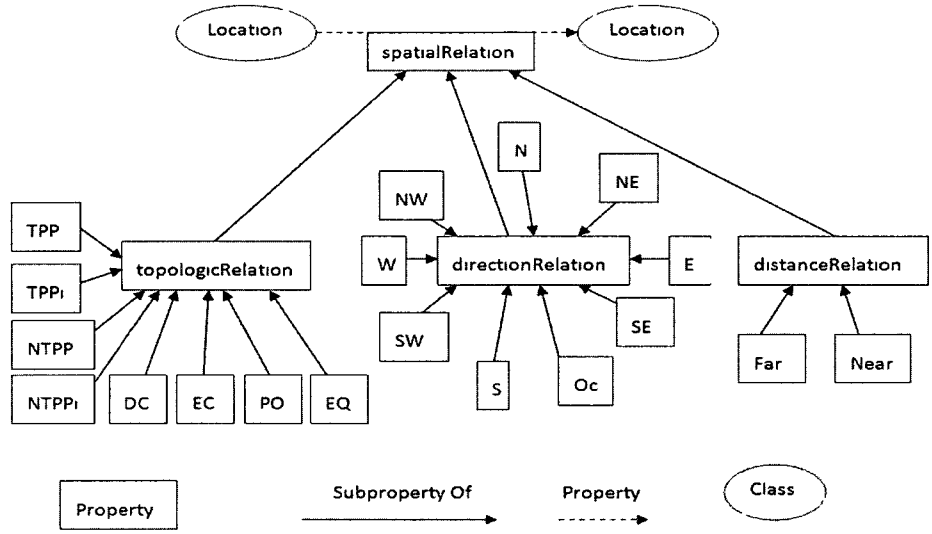

Figure 12: Ontology schema with all spatial relations [BSar]

representing *North, East, South, West,* etc. The distance relations shown simply distinguish between near and far. Figure 13 shows various levels of distance and orientation distinctions [CDFH97].

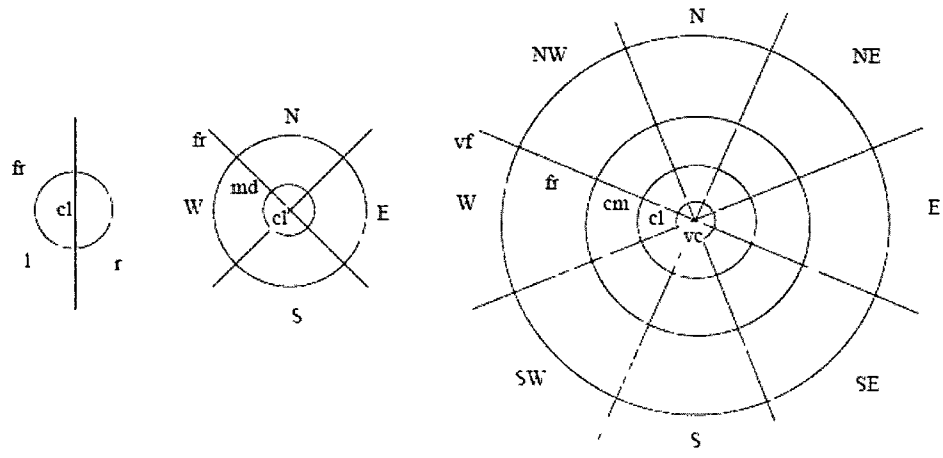

Figure 13: Various levels of distance and orientation distinctions [CDFH97]

These spatial relations must be discussed in the context of a frame of reference,

which is the topic of the following section.

3.3.2 Spatial Relations

Introduction

As mentioned, spatial relations are usually categorized into direction, distance, and topological relations. Direction relations, in particular, are defined in terms of three basic concepts: primary (target) object, reference object, and the frame of reference (FofR) [CH01]. Hence, we shall discuss FofR first, before getting into spatial relations.

Frame of Reference

A reference frame is a means of representing locations of entities in space. Three reference systems that have been widely used across fields of psychology and philosophy are the *intrinsic*, *relative* and *absolute* systems [Lev96]. Table 8 compares the three systems as defined by Levinson [Lev96]. Each system has unique properties, pros and cons, and relevance under different circumstances. As each reference system represents space through different perspectives, we need to understand them better in order to develop a concise, yet powerful, set of predicates to represent these concepts.

Two other spatial reference frames are also very much related: *ego-centric* and *allocentric* [Kla98]. The ego-centric reference frame represents objects orientation and distance with respect to the observer's point of view. In contrary, allocentric reference frame conveys the position of every object by its distance and bearing with respect to an arbitrary reference as origin regardless of any ego's viewpoint. In other words, the ego has no special status with respect to any other objects within the domain. We shall focus on the egocentric reference frame, as this is how

Types	Descriptions	Viewpoint	Predicate Representation	Example
Intrinsic	Relation (rel) of a target object (F) relative to a reference object (G) with respect to that object's intrinsic front, back, left, right	Object-centered (viewpoint-independent)	*rel*(F,G), F and G may be any objects including ego	ball (F) is in front of (*rel*) goal (G)
Relative	Relation of a target object (F) relative to a reference object (G) with respect to the viewer's (V) front, back, left, right	Viewer-centered	*rel*(V,F,G), V and G must be distinct	ball (F) is left of (*rel*) goal (G) with respect to the viewer (V)
Absolute	Relation of a target object (F) relative to a reference object (G) with respect to cardinal directions (eg north, south, east, west)	Environment-centered	*rel*(F,G), F can be found in a domain at the fixed bearing R from G; F may be part of G	ball (F) is west of (*rel*) goal (G)

Table 8: Three Frames of Reference described by Levinson [Lev96]

a RoboCup agent perceives its surrounding space.

The intrinsic reference represents relationships between two objects with respect to the reference object's intrinsic orientation, independent of the viewpoint of any observer. The object's intrinsic orientation can be induced by its motion (eg. a ball moving), as well as its intrinsic front (eg. a person's face). Such reference system allows the following to be expressed concisely: "Given that the ball is close to (or in front of) an opponent, perform an intercept." Thus both orientation and distance are represented, intrinsic to the opponent and the ball themselves. However, with this reference system, relations between objects and the observer are missing.

The relative reference represents the relation between two objects with respect to an observer. Such reference system allows the following to be expressed concisely: "Given that an opponent is on the right side of teammate, pass to the teammate from the left side." Thus, only the ego-oriented bearing of the target object is represented. Inter-object distances are missing. The relative reference frame has converse and transitive properties, such that new relations can be deduced from existing ones.

The absolute system describes relations between objects in cardinal directions (north, east, south, west - see next section). It requires the observer to know its current orientation with respect to an absolute reference direction. Thus, implicit knowledge needs to be given to the observing RoboCup agent that flags, lines,

etc. are stationary and their respective orientation with respect to the cardinal directions. This system has been used by many [LMVH05,DJJ+98,Lat07]. However, since our goal is to have minimal amount of implicit knowledge built-in to our apprentice agent, we shall not consider this reference system in this thesis.

It is clear that neither intrinsic or relative representations can be used on their own, since they do not capture complete information as in the egocentric representation. On the other hand, the egocentric system does not explicitly define many useful relationships between the observed objects themselves, as expressed in the intrinsic and relative systems, which may be significant to the observer's behavior. In addition, it can be seen that the latter two representations complement each other without overlapping, as they represent spatial relationships from different perspectives. As such, it is difficult to say which one is better than the other in capturing the essential and relevant information, but that can lead to more accurate and efficient spatial learning. Therefore, we will seek to compare these representations in this thesis.

Now that we have examined the different frames of reference available, we can proceed to discussing the different spatial relations: direction, distance and topology.

Direction (Orientation) Relation

Figure 13 shows the well known concept of cardinal directions [CDFH97] that represents the angular direction between the observer's position and a target position. This system is considered to be an absolute reference system, where the origin is situated in the middle and target objects can be anywhere around it If the observer is also in the origin, it can also be viewed as the egocentric system. The figure shows three different systems where directions are divided into two, four, and eight cone-shaped subdivisions, respectively. An identity symbol 0_c can be

used to identify the center (neutral) point. This can be used to identify objects that are very close to the observer. The set of eight cardinal directions and an identity symbol, is denoted as $C_8 = N, NE, E, SE, S, SW, W, NW, 0_c$. The system exhibits many useful properties such as inverse, associativity, transitivity, and composition. For example, SW combined with SE should result in S; N combined with E should result in NE, and so on.

An alternative semantics for the cardinal direction is defined by 2D projections. The four directions, as shown in figure 14*a*, are defined by two sets of half planes that are pair-wise opposites, unlike the cone-shaped models as in Figure 13. It has been said [Fra92] that cone-shaped directions better represent the notion of 'going towards', whereas the projection-based directions better represent the relative position of points on a map. Similar to the cone-shaped cardinal directions, composition can also be performed in this system. Figure 14*b* shows the same system with a neutral zone. The width of the neutral zone is not determined explicitly, but must be fixed and consistent within a context.

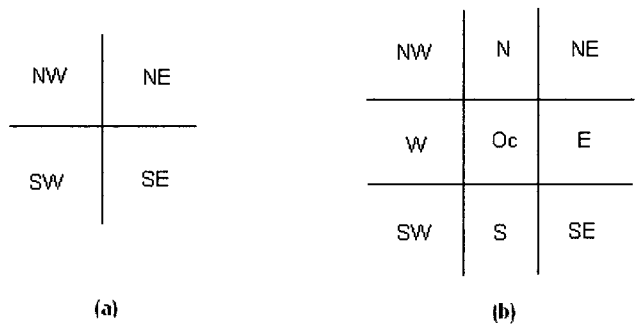

Figure 14: Directions defined by half-planes (a); Directions with neutral zone (b)

This system is used by Lattner [MLVH04] when describing vehicle motions, as well as by Menaka [RK06] without the neutral zone, where the largest object was chosen to be the reference object. Santos [SNM08] defined a relative reference

system named *LocalCardinalSystem* (LCS), whereby each object in the observed situation is located according to a cardinal reference frame defined by its nearest object. In addition, an object is only described within the reference frame of another only if the former is placed *after* the latter. Thus, an implicit temporal ordering constraint is imposed in the way objects are represented within LCS.

Figure 15 shows a simple model for the left/right-dichotomy in a relative reference system [MR08], using the *origin* and *relatum* to define the reference axis. The terms origin, relatum and referent were introduced by Levinson [Lev96], where relatum is the reference object, and referent is the target object. In the figure, the reference lies to the left of the relatum as viewed from the origin.

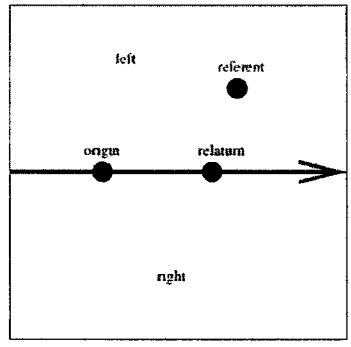

Figure 15: The left/right-dichotomy in a relative reference system

Ligozat extended this system by subdividing the position of the referent with respect to the relatum and origin into five partitions: behind origin, same as origin, in front of relatum, same as relatum, and behind relatum. This results in the representation used by the flip-flop calculus [Lig93]. The system represents "front" and "back" only as linear acceptance regions Freksa's single cross calculus [Fre92] extended this by combining the front/back and left/right models with 8 primitive relations, and was again extended into double cross calculus [Fre92] with 15 primitive relations. All three systems are shown in in figure 16 (a), (b), (c),

respectively.

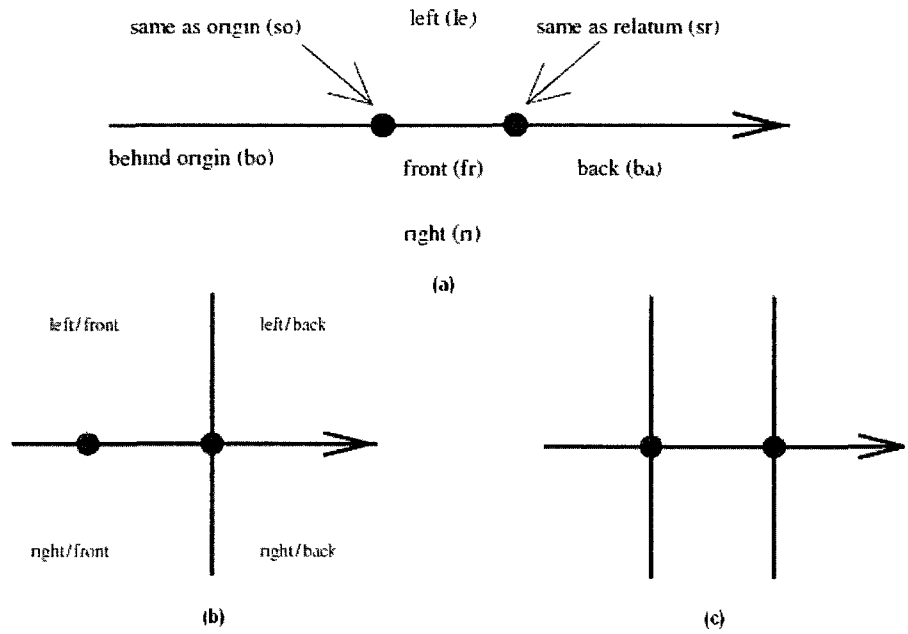

Figure 16: Reference System used by Flip-Flop (a) by Single Cross (b) and Double Cross (c) Calculus

Moratz et al [MR08] presented the reference system used by *Ternary Point Configuration Calculus* (TPCC), which was derived from the single cross calculus. The reference system, as shown in figure 17, makes finer distinctions on the regions by dividing them into 45 degrees angles, making a total of 25 distinct regions. The letters f,b,l,r,s,d,c stand for front, back, left, right, straight, distant, and close, respectively. The origin "sam" stands for "same", which represents the relatum. The black dot represents the origin, and the black square represents the referent.

They concluded that the TPCC model is especially suited to interpret human spatial references [MTBF03], as it is found to be a valid approximation to the linguistic behavior observed in the real world. Figure 18 illustrates the differences between the different relative reference systems.

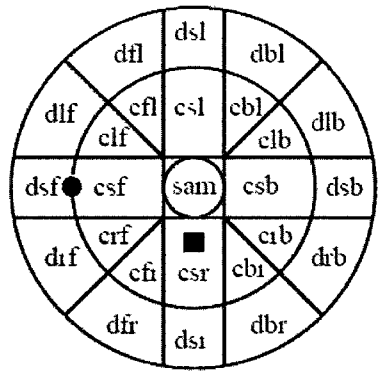

Figure 17: Reference System used by TPCC

	left	behind	leftback
Flip Flop			impossible ?
Double Cross			
TPCC			

Figure 18: Comparisons of different relative reference systems [MTBF03]

Distance Relation

In addition to orientation distinctions, Figure 13 also illustrates the various levels of distance distinctions. The first level of granularity distinguishes between *close* and *far*, in which the plane is divided into two regions centered around the reference object, with the outer region extends to infinity. The level with five distinctions organizes regions into *very close*, *close*, *commensurate*, *far*, and *very far*. Note that these terms do not explicitly imply a numerical value, which is highly context-dependent. For example, 1 meter away from a soccer player may be considered

close, but an ant may consider it very far.

In [HICF95], Hernandez mentioned that distance systems consist of a list of distance relations, and a set of *structure relations*. The first list is the set of qualitative distinctions being made (eg. *close*, *far*), and the second set describes a set of properties specifying how the distance relations in turn relate to each other (eg. order-of-magnitude relations between the various distance ranges). Distance systems with homogeneous properties have structure relations that follow a recurrent pattern. To explain these properties, figure 19 distinguishes between δ_i being the "distance range i", and Δ_i being the "distance range from the origin up to and including the distance range δ_i. The three properties are monotonicity, range restriction, and orders of magnitude.

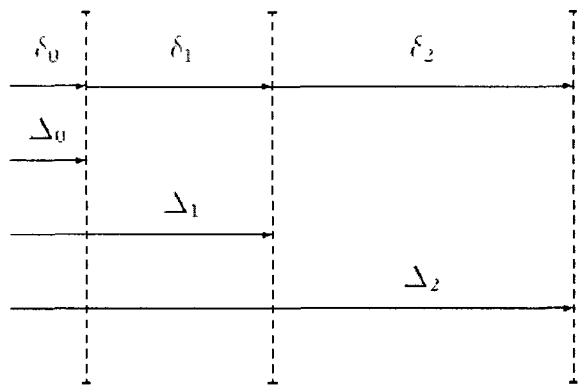

Figure 19: Distance ranges vs. distance from origin [HICF95]

Monotonicity: Any given interval is bigger or equal than the previous one:

$$\delta_0 \leq \delta_1 \leq \delta_2 \leq \ldots \leq \delta_n$$

This property (see figure: 20) came from the observation [CDFH97, HEF95, PZ87] that humans are more inclined to make finer distance distinctions in the neighborhood of the reference object, than when the distances involved get further.

Figure 20: Monotonicity Property of Distance Systems [HICF95]

Range restriction Any given interval is bigger than the entire range from the origin to the previous interval:

$$\delta_i \geq \Delta_{i-1}, \forall\, i > 0$$

Orders of Magnitude If a distance range δ_j is much bigger than a previous one δ_i ($\delta_j >> \delta_i$), then δ_j will absorb δ_i in the composition:

$$\delta_j \pm \delta_i \simeq \delta_j$$

These properties are the most common types of restrictions imposed on homogeneous distance systems, which constrain the resulting sets in the composition of distance relations [HICF95]. On top of this discussion, distance is highly dependent on scale. The FofR influences the scale being chosen. For an intrinsic FofR, the inherent characteristics of relatum such as its size highly determines the scale. For an absolute FofR, distance is determined by some external factor, such as the traveling time or costs involved (eg. a map of Ottawa with unit of "1 km"). In a relative FofR, distance is determined by an external point of view, which can refer to how the observer visually perceives the object.

Topological Relation

Topology is another fundamental aspect of space. It uses shapes and regions as primitives to represent objects, rather than points. Clarke [RCC92] has developed the Region Connection Calculus (RCC) system, which is shown in figure 21. It

illustrates 8 possible relations between two regions and their continuous transitions. The acronyms were already defined in section 3.3.1.

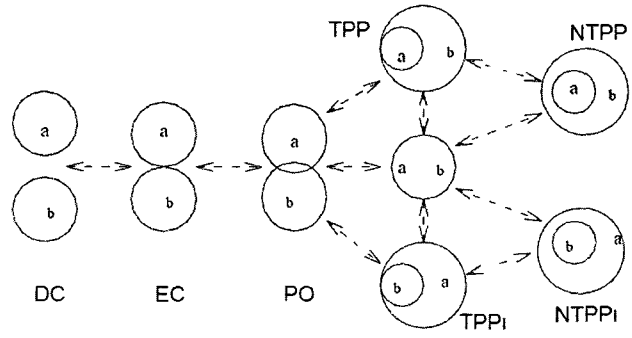

Figure 21: 2D illustrations of the relations of RCC-8 calculus and their continuous transitions [CH01]

In our work within the 2D RoboCup domain, we assume that objects do not overlap. Therefore, topological relations are not relevant to us and shall not be discussed furthermore.

Discussion

All the calculii introduced in this section were conceived to allow qualitative spatial reasoning through manipulation and composition. The basic idea of qualitative reasoning is that the relations between the initial points are only approximate (qualitative) in nature, and the goal is to determine what inferences can be made in the context in a way similar to biological systems. The composition operation combines two relations: given point C in relation to vector $\vec{AB}$, given point D in relation to the vector $\vec{BC}$, deduce the relation of D with respect to $\vec{AB}$ [Lig93]. We can reason about the position of any object from the perspective of any other location in the relations with such operations.

The point of view of our RoboCup teaching agent given through the RoboCup

server is always egocentric, and the apprentice is learning with such egocentric data. Hence, if we want to learn relations in the relative frame of reference, instead of performing composition operations as above, we want to deduce $\vec{AB}$:C or $\vec{AC}$:B from A:B and A:C. Since the egocentric data given is numerical in nature, we can easily determine such relations using euclidean geometry, and then convert the result from quantitative to qualitative, without requiring such calculus for spatial reasoning.

In the next section, we shall look at how spatial predicates are used to represent relations such as distance and orientation, among others.

3.3.3 Spatial Predicates

In this section we shall look at how spatial predicates are defined and used in some existing work. This will help us define our own.

Driessens et al. [DJJ+98] used ILP techniques to perform verification on multi-agent systems (MAS) such as Robocup. A simple agent, KULRoT, was developed, and its actions and perceived state of the world are logged at regular intervals. A sample description of one state (at one particular time cycle) logged is shown in figure 22. Table 9 explains each predicate.

```
begin(model(e647))
   player(my,1,-43 91,5 17,3352)
   player(my,2,-30 02,7 78,3352)

   player(other,10,14 23,15 19,2748)
   player(other,11 0 0,0 0,0)
   ball(-33 73,10 01,3352)
   mynumber(5)
   rctime(3352)
   moveto(-33 73,10 01)
   actiontime(3352)
end(model(e647))
```

Figure 22: Model Representation for KULRot in one state [DJJ+98]

Seen objects were only considered (as part of the model) by the observing player

Predicate	Descriptions
player(*T*, *N*, *X*, *Y*, *C*)	the agent has last seen the player with number *N* from team *T* at location (*X*, *Y*) at time step *C*
ball(*X*, *Y*, *C*)	the agent has last seen the ball at location (*X*, *Y*) at time step *C*
mynumber(*N*)	indicates the model corresponds to the observation of agent *N*
rctime(*C*)	this state was recorded at time step *C*
actiontime(*C*)	the action listed in the current state was executed at time step *C*
moveto(*X*, *Y*), *shoottogoal*, *passto*(*X*, *Y*), *turn*(*X*), *search_ball*, *watch_ball*, *movetoball*, *passtobuddy*, *none*	the agent's action in the current state

Table 9: Predicates used by KULRoT agent

when all identifying information is available. For example, the team name T and the uniform number N of each player must be visible. The absolute coordinates of each object on the field was calculated and recorded, which may not be realistically possible under all circumstances (eg. when not enough information is available to derive such absolute coordinates).

Note that the predicates *player* and *ball* were used to explicitly represent such entities. Some of the actions being recorded (as listed in Table 9) were already at a higher level abstraction than the primitive ones that are executable directly by the SoccerServer (eg. *passto*,*shoottogoal*). This means some domain-specific knowledge was already embedded implicitly in the data by an human expert. But our goal is to be as generic as possible such that learning can be performed in any spatial domain. Hence, the actions we learn from should only be the ones that are directly associated to the actuator (ie. *dash*, *kick*, *turn*).

The language bias, declared through background knowledge, employed predicates from Table 9. Such predicates include *seeball*, *ball_in_my_area*, *haveball*, *ball_near_othergoal*, etc. For example, *haveball* are defined with the following background clause, which indicates that a player *has* the ball if it is within 5 unit distance:

$$haveball() \leftarrow mynumber(A), player(\text{my},A,X1,Y1,C), ball(X2,Y2,C),$$

$$distance(X1,Y1,X2,Y2,Dist),\ Dist =< 5.$$

where the *distance* predicate indicates the Euclidean distance between the two points $(X1,Y1)$ and $(X2,Y2)$ in the argument *Dist*. Predicates like *haveBall* can be generalized to predicates like *haveObj* in order to make them more domain independent.

Lattner [Lat07, LH04] used ILP algorithms to discover frequent patterns in RoboCup games. Similar to Driessens [DJJ+98], the rules learned are at a high-level cognitive abstraction level. They perform experiments in both the RoboCup 2D and 3D simulation leagues. Table 10 shows the predicates used in the 3D scene descriptions.

Predicate	Descriptions
team(P,teamname)	the corresponding team of player *P*
dist(X,Y,distance)	distance between two objects (*distance* = kickable, very close, close, medium, or far)
bearing(P,direction)	direction of player *P* (*direction* = north, north west, west,)
playmode(mode)	the current playmode of the match (*mode* = goalkick, kickoff, playon,)
pass(X,Y)	a successful pass from player *X* to player *Y* is performed
failpass(X,Y)	a failed pass has been performed from player *X* to player *Y*
clear(P)	player *P* cleared the ball
selfAssist(P)	the player *P* moves with the ball (ie dribbling)
inPassRegion(P)	the player *P* is in the region where a pass is performed

Table 10: Predicates available in 3D dynamic scene descriptions

Notice again that some of the predicates used are of higher abstraction level than what could be observed from the raw data. The raw data has been pre-processed to extract information such as a successful pass (*pass*), which can only be determined by examining a sequence of frames, starting from player *X* kicking the ball toward player *Y*, and *Y* successfully obtaining it. The same applies to *failpass*, *inPassRegion*, *clear*, etc. Therefore, like [DJJ+98], such predicates are RoboCup-specific and require human experts to define them in advance. It is again not

surprising that such predicates are employed here since the goal of pattern mining in his work was at the cognitive level, but not at the primitive, execution level.

That being said, many predicates like *team*, *dist* and *bearing* can be directly obtained from the raw data, except for the fact that the bearing of a player was recorded in absolute frame of reference. Since the raw data from the RoboCup SoccerServer is in the egocentric point of view of the observing agent, this implies that the observing agent's absolute coordinates within the field must have been determined via the static objects (flags, lines, goals). This in turn implies assumptions have been made about the mobility of objects, which we would also like to avoid. The *dist* predicates indicates the inter-point Euclidean distance between any two objects, which is something we are interested in as well.

More interestingly, the objects are represented based on class hierarchy. For example, *player* and *ball* are subclasses of *object*, and $p1$, $p2$, b, etc. are declared as direct instances of their corresponding classes. Such hierarchy allows learning to be made more efficient by searching for general clauses that are frequent before refining them. The hierarchical information is specified as background knowledge, as shown in Figure 23.

```
isA(player,object)
directInstanceOf(p1,player)
directInstanceOf(p2,player)

directInstanceOf(q1,player)
directInstanceOf(q2,player)

isA(ball,object)
directInstanceOf(b,ball)
```

Figure 23: Hierarchical information

Figure 24 provides a snippet of how the 3D dynamic scenes in a RoboCup match look like. Scene descriptions for 2D matches are similar to this, although somewhat

different predicates were employed.

The expression "*holds*($p(Arg_1, \ldots, Arg_n)$, *start*, *end*)" indicates that the atom $p(Arg_1, \ldots, Arg_n)$ holds true during the time interval between *start* and *end*. However, as mentioned before we shall not focus on these temporal elements. Each player's uniform number is implicit in the constant term (eg. p6, q8). Also, notice that both *bearing* and *dist* predicates provide qualitative information, converted from the raw numerical values.

Matsui et al. [MIS00] used a first-order formalism and Progol to acquire rules for predicting the results of a RoboCup agent's actions in an attempt to improve its performance. The list of predicates used (listed in figure 25) for learning are mostly implemented as functions in the CMUnited-99 [Uni] RoboCup team.

Most of the predicates are self-explanatory. The predicate *sq_distance*(+*pid*,+*pid*,−*dist*) indicates the square distance between two players. The +*time* parameter indicates the time cycle at which the predicate holds true. Notice that the three types of objects, ball, player and goal, are included in the language, and each object has its own set of *distance*, *angle_from_body*, *angle_from_neck* predicates. It would be appropriate to generalize these predicates into a single set which takes in an object type as a parameter. Furthermore, notice that the predicates *gteq*, *lteq* and = are used to perform numerical comparisons learning by Progol, although the search time could be significantly higher.

Discussion

In this section, we have looked at a fair amount of spatial predicates, especially in the RoboCup domain. The predicates used by Driessens represent players and balls in an absolute frame of reference, making implicit assumptions about the environment which need to be input by a domain expert. Thus, we aim to

```
holds(playmode(kickoffLeft),1,53)
holds(dist(b,q9,far),1,108)
holds(bearing(q9,southWest),1,67)
holds(dist(b,q6,far),5,9)
holds(bearing(q6,south),5,9)
holds(dist(b,q6,far),11,16)
holds(bearing(q6,south),11,16)

holds(inPassRegion(q10),89,99)
holds(inPassRegion(q11),101,108)
holds(dist(p9,q8,medium),110,146)
holds(dist(p9,q8,close),147,152)
holds(dist(p9,q8,veryClose),153,156)

holds(selfAssist(p6),246,260)
holds(pass(q8,q9),428,446)

holds(failPass(p4,q6),516,553)
holds(selfAssist(q6),700,718)
holds(pass(q6,q9),718,753)

holds(selfAssist(q10),831,867)
.
holds(selfAssist(p4),966,980)
holds(failPass(p4,q6),983,1019)
holds(pass(q6,q10),1020,1042)
holds(pass(q10,q8),1047,1071)
holds(failPass(q8,p2),1072,1097)

holds(pass(q4,q8),1383,1416)

holds(pass(q9,q6),1441,1476)

holds(failPass(q8,p3),1559,1587)
holds(failPass(p3,q6),1587,1624)
holds(failPass(q6,p4),1627,1662)

holds(pass(q7,q10),1779,1807)
holds(pass(q10,q8),1808,1829)
holds(failPass(q8,p3),1841,1869)
```

Figure 24: Snippet of 3D match by "MRU vs. ZJU Base" [Lat07]

```
my_body_ang(+time -angle)
my_neck_rel_ang(+time,-angle)
my_speed(+time,-veloc)
my_stamina(+time,-stamina)
ball_distance(+time,-dist)
ball_angle_from_body(+time,-angle)
ball_angle_from_neck(+time,-angle)
ball_speed(+time,-veloc)
ball_moving(+time)
ball_kickable(+time)
ball_catchable(+time)
goal_distance(+time,-dist)
goal_angle_from_body(+time,-angle)
goal_angle_from_neck(+time,-angle)
known_player(+time,-pid)
opponent(+pid)
teammate(+pid)
uniform_number(+pid,#int)
player_distance(+pid,-dist)
player_angle_from_body(+pid,-angle)
player_angle_from_neck(+pid,-angle)
player_speed(+pid,-veloc)
ball_kickable_for_player(+pid)
gteq/2
lteq/2
=/2
sq_distance(+pid,+pid,-dist)
acted_time(+aid,-time)
selected_action(+time,-aid)
action_type(+aid,-action)
previous_time(+time,-time)
```

Figure 25: A list of predicates used by Matsui [MIS00]

represent objects using orientation and distance relations relative to one another. The predicates used by Lattner and Matsui are very much relevant to our own purposes; we can modify them to be made more generic and less RoboCup-domain specific.

In Section 5.5.2, we shall formally describe the spatial predicates that we will be using, based on the discussion in this section. Having studied existing work on spatial ontology and how spatial relations are represented, we can proceed to look at the ILP techniques that put these concepts into use.

3.4 ILP Learning Techniques

3.4.1 Introduction

Since our main objective is to perform behavior imitation on an agent by discovering relevant association rules in the Robocup domain, we want to find a (non-linear) function that maps a given situation into an action, which will then be executed by the imitation agent. The situation, which is represented as a scene, contains spatial

information such as relations of surrounding objects (players, ball, goals, flags, etc.) with respect to the observing agent, etc. Therefore, spatial association rules are of most interest to us.

3.4.2 Spatial Patterns Learning

Menaka et al. [RK06] use ILP to mimic object manipulations in a qualitative approach by learning from visual scenes, where the setting is a set of objects (plates, forks, etc.) being moved across a table. Their system was designed to analyze visual scenes in terms of qualitative object-to-object spatial and temporal relations, and predict where the objects are placed at different moments. Captured attributes included the frame number, type of object, object identifier, center coordinates, bounding box, and whether the object is moving. Direction relations were described using the eight cardinal direction model. Topological relations were identified in terms of RCC-8 (see Section 3.3.1). Allen's temporal interval [All90] was used to account for qualitative temporal relations. Thus, they avoided using quantitative methods of knowledge representation and reasoning. In addition, the entire space of scene was divided into a grid system. If two objects belong to the same tile, the same reasoning mechanism will be used to carry out to find the relative orientation of the two objects.

In a pre-processing phase, spatio-temporal relations between any two objects were first generated from observed quantitative data into qualitative predicates using a Qualitative Knowledge (QK) module. During this phase, positive and negative examples of each predicate to be learned were also generated. These examples are ground facts, so they are highly context-dependent (eg. the time-stamp of each example carries a constant value). In the second phase, Progol was employed to generalize these relations generated by QK into context-independent

rules that satisfy the given positive and negative examples.

In measuring performance on six sets of data [RKR08], three of which are real world data (with noise) and three of which are hand coded data (noise-free), they were able to produce rules with accuracy up to 96%. Human subjects were asked to describe the relations that exist between the objects, and their decisions were comparable to the rules generated by their system. Hence, they argued that exploitation of spatial and spatio-temporal relations is a good approach to learn rules from visual scenes. In their work, Progol was the only algorithm used, without justification or comparison to any other algorithms as to whether they might provide better results. Their use of *QK* to pre-generate spatio-temporal relations is different from our intention of using background knowledge built-in to most ILP algorithms to derive these relations automatically as well as mining association rules, all in the same phase. This would make our system more portable to other applications.

Santos et al [SNM08] also used Progol to obtain a set of rules that describe a simple setting of a set of colored blocks having different arrangements on a table top. Five experiments were captured where blocks were being stacked on the table top, each one having a distinct arrangement. The data obtained was then turned into a symbolic (qualitative) description of states of the objects. Some experiments involved more complicated arrangements, such as having two stacks of blocks, longer repetition sequence, etc. Contrary to the previous works by Menaka et al, only positive examples are used here for learning by Progol. The ability of Progol to induce rules solely from positive examples makes it suitable for learning rules from passive observations, where negative examples may not be available. In this thesis, we shall not use this mode of operation for Progol, as the positive and negative examples can easily be extracted from the data automatically for each action and

its associated (qualitative) parameters. The authors also pointed out that Progol needed at least 20 (noise-free) examples to learn the rules above. Otherwise, the quality of the learned rules degraded proportionally. This is another reason for us not to use positive-only learning, as some actions, such as the action *kick* in Krislet, happen very infrequently.

The works described so far are relatively simple, in terms of the number of objects and attributes being learned in a scene. More complicated examples, such as the RoboCup multi-agent system (MAS), has also been the focus of experiments with ILP. MAS are complex systems that are usually hard to verify against their intended behavior. Driessens et al. [DJJ+98], described in section 3.3.3, used ILP techniques to produce a specification directly from the KULRot agent, so that its behavior can be verified by human expert. The hypotheses learned were in terms of high-level predicates, allowing the human verifier to focus on the features more easily. This helps simplify the language bias and reduces the search space significantly, and also eliminates the need to associate each action with a strength or direction parameter. TILDE and CLAUDIEN were chosen for learning the rules. The two systems complemented each other to find inconsistencies in the agents' actions. TILDE found the decision tree as shown in figure 26.

The predicate *bucket* carries a value which gets decremented every cycle. It is used for bringing the agent back to its home position whenever its value reaches zero, at which point *bucket_was_empty* becomes true. The prediction of the *action(watch_ball)* only reached an accuracy of 73%. Since TILDE is simply a best-effort classification algorithm which separates examples into their respective classes, the authors used CLAUDIEN to help discover the missing rule that explains the discrepancy. CLAUDIEN found these two rules:

- *action(watch_ball)* :- *not(action(none))*, *seeball*, *not(ball_in_my_area)*,

```
seeball ?
+--yes: ball_in_my_area ?
|       +--yes: haveball ?
|       |       +--yes: ball_near_othergoal ?
|       |       |       +--yes: action(shoottogoal) [15 / 15]
|       |       |       +--no:  action(passtobuddy) [122 / 124]
|       |       +--no:  action(movetoball) [1007 / 1015]
|       +--no:  bucket_was_empty ?
|               +--yes: action(moveback) [342 / 347]
|               +--no:  action(watch_ball) [2541 / 3460]
+--no:  action(search_ball) [7770 / 7771]
```

Figure 26: TILDE decision tree on KULRoT [DJJ+98]

not(*bucket_was_empty*).

- *action*(*moveback*) :- *not*(*action*(*none*)), *seeball*, *not*(*ball_in_my_area*), *not*(*bucket_was_empty*).

Notice that both clauses have the same body, which means either actions could be chosen by prediction. The first rule has 73% accuracy, which is the same one found by TILDE. The second rule has an accuracy of 27%, which shows the missing rule. This indicated that the agent would move back to its home location when it was not supposed to, for 27% of the time. Hence, the behavior of the agent can be corrected accordingly.

Since the skills are learned at a high abstraction level, a domain expert is needed to define these high-level predicates based on the low-level primitive ones. Their work focused on the verification of multi-agent systems, and hence there is no actual transfer of prediction knowledge into executable actions. Our goal is to learn with a set of generic spatial predicates, and the rules learned can then be directly used to execute a RoboCup agent, with minimal human intervention or expert knowledge required. Performing executable actions involve converting high-level predicates into low-level primitive actions, which requires the extra

work of learning the parameters associated with each action (eg. power, direction, distance, etc.). Driessens' work avoided this step by only focusing on high-level abstractions.

Perhaps the work that most resembles our own is Lattner et al's([Lat07]). They developed a top-down induction algorithm called *MiTemp* that can discover frequent sequential patterns. The algorithm basically consists of five refinement operations:

1. *l*: Lengthening (Basic pattern generation)
2. *t*: Temporal Restriction
3. *u*: Variable Unification
4. *c*: Class Specialization
5. *i*: Instantiations

The first step generates patterns using combinations of available predicates based on criteria such as minimum frequency of match (ie. support). This is done by combining frequent patterns from the previous refinement level, same as the technique used in Apriori. In this first step, the variables in each literal of the frequent patterns found always start off as instances of the most general class (ie. *object*), since the algorithm mines patterns from the most general to the most specific. The second step is to associate all pairs of literals within each pattern through temporal relationship operators such as those from Allen's interval [All90] or Freksa's semi-interval [FF92]. The third step is to unify variables from different literals if possible, meaning that the literals become more "connected" via the common variables. Class specialization refines frequent patterns by making class

of variables more specific. The last step is to instantiate variables with constants (ie. ground facts) where possible.

While many ILP algorithms, including Apriori and WARMR, treat the pattern size and refinement level as equivalent (ie. the number of predicates allowed in a pattern is the same as the number of refinement levels), *MiTemp* treats the two parameters separately since lengthening is not the only way to refine a pattern in this case. For example, if the maximal pattern size is 3 and maximum refinement level is 10, all patterns up to 3 predicates can still be refined 7 further steps through the other four refinement operations (ie. temporal refinement, unification, class specialization and instantiation). The main algorithm of *MiTemp* is quite similar to WARMR. It searches patterns in a general-to-specific manner, keeping only frequent patterns that meet minimum support. Only frequent patterns within a user-specified size are kept. Like CARMR, constraints are used to enforce ordering so that redundant patterns are not included. The frequent patterns for the current level are then used for next-level candidate generation.

The ultimate goal of Lattner's work is to use the frequent patterns found to predict future unseen situations. After all frequent patterns are collected, each of them is then used to create prediction rules, using techniques similar to Apriori (see section 2.2.5). Like WARMR, CLAUDIEN and other characteristic inductive algorithms, *MiTemp* can generate multiple rules that cover the same examples.

Experiments were performed using real-world matches from five different teams of 2D and 3D RoboCup agents, to generate frequent patterns and prediction rules. In section 3.3.3, a snippet of a 3D match by "MRL vs. ZJU Base" shown in figure 24 was described and explained In this match, there are a total of 2922 *holds* facts. Of all these, only 19 of them are actions (ie. *selfAssist*, *pass* and *failPass*), and all of them are listed in the figure. Notice that some of the actions are not truly consecutive (ie. the ball is not consistently seen from being passed from one player

to another, or being self-assisted), possibly due to inaccuracy during the data extraction step. This means that there could be valid frequent patterns that might not be captured by the algorithm. Figure 27 shows an example of a prediction rule generated from the match.

```
[dist(_h1155,_h1156,far),pass(_h1161,_h1162)] => [dist(_h1155,_h1161,far)]
temp(tr([olderContemp],[olderContemp]),tr([olderContemp]))
conceptRestr(ball,player,distance,player,player,ball,player,distance)

Eval: 0.5848
(f: 0.0119, c: 0 8966, j: 0.0003, s: 1 0000, r: 0.6000, p: 1 0000)
```

Figure 27: Prediction rule generated from "MRU vs. ZJU Base" match [Lat07]

The rule states that if the ball (_h1155) is far from some player (_h1156), then when the ball is passed from a second player (_h1161) to a third player (_h1162), then the second player will be far from the ball. The temporal relations among the predicates are *older and contemporary* in all cases (all 3 combinations from the 3 predicates), based on Freksa's semi-interval [FF92]. As one can see in figure 27, Lattner captured all kinds of prediction rules in his work, not just the ones that are useful for predicting actions (ie. the head of the rule can be any predicates including non-actions).

Lattner also performed experiments that have shown the average accuracy on unseen data decreases monotonically with an increasing refinement level. As expected, he concluded that at higher refinement levels (ie. patterns have more constraints), the prediction rules found are more characteristic for a certain soccer match (data-fitting), rather than general for all matches. Using a small example from a RoboCup match, Lattner showed that *MiTemp* performed more efficiently than WARMR by generating no redundant patterns at all. CARMR, an upgrade of WARMR, is similar to *MiTemp* in this respect that it reduces pattern redundancy.

Currently *MiTemp* is not in the public domain, and attempts to obtain the executable from Lattner have so far failed.

3.4.3 Action Model Learning

Matsui et al. [MIS00] built an Inductive Learning Agent (ILA) by using first-order formalism and inductive logic programming to acquire rules to predict the results of its actions, in an attempt to improve the performance of a simple agent. The agent is made up of an *Observer, Planner , Learner, Checker* and *Actor*, as shown in figure 28. Source code from CMUnited-99 [Uni] was used to implement the low-level functionalities for Observer and Actor to perform sensing and motor functions.

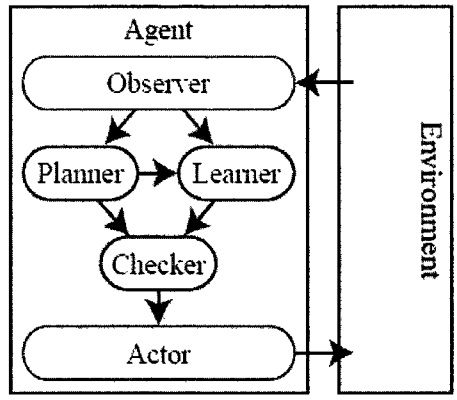

Figure 28: Architecture of ILA [MIS00]

The Learner acquires prediction rules from results of past behavior using the Progol inductive learning algorithm. The Planner provides a few alternatives for actions, and the Checker predicts whether a selected action will succeed or not using prediction rules. If the agent forecasts that a selected action will fail, the agent changes its plan to the next one. Hence the agent tries to choose the best plan to execute according to the Checker's suggestion.

Two simple planners were implemented: one for a shooter and one for a passer. The shooter will try to kick the ball into the goal from anywhere at any time if the ball is reachable (similar to Krislet [Lan]). The passer will try to pass the ball to a teammate when it knows the teammate's location and the ball is within kickable distance.

A teacher's knowledge, expressed in first-order logic consisting of a list of predicates that represents background knowledge in the soccer domain (eg. *my_speed*, *goal_distance*, *ball_moving*, *action_type*, etc), is given to the agent in order to classify examples of past actions into successes or failures. One piece of such teacher knowledge is to classify whether the ball can be kicked to the goal with success or not. Using such classified positive (successes) and negative (failures) examples, the Learner acquired the following Horne clauses that targets the action *kick_to_goal* (*can_kick_to_goal*):

- *can_kick_to_goal*(Frame) :- *my_speed*(Frame,MySpeed), *goal_distance*(Frame,D), *gteq*(MySpeed,0.080), *lteq*(D,18.260).
- *can_kick_to_goal*(Frame) :- *my_speed*(Frame,MySpeed), *goal_distance*(Frame,D), *lteq*(MySpeed,0.030), *lteq*(D,8.180).

Experiment has shown that the success rate of shooting the ball into the goal increased from 36.2% to 90.3% when the agent followed the Checker advice by proceeding with the action only if these prediction rules say they will succeed. A similar procedure was performed on the passer agent and generate clauses for *can_pass_ball*, although the performance only increased slightly from 31.9% to 38.8%, possibly due to the many more variations of the situations the passer agent faces, making the learner more difficult to generalize.

3.4.4 Discussion

As the works in this section show, quantitative observations are often converted into discrete values (eg. *leftof*, *northeast*, *close*) before being taken as inputs by an ILP algorithm. This is usually an important step before knowledge discovery using ILP, since otherwise the sheer amount of quantitative data can lead to search space explosion. Menaka and Santos chose Progol and it could indeed produce accurate results for such classification tasks. Also, since Progol is a covering algorithm, it could find those rules relatively faster than a characteristic induction algorithm. However, rather than pre-processing the data to generate spatial relations as they had done, we decide to perform everything using the ILP algorithm, taking advantage of the ILP capability of including background knowledge in the search. This can lead to a more generic solution applicable to other spatial domains.

Driessens and Lattner chose to use characteristic inductive algorithms such as CLAUDIEN and *MiTemp* (similar to WARMR). These algorithms could be more promising in situations where regular patterns within complex environments can be numerous and are not simply a matter of separating them into appropriate classes. In other words, covering an example only once may not be sufficient, depending on the underlying complexity of the agent and environment. TILDE was also used by Driessens and proved to be inadequate, as CLAUDIEN was able to find an important missing rule not covered by TILDE.

Although both Driessens and Lattner perform learning on RoboCup data using ILP algorithms, their modeling approaches are very different. In Driessens' work, facts are explicitly listed in each time step (figure 22), even though they overlap with the same facts in previous and following time steps. In Lattner's work, facts that are true within consecutive time steps are consolidated into one through the *holds* predicate. Hence the total number of facts to be processed are dramatically

reduced, which can significantly improve search time.

Although the focus in Lattner's works was not solely on mining rules for action predictions, the five refinement operations allow significant pruning of the search space by eliminating redundant patterns. We shall borrow some of these ideas and use it in our work. Lattner only used a minimum confidence of 50%. It will be interesting to see what happens to both accuracy and coverage of the prediction rules if the minimum confidence is increased. We hypothesize that as it increases, the accuracy of the generated rules will increase, but the number of such rules found may also decrease (hence less examples covered) since it is more difficult to find rules that have high confidence. Such experiments are important for behavior imitation as we desire to have as many scenarios (examples) covered as possible, while providing high accuracy. His work also uses Robocup-specific actions such as *pass*, *selfAssist*, etc. which makes the work harder to apply to other domains. Using such high-level actions also allows Lattner to avoid dealing with action parameters such as strengths, directions, etc., which are necessary attributes for low-level behavior imitation.

Matsui's work requires a human expert to classify the target actions, high-level abstractions of low-level primitives, into successes and failures (ie. positive and negative examples), which we want to avoid. Interestingly though, unlike other work involved in ILP learning, they did not attempt to discretize numerical values before learning. They also employed a rich set of predicates, based on functions used in the CMUnited-99 RoboCup agent, in the background knowledge for Progol. These predicates are shown in section 3.3.3. This suggests that Progol is capable of performing search on rules that are numerical in nature with reasonable performance, although the experiments Matsui performed used only 200 odd examples.

3.5 Conclusion

In this section, we looked at different definitions of spatial predicates that have been used across various work. We also studied different frames of reference and representations of spatial relations. Most importantly, we examined existing work that used ILP as a learning tool to discover regular spatial patterns, as a verification tool, and for knowledge discovery. Some techniques of improving ILP search time were also discussed Now, we can apply this knowledge to our own work. Below are the key lessons learned:

- Each of the frames of reference: ego-centric, intrinsic and relative, provides unique information and advantage that can complement one another. We want to study the usefulness of each within the RoboCup domain.
- Using an absolute frame of reference implies implicit assumptions of advance knowledge of the environment boundary, which requires a domain expert to specify. This is something we want to avoid.
- Spatial predicate definitions should be generic and domain-independent, allowing deployment for a wide range of applications. Objects should be predicate arguments for easier generalization.
- We will use the cardinal direction system [CDFH97] for direction relations and distance systems with homogeneous properties [HICF95].
- Numerical data, in most cases, is converted into qualitative, discrete values before learning. This decreases the hypothesis search space significantly.
- Driessens [DJJ+98] showed that TILDE alone (which is a discriminant induction algorithm) is not adequate in finding all frequent patterns. Using a

characteristic algorithm such as WARMR, CLAUDIEN, or MiTemp can fill the gap.

- Progol (a discriminatory induction algorithm) has been a popular choice of ILP algorithm among researchers due to its efficient scheme of bounding both the top and bottom of the hypothesis search space.

- Learning with hierarchical information can help reduce hypothesis search space, thus saves on computation effort.

To the best of our knowledge, all existing work on ILP learning placed most focus on learning high-level cognitive tasks. While that is definitely a worthwhile research area, our current goal is to attempt to study how ILP can perform when learning low-level primitive actions. This involves the additional learning of the parameters associated with each action, making the problem bigger since the hypothesis search space now needs to take into account these action parameters.

From section 2, we learned that ILP algorithms generally belong to two families: discriminant and characteristic induction. In this section, we have seen that Progol (predictive) and WARMR/CARMR (descriptive) have been popular choices among the ILP community. Their operations have been well defined and documented, and as such, we shall also proceed to employ these two algorithms for our comparison purposes.

Chapter 4

Methodology

4.1 Overview

The methodology that this thesis will follow has been used in the field of ILP, specifically in literature where ILP has been used to obtain prediction rules ([Lat07, RK06, DJJ+98], etc.). The steps of the methodology, with specific details for our application, are as follow:

1. *Information Preparation:* The language bias, background knowledge, as well as settings are prepared for the specific ILP algorithm to be employed (see Chapter 5). Raw data logs obtained from RoboCup simulations are saved in a knowledge base after facts in each time cycle are converted into first-order logic models.

2. *ILP Execution:* Execute the ILP algorithms on each selected agent's knowledge base using the prepared files from step (1). Definite clauses (rules) are generated as a result. This step may require multiple iterations in order to tune performance and improve accuracy of the generated rules (see Chapter 6).

3. *Measurements:* The results from each experiment run will be measured against a set of metrics (see section 4.4.1). In addition, they will also be compared

against the results obtained from CBR research previously conducted in our lab [FEL08].

4. *Agent Execution:* The generated rules will be applied to a live agent capable of sending actions to the RoboCup server based on the real-time input stimulus. This step may include post-processing of the rules if necessary (see section 4.5).

4.2 Unit of Analysis

4.2.1 Imitated Agents

For this thesis, we have selected a few RoboCup soccer agents with varying goals and complexity to use to test the ILP algorithms. RoboCup is also a very active research domain, with numerous conferences, work-shops and competitions devoted to it every year. As a result, data and software are often made freely available. This is especially useful in the simulated soccer league, since teams can be downloaded and used for experimental purposes. Our intention is to vary the complexity of the tested agents so that we can identify how the algorithms perform for imitating different complexities of behavior. Due to the vast amount of computation resources required, we could only experiment with three agents. They are listed below:

1. **Sprinter:** runs from one goal net to the other goal net. This agent will simply run laps from one end of the soccer field to the other, completely ignoring the soccer game going on around it. It is included to show the ability to imitate general behavior and to show that the techniques are not over-fitted to imitating soccer players only.

2. **Krislet [Lan]:** follows the ball around the soccer field. If the agent cannot

see the ball it turns until the ball enters its field of vision. The agent then runs toward the ball and then kicks the ball toward the opponents goal when it gets near the ball. Unlike Sprinter, Krislet will actively try to play soccer by scoring goals. We also modified Krislet into Krislet2 which takes intrinsic distance into consideration.

3. **CMUnited [Uni]:** a RoboCup Champion team which uses a layered learning architecture and a number of strategies including formation strategies. CMUnited players can have multiple states of behavior and rely on inter-agent communication, making this team significantly more complex than the other teams that will be examined. We do not believe we will be able to successfully imitate this team since our approach does not currently handle multi-state behavior or use non-visual stimuli like verbal communication. While we do not hope to successfully imitate this team, it provides a benchmark to help guide further research.

4.2.2 Experimental Constants

In this section, we will examine various settings that will be used in the thesis. Because ILP algorithms are basically a search technique, these settings have a direct impact on how the rules are generated, how long it takes to generate them, and whether they are generated at all. Therefore, we shall look at the importance of each of these in more details.

Rule Settings

The most important settings required in using ILP are the spatial predicates employed and the language bias. We explored some existing literature in Section 3

on spatial predicates. A good set of spatial predicates can provide relevant information about the spatial attributes of objects in a clear and concise manner. The way that these predicates are organized in the language bias can allow variables to unify different predicate arguments or generalize them. Having a language bias that is too flexible, however, can induce a huge search space that can take forever to explore.

The language bias and spatial predicates affect the length of the rules being generated (ie. the number of literals in the clause), so it is necessary to understand the former first in order to come up with a reasonable number. If the limit on such length is too small, the search may terminate prematurely and useful rules may never be found. On the other hand if it is too large, it may take forever to finish the search, or the rules found are so long that they are too over-fitting and lose their usefulness. Given the results we find in Section 6.2 which indicates an average rule length between 8 to 12 (based on heuristic search), we believe that a maximum length of 20 literals is reasonable. Experimenting with this parameter is not worth the effort, given the large amount of computing resources already required.

Most ILP algorithms have a parameter called the minimum accuracy, which is equivalent to the minimum confidence level described in Chapter 2. Clauses that do not meet this minimum confidence are rejected. Since we want to have an executable, imitating agent, it is natural for us to choose a high confidence level, so that the rules have high quality. Hence, we set the value to 95%, taking into account noise and inaccuracies inherent to the training data itself. Setting this number to a high value also allows the ILP algorithms to run faster, as a candidate rule can be rejected with fewer matching negative examples. In some circumstances, we may need to lower this value in order to obtain rules at all. We shall point out these cases in the experiments.

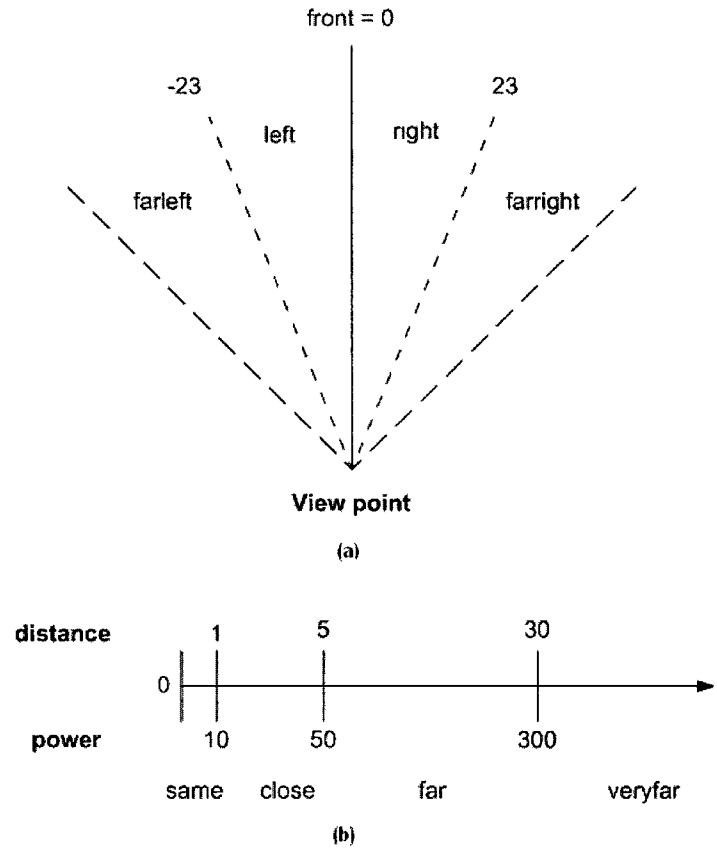

Figure 29: Spatial Relation System: Direction (a) Distance (b)

Spatial Relation System

As mentioned, we need to use spatial relation systems to discretize space so that we can describe it qualitatively, in order to reduce the hypothesis search space. Taking into account that the agent's angle of the field of view is 90 degrees, we modified the 2D-projection direction system [CDFH97] slightly into Figure 29a. We also define our distances using a homogeneous distance system [HICF95] as in Figure 29b.

Unfortunately, the required usage of the 2D-projection direction system is actually a major limitation. It has to do with Krislet's and Sprinter's explicit behavior

in dealing with numerical values of directions. A similar problem with the distance system is that the power parameter must scale linearly with distance. In Section 6.2.3, we shall explain why a cone-shaped cardinal direction system, where the front region does not need to be a discrete value, fails to work. We will also propose some remedies and suggestions for future work.

Action Selection

Multiple generated rules can cover the same example or input stimulus. Lattner [Lat07] evaluates each prediction rule based on metrics like support, confidence, clause length, predicate preference, etc. In this thesis, we sort the rules based on the compression evaluation function from best to worst, taking into account the number of positive and negative examples covered by the rule, along with its length, and then choose the action of the first rule that matches the input pattern. This shall be further discussed in Section 6.2.2.

Summary of Constant Settings

- **Spatial Predicates and Language Bias:** (see Chapter 5.5)
- **Maximum Rule Length:** 20
- **Minimum Rule Accuracy (Confidence):** 95% (Progol and CARMR)
- **Minimum Support:** 2 (Progol and CARMR)
- **Number of examples to process per iteration:** Max # available in log (Progol); 1000 (CARMR)
- **Spatial Relation System:** adaptation of 2D-projection direction system; homogeneous distance system (see Figure 29)

- **Action Selection:** first rule found

4.2.3 Experimental Parameters

Our experiments will vary different parameters in order to address the following research questions, which were already presented in the Introduction:

- *Does the ILP techniques work when imitating a variety of teams?* Our goal is to identify techniques that are applicable to any spatially-aware software agent we wish to imitate. Each experiment will be conducted using data from three RoboCup agents (Section 4.2.1), each with different goals and behavior.

- *What ILP algorithms, among the many available, are the best to perform behavior imitation?* As we have thoroughly described many ILP algorithms in Chapter 2, we will compare the two ILP families using CARMR (characteristic) and Progol (discriminative). There are multitude of options within each algorithm itself, and we shall explore a few of them.

- *Does the use of different frames of reference help in performing better imitation?* RoboCup soccer server provides information of each agent in an egocentric view. We shall investigate whether using relative and intrinsic frame of reference on top will improve imitation, and how it impacts ILP search time.

Summary of Parameters

The following summarizes the parameters that will be varied during experimentation. It should be noted that not all combinations of these parameters will be used. For example, if 5000[1] maximum nodes are found to provide good results, then the subsequent sections will only be using 5000 maximum nodes. The following

[1] 5000 is an arbitrary number picked that provided reasonable accuracy and run-time

parameters apply to Progol only. We shall not vary any parameters for CARMR, except that all three teams shall be used.

- **Algorithm Families:** characteristic (CARMR), discriminative (Progol)
- **Teams:** Sprinter, Krislet, CMUnited
- **Maximum number of nodes to search:** 2500, 5000, 10000
- **Search Options:** breadth-first (BF), depth-first (DF), heuristic
- **Frame of Reference:** Egocentric Only vs. Egocentric + Relative + Intrinsic

4.3 Data Collection

The data collection process involves collecting input stimulus that represent an agent's perception of the environment and its reaction. The collection process is done in an automated manner using a tool [Mar04] that acts as a proxy between the RoboCup player that is to be imitated and the RoboCup soccer server. Instead of connecting directly to the server, the player connects to the proxy utility and the proxy utility connects to the server. This places the tool directly between the player and server. All communication between the player and server is logged before it is forwarded along (Figure 30). The logged data can then be converted into first-order logic model by mapping server messages (representing what the agent can see) with agent messages (the actions that the agent performs) that occur directly after them. All logged data used in this thesis can be found in [NMAa].

For each of the three types of agents being examined (Section 4.2.1), logs were generated by observing a player of that agent type playing 5 complete soccer games. Each game involved 11 players (the maximum allowable size of a soccer team) of one type of agent playing against 11 opposing players. Krislet was selected as

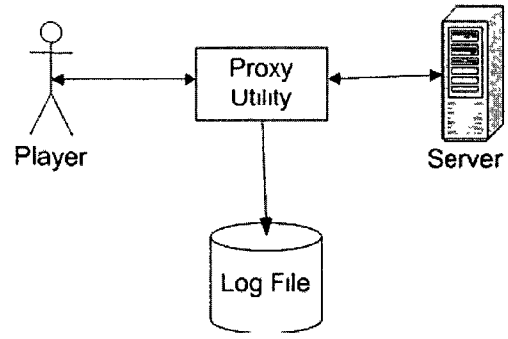

Figure 30: Obtaining an agent log file using a proxy utility [Mar04]

the opposing players, for all three types of agents being logged, in order to keep consistency. Krislet was selected as the opponents because they actively play soccer, unlike Sprinter, and they provide the other team an opportunity to possess the ball during the game. This is in contrast to a team like CMUnited who might be so skilled that the other team would never have a chance to kick the ball.

Unfortunately, the Krislet log files contain very few kick actions as examples. This can affect significantly affect the search process (ie. no example can be found at all). Therefore, even though we are selecting only 5 players, we also need to "borrow" some kick examples from more game plays. There are actually a total of 24 games available from [NMAa].

In summary, the data collection constants are:

- **Collected logs:** 5 games per team
- **Team Size:** 11 players
- **Opponent team:** Krislet

4.4 Data Analysis

Using the 5 players from each agent type, we perform 5-fold cross validation. The knowledge base for each player (per type) will be independently applied to the ILP search, and thus a set of rules will be generated for each player. Then, to perform cross validation, we combine the rules of the 4 other players, and apply them against the knowledge base of the remaining player. This way, any inconsistencies in rule generations (due to noise and erroneous training data) can be smoothed out. The combined rules will likely contain many redundant and overlapping rules, which is the ideal case anyway. This is repeated 5 times for the 5 players, for each agent type, and for each experiment parameter as listed in section 4.2.3. The average values of the computed metrics will be compared and discussed.

4.4.1 Metrics

The metrics collected during experimentation are calculated as follows:

- **Recall:** This is the number of times the action j was correctly selected (T_{pj}), divided by the number of times the action should have been selected (n_j). This is essentially the accuracy of each individual action. Recall values range from 0 (low) to 1 (high).

$$Recall_j = \frac{T_{pj}}{n_j} \tag{1}$$

- **Precision:** This is the number of times the action j was correctly selected (T_{pj}), divided by the number of times the action was actually selected (w_j). This statistic will show if an action is often being incorrectly selected. Precision values range from 0 (low) to 1 (high).

$$Precision_j = \frac{T_{pj}}{w_j} \tag{2}$$

- **f-measure:** The f-measure combines the recall and precision values into a single metric. It ranges from 0 (low) to 1 (high).

$$F_j = \frac{2 \cdot (Precision_j \cdot Recall_j)}{Precision_j + Recall_j} \tag{3}$$

If we combine the f-measure values for all j actions, we get the global f-measure value:

$$F_{global} = \frac{1}{j} \sum_{i=1}^{j} F_i \tag{4}$$

For all experiments performed, the global f-measure value will be the primary metric used for comparison. In addition to these standard metrics, we also include a few others that provide more insight to the rules learned and the algorithms used:

- $Runs_{tot}$: # of successful runs / Total # runs performed. # of successful runs indicates the number of experiments that ran to completion (some experiments never finished in 2 days; some crashed in the middle).
- $Rules_{tot}$: Total # of rules generated in all experiment runs combined.
- *CovRatio*: Total # of positive examples covered / $Rules_{tot}$. This indicates an average number of positive examples in the **test data sets** that are covered by each rule discovered.
- $RuleLen_{avg}$: Average # of literals within each rule.

Last but not least, the following metrics measure the ILP-based imitation system with respect to performance:

- $Time_{avg}$: This is the average amount of time it takes for the ILP algorithm to complete its task of rule generations. In many cases, we had to abort the search before its completion because it had been running for too long (> 12 hours).

- *Imitation Execution Time.* This is the mean amount of time it takes to select an action to perform after receiving input from the RoboCup soccer server. This measures how long the forward-chaining reasoning process took to complete.

$$IET_{mean} = \frac{\sum_{i=1}^{n} IET_i}{n} \tag{5}$$

Where IET_i is the execution time to select an action when given the i^{th} input stimulus, and n is the total number of input stimuli processed.

4.5 Agent Execution

The final step is the agent execution, which is to see how the agent actually performs on the Robocup soccer field, using the generated rules. Its performance is visualized through the Soccer monitor, and qualitative analysis is used to determine whether the agent actually behaves like the one it has learned from.

Chapter 5

ILP Imitation Agent

5.1 Introduction

Based on the state of the art and background information collected in the previous sections, we now present our behavior imitation agent based on ILP. Section 5.2 shall first list the assumptions we made and limitations of the agent. Then, section 5.3 presents an overview of how the agent architecture looks like. Section 5.4 describes the two ILP algorithms that we shall employ for rules generation. Input and output of the ILP algorithms are described in section 5.5 and 5.6, respectively. The inference engine, which receives the generated rules as input as well as stimulus from the RoboCup Soccer Server, is described in section 5.7. Finally, agent execution is described in section 5.8, and conclusion is in section 5.9.

5.2 Limitations and Assumptions

This section lists the assumptions and limitations that we impose on our imitation agent. They allow us to focus on the scope of learning, bound the problem complexity and hence algorithm search time.

- As mentioned before, our learning agent is stateless and reactive in nature. It

has no memory and only takes into account stimulus that is currently available to it. This is the same assumption as the work on CBR [FE08] so that we may compare similar results against each other. Having a stateful agent means that past cases are assumed to be relevant for the current cycle. Tackling such agents would require exploring a much bigger solution space, and we leave it as future work. In the meantime, we focus on imitating reactive agents (except for CMUnited, which is complex in many other ways as well and is only used here to show us how far we are from ultimate success).

- The learning agent supports spatial attributes such as direction and distance between players and objects, in addition to player information such as team and uniform number. However, it ignores face/body/neck directions of individual players, velocity and acceleration of any moving object, stamina of players or any state of game play. Table 11 summaries the objects and their parameters that shall be used for learning in this thesis. It also does not take into account any communications between players and/or coach. We also do not consider the goalie.

- Only a subset of flags are included as part of the hypothesis search space, in order to limit the search time.

- As mentioned before, we do not distinguish between stationary and movable objects for the sake of generality. Thus, flags, lines, players, and balls are all given equal treatment.

- Attributes, such as team name and uniform number, are specific to the RoboCup domain. Other applications can create their own attributes following these as examples.

- Only three actions are being learned, with their associated parameters: *turn*,

dash, and *kick*. The reason is that the work on CBR [FE08] only employed these three actions as well, so comparison data is readily available. Also, the players we are using only employed these actions.

- The knowledge base excludes scenes that do not have an associated action. Many scenes have no associated action not because of dependency on the state of the agent, but instead due to delay in processing the previous command. The same problem was also observed in [DJJ+98].

- Spatial predicates are restricted to binary relations between two objects at most for simplicity and to reduce state-space explosion (ie. no predicate such as *isBetween*(A,B,C)).

- Due to reasons of practicality, searches performed by the ILP algorithms that take too long (> 12 hours) will be terminated because computing resources are limited.

Object	**Parameters**
Ball	distance, direction
Goal	distance, direction, side
Line	distance, direction, identifier
Flag	distance, direction, identifier
Player	distance, direction, team, uniform number

Table 11: Visible objects and their parameters used in this thesis

5.3 System Overview

We now presents our imitation agent system based on ILP, which is shown in Figure 31.

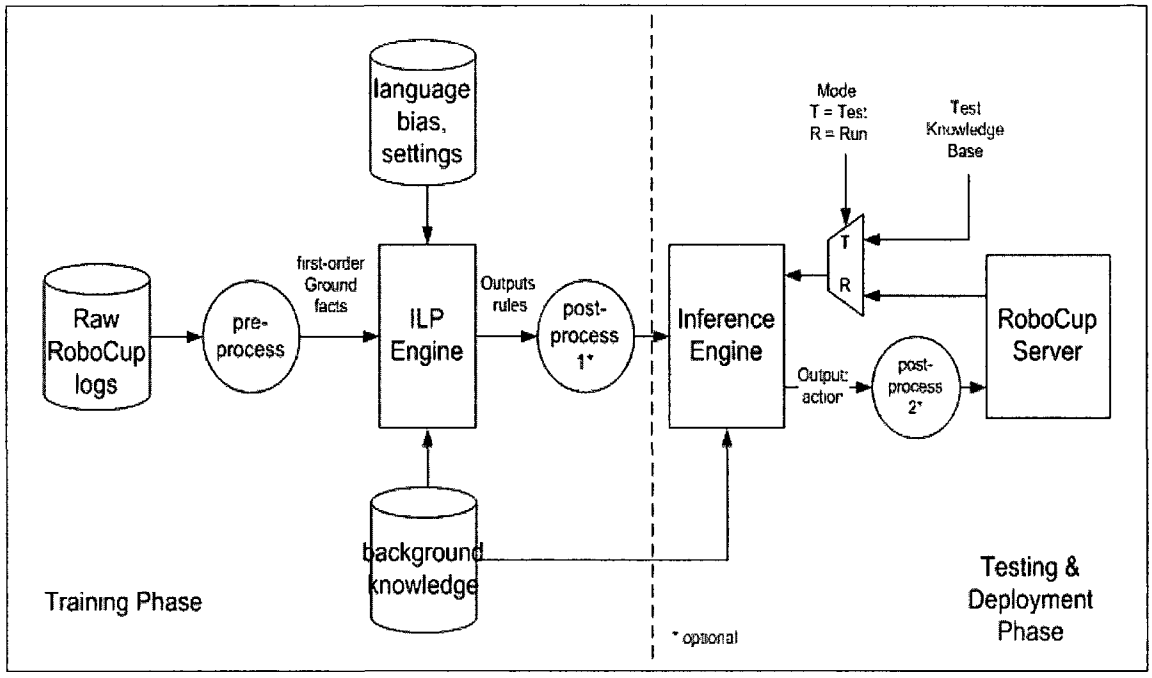

Figure 31: ILP Imitation Agent

The left half of the figure shows the training phase (rule generation), and the right half shows both the testing and deployment phases. First, recorded RoboCup game logs are pre-processed and converted into first-order ground facts, which are accepted by the ILP algorithms. Along with background knowledge, language bias and appropriate settings, the ILP algorithm searches for regular patterns in the data and generates rules. This is the main focus of this thesis. After this, the rules generated may go through a post-processing stage, where the rules can be sorted, modified or eliminated to improve performance when they are used to predict actions in real-time in the deployment phase. The inference engine is used for both testing and deployment purposes. In the testing phase, cross validation uses the inference engine to collect metrics as described in Section 4, and it is again used in the deployment phase, where it accepts input stimulus from the RoboCup Soccer Server, predicts an action based on the generated rules, and outputs it back to the server. The predicted action may also require some post-processing before

being handed off to the server. Notice that the inference engine requires the same background knowledge given to the ILP algorithm in order for it to make correct inferences.

Since we know that most ILP algorithms use first-order logic (or a subset) as their underlying representation language, it follows that our entire imitation system: knowledge base, background knowledge, language bias, and the inference engine is based on first-order logic. This will make the system more coherent, with few conversions of data formats in between.

Next, we shall briefly describe the actual ILP algorithms we use in this thesis, before we look at their inputs and outputs.

5.4 ILP Algorithms

As we have already concluded in the Section 3, we will be using Progol and CARMR as our two choices of ILP systems. The former is a discriminative, covering algorithm while the latter is a characteristic induction algorithm. In reality, we will be using the software package ALEPH [Sri] (A Learning Engine for Proposing Hypotheses), which was first written with the intention to understand Muggleton's inverse entailment [Mug95], the underlying theory behind Progol. Since then, it has evolved to emulate functionality of several ILP systems, such as Progol, FOIL, TILDE, etc. It must be understood that ALEPH only implemented these ILP algorithms at the idea level through multiple configurable parameters. Although it will most likely be more efficient to use CProgol [cpr], we feel that the flexibility in ALEPH in supporting different configurable parameters to emulate multiple ILP algorithms will allow us to tweak the algorithm with higher degree of freedom for our experimentations.

CARMR, as discussed in Section 2, is an improvement over WARMR [Deh99]

by reducing the amount of redundant clauses generated, and uses the concept of delta-free-ness to eliminate clauses that do not meet minimum confidence. The actual software we use is called the CLASSIC'CL [Kar], which is another integrated ILP system capable of modeling algorithms like WARMR, CARMR, CLAUDIEN, ICL, etc. This flexibility is also a reason for us to choose CLASSIC'CL.

Both ALEPH and CLASSIC'CL are in the public domain. They are written in Prolog and can run on the free YAP Prolog Compiler [Wie]. Hence, all input and output of both systems are in the form of standard Prolog first-order predicate logic, which allows us to share the same set of input files on both systems with minor or no modifications. The output rules generated by both systems are also in standard Prolog, so that the inference engine can accept them in the same format. With free source code readily available, this also allows us to understand the algorithms better, debug and modify them as needed.

5.4.1 Basics of ALEPH

Below lists the basic algorithm of ALEPH, which is conceptually the same as the Progol algorithm described in Section 2.2.5.

1. Select example. Select an example to be generalized. If none exist, stop, otherwise proceed to the next step.

2. Build most-specific-clause. Construct the most specific clause that entails the example selected, and is within language restrictions provided. This is usually a definite clause with many literals, and is called the "bottom clause." This step is sometimes called the "saturation" step.

3. Search. Find a clause more general than the bottom clause. This is done by searching for some subset of the literals in the bottom clause that has the "best" score. This step is sometimes called the "reduction" step.

4. Remove redundant. The clause with the best score is added to the current theory, and all examples made redundant are removed. This step is sometimes called the "cover removal" step. Return to Step 1

5.4.2 Basics of CLASSIC'CL

Below lists the basic algorithm of CLASSIC'CL, which is conceptually the same as the WARMR algorithm described in Section 2.2.5.

- FrequentSet-0 = empty
- For each level l (ie. # of literals) from 1 to MaxLevel:
  - Read in frequent set FS of the last level.
  - For each frequent query $Q1$ in FS:
    1. Refine $Q1$ to generate $Q2$.
    2. $Q2$ must not subsume any infrequent query or strong rules found previously. Otherwise, $Q2$ is discarded and return to step 1.
    3. Generate a definite clause from $Q2$, using action has the head. If a strong rule can be found, it is stored in the strong rule list.
    4. Calculate frequency of $Q2$, and store it on FrequentSet-l, or the infrequent list, depending on whether it meets minimum support.

In the next section, we shall look at the input of these two ILP systems.

5.5 Input of ILP Algorithm

5.5.1 Introduction

In this section, we examine the input required by our two ILP systems. Since they both run on the same YAP Prolog Compiler, many of these input files can be shared with little or no modifications. Section 5.5.2 shall discuss the knowledge base which contains the ground facts generated from a RoboCup game play. This is used as the training data set. Section 5.5.3 shall discuss the background knowledge which contains the clauses that help define the language bias, and are necessary for the ILP algorithms to resolve literals that are not part of the ground facts. Lastly Section 5.5.4 shall discuss the language bias that defines the hypothesis search space.

5.5.2 Knowledge Base

The object information sent from the soccer server every time cycle is presented in several formats as described by *ObjInfo* back in Figure 1. A typical message from the RoboCup Soccer Server looks like one shown in Figure 32.

The pre-processing stage shown in Figure 31 is written as a Java procedure using JavaCC [jav], a popular parser generator for use with Java applications. This procedure converts incoming messages into first-order ground facts. Here, we look at two alternatives for representing *ObjInfo* using first-order logic below:

1. Use one predicate per object type to cover all possible attributes of an object.

2. Use one predicate to represent one attribute of an object.

In the first option, we use the fact that Prolog allows multiple predicate definitions of the same name with different arities. Figure 33 shows how the first

```
(see 67
((flag c) 14 7 0 -0 294 -0) ((flag c b) 48 4 -9) ((flag l b) 73 7 35)
((flag p l b) 52 5 35) ((ball) 14 9 0 -0 298 -0) ((player Opponents) 30 17)
((player) 60 3 37) ((player Opponents 3) 27 1 39 -0 0 4)
((player Opponents) 22 2 3) ((player Opponents) 36 6 4)
((player Opponents) 40 4 22) ((player Opponents) 36 6 -2)
((player Opponents 8) 22 2 15 -0 444 0 2) ((player) 66 7 34)
((player Opponents) 40 4 10) ((player) 54 6 33)
((player Krislet 1) 9 -35 -0 18 -1 1) ((line b) 49 9 76))
(dash 149 0)

(see 92 ((flag l t) 60 3 14) ((flag g l b) 56 8 -26) ((goal l) 55 1 -19) ((flag g l t) 54 6 -12)
((flag p l c) 39 3 -23) ((flag p l t) 39 6 5) ((player Opponents 3) 20 1 -41 -0 402 -0 4) ((line l) 55 7 79))
(turn 40 0)
```

Figure 32: Typical Messages from RoboCup server

option looks like. This representation has been used by [DJJ+98]. The argument *cycle* indicates the exact time cycle at which the particular object is being seen Although this representation is quite straight forward, it is hard for human to read when they are instantiated as ground facts (ie. must correlate each argument to the original predicate definition). In addition, a predicate definition must be changed if attributes need to be added to or removed from a specific object.

```
player(cycle, playerId, teamId, uniformNum, dist, dir)
player(cycle, playerId, teamId, uniformNum, dir)
ball(cycle, ballId, dist, dir)
ball(cycle, ballId, dir)
line(cycle, lineId, dist, dir)
line(cycle, lineId, dir)
flag(cycle, flagId, dist, dir)
flag(cycle, flagId, dir)
goal(cycle, goalId, side, dist, dir)
goal(cycle, goalId, side, dir)
```

Figure 33: Each object represented by one predicate

In the second option, each object is still identified by an individual predicate, but their attributes are individually identified apart from the object predicates. Figure 34 shows how the representation looks like. It is very similar to the one used by Lattner [Lat07]. Each concept (attribute) is explicitly stated, which allows for finer-grained knowledge representation and reasoning. Non-spatial attributes, such as *inTeam*, *hasUnum*, are separated from the spatial attributes. The two spatial predicates, *dirQn* and *dirQn*, carry the direction and distance values between the two objects *object1Id* and *object2Id*. They are used in representing all frames of reference: ego-centric, relative and intrinsic, as we shall see in Section 5.5.3. The suffix "*Qn*" stands for "quantitative" to indicate the predicate carries numerical values instead of qualitative ones, which would be suffixed by "*Ql*".

The non-spatial attributes are domain-dependent, while the spatial attributes are only spatial domain-dependent, and are useful for applications other than RoboCup. Furthermore, the spatial predicates can apply to different objects (as indicated by *objectId*), which allows for a more concise language bias and for generalizations to be performed across different object types with the same spatial attribute. Having attributes represented separately from the object predicates also allow missing attributes to be nicely ignored. For example, if a player is too far to identify its uniform number, the predicate *hasUnum*/2 would simply be absent from the ground facts, whereas in the first option, the predicate *player*/6 would require a special *null* term, or something similar, to indicate that the information is missing. Lastly, this representation also allows for object attributes to be added and removed easily. Hence, we decide to choose the second option.

Next, we need to look at how an action is represented in first-order logic Figure 35 shows the actions and their associated parameters that our imitating agent will be learning. Again, we examine two alternatives.

player(playerId)
ball(ballId)
goal(goalId)
team(teamId)

inTeam(playerId, teamId)
hasUnum(playerId, uniformNum)

objectSeen(cycle, object1Id)
objectSeen(cycle, object2Id)
distQn(cycle, object1Id, object2Id, distanceValue)
dirQn(cycle, object1Id, object2Id, directionValue)

Figure 34: Each attribute represented by one predicate

1. Use one predicate per action type, with the associated parameters as arguments (Figure 36)

2. Use a predicate *actionQn* to represent all action types, with different arities to support different number of parameters (Figure 37)

- (*dash* Power)
  Power .:= 0 to 600
- (*kick* Power, Direction)
  Power ::= 0 to 600
  Direction . = -180 to 180 degrees
- (*turn* Moment)
  Moment ·= -180 to 180 degrees

Figure 35: RoboCup Actions to learn with their parameters

In the first option, the action type is explicitly indicated through the predicate name. This means for new actions, new predicates must be created. In the second option, we again use the fact that Prolog supports multiple predicate definitions of the same name with different arities. This option uses the same *action* predicate to explicitly represent the different types of actions, which makes it more general to

other types of applications. More importantly, as we shall see in Section 5.7, using a single action predicate makes it very simple for the inference engine to work.

turn(cycle, moment)
dash(cycle, power).
kick(cycle, power, direction).

Figure 36: One predicate per Action Type

actionQn(cycle, turn, moment)
actionQn(cycle, dash, power)
actionQn(cycle, kick, power, direction).

Figure 37: One predicate for all Action Types

Formal Snapshot Definition

We now combine all the ideas described previously and give the definition of a snapshot using Backus-Naur Form in Table 12. The predicate $key/1$ is required by CARMR only, and shall be discussed in Section 6.3. As mentioned earlier, only 7 flags are used to limit the search space. It is also worth mentioning that we follow the way Lattner [Lat07] represents players, where $p1$ represents a teammate with uniform #1, $q2$ represents an opponent with uniform #2, and $u\#$ represents an unknown player whose uniform number or belonging team or both cannot be identified. The index is simply used to distinguish multiple unknown players in the current cycle, if any. The player $self$ represents the observing agent itself, so it implies the direction or distance is represented in the ego-centric frame of reference. Figure 38 shows an example of the generated ground facts that represent the two cycles shown in Figure 32

Now that we understand how the knowledge base is represented, the next section talks about Background Knowledge, which is an important component of the ILP search.

SnapShot	→	*Sentence*
Sentence	→	*key(Cycle)* | *AtomicSentence* | *Sentence* | *ActionInfo*
AtomicSentence	→	*PlayerInfo* | *BallInfo* | *GoalInfo* | *FlagInfo* | *LineInfo*
ObjectInfo	→	*objectSeen(Cycle, ObjTerm)* ∧ *distQn(Cycle, ObjTerm, ObjTerm, Distance)* ∧ *dirQn(Cycle, ObjTerm, ObjTerm, Direction)*
	|	*objectSeen(Cycle, ObjTerm)* ∧ *dirQn(Cycle, ObjTerm, ObjTerm, Direction)*
ObjTerm	→	*PlayerTerm* | *BallTerm* | *GoalTerm* | *FlagTerm* | *LineTerm*
PlayerTerm	→	*self* | *p1* | . | *p11* | *q1* | | *q11* | | *u1* | . | *u22*
BallTerm	→	*ball1*
GoalTerm	→	*gl* | *gr*
FlagTerm	→	*fc* | *flt* | *flb* | *fct* | *fcb* | *frt* | *frb*
LineTerm	→	*ll* | *lr* | *lt* | *lb*
ActionInfo[1]	→	*actionQn(Cycle, turn, Direction)*
	|	*actionQn(Cycle, dash, Power)*
	|	*actionQn(Cycle, kick, Power, Direction)*
Cycle	→	0 < *maxcycle* >
Direction	→	-180 180
Distance	→	0 60
Power	→	0 600

Table 12: Definition of a Snapshot in Backus-Naur Form

5.5.3 Background Knowledge

Introduction

The Background Knowledge (BG) consists of many ground facts and clauses that make up the language search space that the ILP algorithms search upon. Since BG is declared in standard Prolog in our case (and in most cases), it becomes a powerful medium through which new features can be extracted from the data and be placed as part of the hypothesis search space, all within the ILP paradigm without requiring the help of external pre-processors. This allows the entire process to be more seamless, and much more portable to other applications or domains. In our case, we have used BG to declare clauses that discretize direction and distance into

```
key(67)
objectSeen(67,fc) dirQn(67,self,fc,0 0) distQn(67,self,fc,14 7)
objectSeen(67,fcb) dirQn(67,self,fcb,-9 0) distQn(67,self,fcb,48 4)
objectSeen(67,flb) dirQn(67,self,flb,35 0) distQn(67,self,flb,73 7)
objectSeen(67,ball1) dirQn(67,self,ball1,0 0) distQn(67,self,ball1,14 9)
objectSeen(67,u1) dirQn(67,self,u1,17 0) distQn(67,self,u1,30 0)
objectSeen(67,u2) dirQn(67,self,u2,37 0) distQn(67,self,u2,60 3)
objectSeen(67,q3) dirQn(67,self,q3,39 0) distQn(67,self,q3,27 1)
objectSeen(67,u3) dirQn(67,self,u3,3 0) distQn(67,self,u3,22 2)

objectSeen(67,p1) dirQn(67,self,p1,-35 0) distQn(67,self,p1,9 0)
objectSeen(67,lb) dirQn(67,self,lb,76 0) distQn(67,self,lb,49 9)
actionQn(67,dash,149 0)

key(92)
objectSeen(92,flt) dirQn(92,self,flt,14 0) distQn(92,self,flt,60 3)
objectSeen(92,gl) dirQn(92,self,gl,-19 0) distQn(92,self,gl,55 1)
objectSeen(92,q3) dirQn(92,self,q3,-41 0) distQn(92,self,q3,20 1)
objectSeen(92,ll) dirQn(92,self,ll,79 0) distQn(92,self,ll,55 7)
actionQn(92,turn,40 0)
```

Figure 38: Examples of generated Ground Facts from Figure 32

distinct regions, and more interestingly, to derive values for relative and intrinsic frames of reference from the default ego-centric frame. We can envision future extensions where background clauses can be created to predict future location of a moving object based on its current location, velocity and acceleration.

We have divided BG into several categories. First, some basic definitions of objects are given. Second, the class hierarchy of these objects is declared. Third, region discretization is presented. Fourth, we will show some miscellaneous clauses that assist in shaping the language bias. Last but not least, derivation of the relative and intrinsic frames of reference is presented.

Basic Definitions

Figure 39 shows the basic object definitions that will be used throughout this thesis. The importance of such declarations is that anything else not declared as facts will not become part of the search space. This is because Prolog uses backtracking to look for all answers that can satisfy a query. Obviously, anything that is not declared as fact cannot be found.

It is obvious that these basic definitions are RoboCup-specific. Other applications can declare their own object and attribute predicates as desired in the background knowledge. However, the spatial predicates, as we shall see, can remain the same across different spatial domains. Note that as mentioned before, the unknown players are the ones whose uniform number or belonging team cannot be identified. Thus the predicate *hasUnum*/2 does not apply to them.

Object Class Hierarchy

Figure 40 shows clauses that define the object class hierarchy. This background knowledge is reused from Lattner [Lat07] and mimics the idea of SPADA [ML01]. The idea is that each of the objects, *ball*, *player*, *goal*, etc., is an *instanceOf object*. As we shall see in Section 5.5.4, *instanceOf* is part of the language search space. The idea is that an object variable X can be used in either *instanceOf*(X, player), for example, or *instanceOf*(X, object), with the latter being more general. Hence, this allows different types of objects to be unified to the same general variable X, which permits a concise yet flexible language. Currently, there are only two layers in the hierarchy. More can be added as needed.

Another important usage of the *instanceOf* predicate is for objects that have multiple instances such as players, goals, lines and flags, a single literal *instanceOf*(X, player) can represent all the individual players, without unnecessarily referring to

```
team(team1) team(team2) team(unkn)

player(self)
player(p1) hasUnum(p1,1) inteam(p1,team1)
player(p2) hasUnum(p2,2) inteam(p2,team1)
player(p3) hasUnum(p3,3) inteam(p3,team1)
player(p4) hasUnum(p4,4) inteam(p4,team1)
player(p5) hasUnum(p5,5) inteam(p5,team1)
player(p6) hasUnum(p6,6) inteam(p6,team1)
player(p7) hasUnum(p7,7) inteam(p7,team1)
player(p8) hasUnum(p8,8) inteam(p8,team1)
player(p9) hasUnum(p9,9) inteam(p9,team1)
player(p10) hasUnum(p10,10) inteam(p10,team1)
player(p11) hasUnum(p11,11) inteam(p11,team1)
player(q1) hasUnum(q1,1) inteam(q1,team2)
player(q2) hasUnum(q2,2) inteam(q2,team2)
player(q3) hasUnum(q3,3) inteam(q3,team2)
player(q4) hasUnum(q4,4) inteam(q4,team2)
player(q5) hasUnum(q5,5) inteam(q5,team2)
player(q6) hasUnum(q6,6) inteam(q6,team2)
player(q7) hasUnum(q7,7) inteam(q7,team2)
player(q8) hasUnum(q8,8) inteam(q8,team2)
player(q9) hasUnum(q9,9) inteam(q9,team2)
player(q10) hasUnum(q10,10) inteam(q10,team2)
player(q11) hasUnum(q11,11) inteam(q11,team2)
player(u1) player(u2) player(u3) player(u4) player(u5) player(u6) player(u7) player(u8)
player(u9) player(u10) player(u11) player(u12) player(u13) player(u14) player(u15)
player(u16) player(u17) player(u18) player(u19) player(u20) player(u21) player(u22)
inteam(u1,unkn) inteam(u2,unkn) inteam(u3,unkn) inteam(u4,unkn) inteam(u5,unkn) inteam(u6,unkn)
inteam(u7,unkn) inteam(u8,unkn) inteam(u9,unkn) inteam(u10,unkn) inteam(u11,unkn) inteam(u12,unkn)
inteam(u13,unkn) inteam(u14,unkn) inteam(u15,unkn) inteam(u16,unkn) inteam(u17,unkn) inteam(u18,unkn)
inteam(u19,unkn) inteam(u20,unkn) inteam(u21,unkn) inteam(u22,unkn)

ball(ball1)
goal(gl) goal(gr)
line(ll) line(lr) line(lt) line(lb)
flag(fc) flag(flt) flag(flb) flag(fct) flag(fcb) flag(frt) flag(frb)
```

Figure 39: Basic Definitions

a specific instance of players. This allows the search to be much more general in nature, and prevents over-fitting.

```
isA(ball, object)
isA(player, object)
isA(goal, object)
isA(flag, object)
isA(line, object)

directInstanceOf(X,ball) - ball(X)
directInstanceOf(X,player) - player(X)
directInstanceOf(X,goal) - goal(X)
directInstanceOf(X,line) - line(X)
directInstanceOf(X,flag) - flag(X)

subConceptOf(X,Y) - isA(X,Y)
subConceptOf(X,Y) - isA(Z,Y), subConceptOf(X,Z)
instanceOf(Inst,Concept) - directInstanceOf(Inst,Concept)
instanceOf(Inst,Concept) - directInstanceOf(Inst,SubConcept), subConceptOf(SubConcept,Concept)
```

Figure 40: Object Class Hierarchy

Region Discretization

Figure 41 shows how region discretization is achieved. As mentioned in Section 4, direction is discretized into 5 regions and distance is discretized into 4. Basically, the predicates *dirQn*/4 and *distQn*/4 are translated into *dirQl*/4 and *distQl*/4 respectively. The discretization values are hard-coded as described back in Figure 29. Similarly, *actionQn*/3 and *actionQn*/4 predicates are translated into a single *action*/4 predicate as shown in Figure 42. Using a single action predicate here simplifies the language bias (see Section 5.5.4), and makes it simple for the inference engine to work (see Section 5.7).

Note that these particular discretization values were picked because we have advance knowledge that these values are part of the intrinsic behavior of Krislet and Sprinter. This is a major limitation of our current approach which we will

further explain in Section 6.2.3.

```
dir(farleft) dir(left) dir(front) dir(right) dir(farright)
dist(same) dist(close) dist(far) dist(veryfar)

dir_farleft(A) - A ≤ -23 0
dir_left(A) - A > -23 0, A < 0 0
dir_front(A) - A == 0 0
dir_right(A) - A > 0 0, A < 23 0
dir_farright(A) - A ≥ 23 0

dist_same(A) - A < 1 0
dist_close(A) - A ≥ 1 0, A < 5 0
dist_far(A) - A ≥ 5 0, A < 30 0
dist_veryfar(A) - A ≥ 30 0

power_same(A) - A < 10 0
power_close(A) - A ≥ 10 0, A < 50 0
power_far(A) - A ≥ 50 0, A < 300 0
power_veryfar(A) - A ≥ 300 0

dirQl(C,self,X,farleft) - key(C), objectSeen(C,X), instanceOf(X,object), dirQn(C,self,X,D), dir_farleft(D)
dirQl(C,self,X,left) - key(C), objectSeen(C,X), instanceOf(X,object), dirQn(C,self,X,D), dir_left(D)
dirQl(C,self,X,front) - key(C), objectSeen(C,X), instanceOf(X,object), dirQn(C,self,X,D), dir_front(D)
dirQl(C,self,X,right) - key(C), objectSeen(C,X), instanceOf(X,object), dirQn(C,self,X,D), dir_right(D)
dirQl(C,self,X,farright) - key(C), objectSeen(C,X), instanceOf(X,object), dirQn(C,self,X,D), dir_farright(D)

distQl(C,self,X,same) - key(C), objectSeen(C,X), instanceOf(X,object), distQn(C,self,X,D), dist_same(D)
distQl(C,self,X,close) - key(C), objectSeen(C,X), instanceOf(X,object), distQn(C,self,X,D), dist_close(D)
distQl(C,self,X,far) - key(C), objectSeen(C,X), instanceOf(X,object), distQn(C,self,X,D), dist_far(D)
distQl(C,self,X,veryfar) - key(C), objectSeen(C,X), instanceOf(X,object), distQn(C,self,X,D), dist_veryfar(D)
```

Figure 41: Region Discretization

Other Assisting Clauses

Figure 43 shows several clauses that are part of the language bias. For example, the predicate *not_objectSeen*/2 is an important predicate that allows objects that are not present in the current snapshot to be searched. Obviously, this can significantly increase the search space since a lot of objects can be absent from a particular snapshot. We shall see how we mitigate this in Section 5.5.4.

The three predicates, *eq_obj_const*/2, *eq_obj_dirQl_O*3, and *eq_obj_distQl_O*3, are also reused from Lattner [Lat07]. Basically, they are used to bind a variable X to a particular constant, be it an object, a direction or a distance. For example,

```
action(C,turn,farleft,same) - key(C), actionQn(C,turn,D), dir_farleft(D)
action(C,turn,left,same) - key(C), actionQn(C,turn,D), dir_left(D)
action(C,turn,right,same) - key(C), actionQn(C,turn,D), dir_right(D)
action(C,turn,farright,same) - key(C), actionQn(C,turn,D), dir_farright(D)

action(C,dash,fw,same) - key(C), actionQn(C,dash,P), power_same(P)
action(C,dash,fw,close) - key(C), actionQn(C,dash,P), power_close(P)
action(C,dash,fw,far) - key(C), actionQn(C,dash,P), power_far(P)
action(C,dash,fw,veryfar) - key(C), actionQn(C,dash,P), power_veryfar(P)

action(C,kick,farleft,same) - key(C), actionQn(C,kick,P,D), power_same(P), dir_farleft(D)
action(C,kick,left,same) - key(C), actionQn(C,kick,P,D), power_same(P), dir_left(D)
action(C,kick,front,same) - key(C), actionQn(C,kick,P,D), power_same(P), dir_front(D)
action(C,kick,right,same) - key(C), actionQn(C,kick,P,D), power_same(P), dir_right(D)
action(C,kick,farright,same) - key(C), actionQn(C,kick,P,D), power_same(P), dir_farright(D)

action(C,kick,farleft,close) - key(C), actionQn(C,kick,P,D), power_close(P), dir_farleft(D)
action(C,kick,left,close) - key(C), actionQn(C,kick,P,D), power_close(P), dir_left(D)
action(C,kick,front,close) - key(C), actionQn(C,kick,P,D), power_close(P), dir_front(D)
action(C,kick,right,close) - key(C), actionQn(C,kick,P,D), power_close(P), dir_right(D)
action(C,kick,farright,close) - key(C), actionQn(C,kick,P,D), power_close(P), dir_farright(D)

action(C,kick,farleft,far) - key(C), actionQn(C,kick,P,D), power_far(P), dir_farleft(D)
action(C,kick,left,far) - key(C), actionQn(C,kick,P,D), power_far(P), dir_left(D)
action(C,kick,front,far) - key(C), actionQn(C,kick,P,D), power_far(P), dir_front(D)
action(C,kick,right,far) - key(C), actionQn(C,kick,P,D), power_far(P), dir_right(D)
action(C,kick,farright,far) - key(C), actionQn(C,kick,P,D), power_far(P), dir_farright(D)

action(C,kick,farleft,veryfar) - key(C), actionQn(C,kick,P,D), power_veryfar(P), dir_farleft(D)
action(C,kick,left,veryfar) - key(C), actionQn(C,kick,P,D), power_veryfar(P), dir_left(D)
action(C,kick,front,veryfar) - key(C), actionQn(C,kick,P,D), power_veryfar(P), dir_front(D)
action(C,kick,right,veryfar) - key(C), actionQn(C,kick,P,D), power_veryfar(P), dir_right(D)
action(C,kick,farright,veryfar) - key(C), actionQn(C,kick,P,D), power_veryfar(P), dir_farright(D)
```

Figure 42: Qualitative Action Predicates

dirQn(C,self,X,D), *eq_obj_const*(X, p1) binds the output variable *X* of *dirQn*/4 to the constant *p*1. We shall see more on this in Section 5.5.4. Notice that the predicate *allowableObj*/1 prevents a variable to be instantiated to an unknown player (eg. *u*1). This is because as we have mentioned, the index of an unknown player has no meaning in itself. However, the literal *inteam*(X,unkn) is still allowed.

```
not_objectSeen(C,X) - key(C), instanceOf(X,object), \+ objectSeen(C,X), X \= self

allowableObj(X) - instanceOf(X,object), \+ (player(X), inteam(X,unkn))

eq_obj_const(X, C) - allowableObj(C), X = C
eq_obj_dirQl_O3(X, C) - dir(C), X = C
eq_obj_distQl_O3(X, C) - dist(C), X = C
Note "\+" means "not"
```

Figure 43: Other Assisting Clauses

Relative and Intrinsic Frame of Reference

Figure 44 shows all the clauses necessary to derive the relative and intrinsic frame of reference. For simplicity purpose, we only derive the intrinsic distance between two objects, and the relative angle between the two objects from the observing agent in the relative frame of reference.

First, we define a predicate *two_objects_seen* that is true if two different and allowable objects are present in the current cycle. Second, given N objects, there are $\frac{N(N+1)}{2}$ pairs of relations. We order each pair of object so that the pair is only processed if object X1 is closer to the observing agent than object X2 in the predicate *processObjects*/5. By ordering them the search space is cut in half without losing any information.

For the relative frame of reference, we simply need to calculate the delta angle between the two objects. For the intrinsic distance, we calculate the square distance between the two objects. What is particularly elegant here is that Prolog allows us to use the same predicates *dirQn*/4, *distQn*/4, *dirQl*/4 and *distQl*/4 and to add new

definitions on top, where the variable X1 now does not equal to *self*. This allows a very concise yet sufficient language bias using just a small set of spatial predicates

```
two_objects_seen(C,X1,X2) -
                objectSeen(C,X1), allowableObj(X1), objectSeen(C,X2), allowableObj(X2),
                X1 \= self, X2 \= self, X1 \= X2

% This ensures the pair is only processed once by using the closer object
processObjects(C,X1,X2,D1,D2) -
              distQn(C,self,X1,D1), distQn(C,self,X2,D2), D1 =< D2

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Relative Frame of Reference

dirQn(C,X1,X2,Angle) -
      processObjects(C,X1,X2,D1,D2),
      dirQn(C,self,X1,A1), dirQn(C,self,X2,A2),
      Angle is (A2 - A1)

dirQl(C,X1,X2,farleft)    - key(C), two_objects_seen(C,X1,X2), dirQn(C,X1,X2,A), dir_farleft(A)
dirQl(C,X1,X2,left)       - key(C), two_objects_seen(C,X1,X2), dirQn(C,X1,X2,A), dir_left(A)
dirQl(C,X1,X2,front)      - key(C), two_objects_seen(C,X1,X2), dirQn(C,X1,X2,A), dir_front(A)
dirQl(C,X1,X2,right)      - key(C), two_objects_seen(C,X1,X2), dirQn(C,X1,X2,A), dir_right(A)
dirQl(C,X1,X2,farright)   - key(C), two_objects_seen(C,X1,X2), dirQn(C,X1,X2,A), dir_farright(A)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Intrinsic Frame of Reference (Interobject Distance)

% Law of Cosines for Euclidean Geometry
sq_distance(D1,D2,A1,A2,Total) -
           Gamma is abs(A1-A2),
           Total is sqrt(D1*D1 + D2*D2 - 2*D1*D2*cos(Gamma))

distQn(C,X1,X2,Dist) -
       processObjects(C,X1,X2,D1,D2),
       dirQn(C,self,X1,A1), dirQn(C,self,X2,A2),
       sq_distance(D1,D2,A1,A2,Dist)

distQl(C,X1,X2,same)      - key(C), two_objects_seen(C,X1,X2), distQn(C,X1,X2,D), dist_same(D)
distQl(C,X1,X2,close)     - key(C), two_objects_seen(C,X1,X2), distQn(C,X1,X2,D), dist_close(D)
distQl(C,X1,X2,far)      .- key(C), two_objects_seen(C,X1,X2), distQn(C,X1,X2,D), dist_far(D)
distQl(C,X1,X2,veryfar)   - key(C), two_objects_seen(C,X1,X2), distQn(C,X1,X2,D), dist_veryfar(D)
```

Figure 44: Derivation of Relative and Intrinsic Reference

Generating Positive and Negative Examples

A necessary input for ALEPH is positive and negative example files We need to generate these files for each action (along with its qualitative parameters). Table 13 shows an example for the action *turn.left*. The only difference between the two

files are the cycle numbers. The left side shows all the ground facts that are true for those particular time cycles, the right side those that are false. These two files are then used by ALEPH for learning the action *turn.left*. A simple Prolog program *imisys_gentest.pl* [NMAb] was written to generate these two files.

PositiveExamplesFile	*NegativeExamplesFile*
action(38,turn,left,same)	*action*(0,turn,left,same)
action(65,turn,left,same)	*action*(1,turn,left,same)
action(68,turn,left,same)	*action*(2,turn,left,same)
action(74,turn,left,same)	*action*(5,turn,left,same)

Table 13: Positive and Negative Examples Files

Now that we have seen how the Background Knowledge is used, it is time to put together everything to see how the language bias is created.

5.5.4 Language Bias

The language bias is the most important component in ILP search. It defines the boundary of the search space, confining how the search should be performed. It consists of templates of predicate definitions, which indicate what predicates can occur within a candidate clause, how many times each predicate can appear, and in what manner the predicate arguments should be used: whether they are treated as input(+), output(-), or constant(#). Section 2.2.4 explained these in more details.

Figure 45 shows how the language bias is declared for RoboCup, specifically for ALEPH. Conceptually, the language bias is very similar to CARMR, except for a few syntactic differences (see Section 6.3.3). It starts off with the *modeh* predicate that defines the head of the definite clause. Obviously, it is the *action* predicate in our case, along with its parameters. Declaring the action parameters as constants allows the search to focus on the single action, along with its fixed set of parameters, one at a time. This allows us to run searches for multiple actions/parameters in

parallel on multiple machines to speed up the process several folds. One may notice that the parameters #dir/#dist can be declared as variables instead. The problem with that is Progol will only be able to find rules where the body contains literals that bind with these variables. A valid rule that has no such binding will not be found. However, we understand that parameter binding is an important and useful feature (eg. angle-to-turn-moment mapping). Therefore, we propose that variable binding can be done as a post-processing step easily by replacing all common constants in a rule with the same variable. However, we shall leave this as future work.

```
- modeh(1,action(+k,#action,#dir,#dist))

- modeb(1,not_objectSeen(+k,-object))
- modeb(*,dirQl(+k,-object,-object,-dir))
- modeb(*,distQl(+k,-object,-object,-dist))

- modeb(*,instanceOf(+object,#concept))
- modeb(*,eq_obj_const(+object,#object))
- modeb(*,eq_obj_dirQl_O3(+dir,#dir))
- modeb(*,eq_obj_distQl_O3(+dist,#dist))

- modeb(*,inteam(+object,#team))
- modeb(*,hasUnum(+object,#unum))

- determination(action/4, not_objectSeen/2)
- determination(action/4, dirQl/4)
- determination(action/4, distQl/4)
- determination(action/4, instanceOf/2)
- determination(action/4, eq_obj_const/2)
- determination(action/4, eq_obj_dirQl_O3/2)
- determination(action/4, eq_obj_distQl_O3/2)
- determination(action/4, inteam/2)
- determination(action/4, hasUnum/2)
```

Figure 45: Language Bias for RoboCup

For the body clause, the predicates *not_objectSeen*/2, *dirQl*/4 and *distQl*/4 declare output variables, which allows them to be unified by any of these predicates:

instanceOf/2, *eq_obj_const*/2, *inteam*/2, and *hasUnum*/2. This provides flexibility to the language since if the parameters are constants instead of variables, they will have to be bound to a particular constant. In Section 6, where we present experimental results, we compare some results when these parameters are declared either as variable or constants.

The '*' at the first argument indicates that the predicate can appear in the candidate clause any number of times, except for *not_objectSeen*/2. Hence, the predicates *dirQl*/4 and *distQl*/4 can appear once per object that appears in a snapshot. As discussed previously, the predicate *not_objectSeen*/2 indicates any object that is absent from the current snapshot, which can significantly increase the search space In order to place a limit on it, we restrict that only one *not_objectSeen*/2 can appear in any clause. This is based on the assumption that humans cognitively rely more on the knowledge of objects that are seen to make decisions, rather than those are not.

The declarations using the *determination*/2 predicate are required simply to let ALEPH know the second argument is to be included in the search for the head specified in the first argument. Their order has no significance here.

Search Space Size

In order to appreciate the size of the problem we are facing, we can perform a naive estimation of the search space size. In simple terms, let's assume we are only declaring *dirQl*/4, *eq_obj_const*/2 and *eq_obj_dirQl_O3* with the *determination*/2 predicates, and we shall use a maximum clause length of 20 as specified in Section 4. We also assume for now B1 = *self* (ego-centric view only). The search space lattice (assuming no pruning or any optimization) would look something like the one shown in Figure 46.

Of course, there are many intermediate refinement candidates skipped, but the figure serves the purpose of showing the exponential growth of the search space.

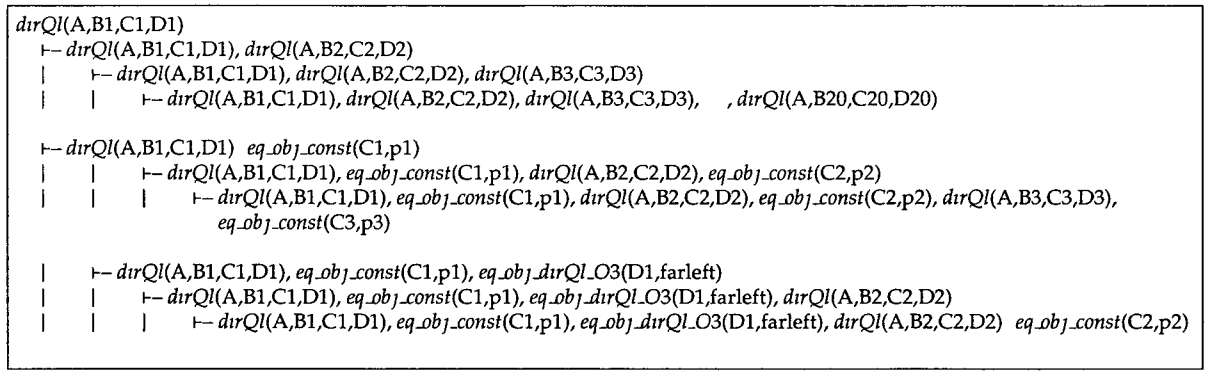

Figure 46: Example Search Space Lattice

It can be seen that with 10 instantiations of *eq_obj_const*/2 in a permutation of the 59 objects declared in our Background Knowledge, the search space is already 59^{10}, assuming the search engine naively picks objects repeatedly. When adding the remaining predicates to the search space, the size becomes astronomical.

Fortunately, the ILP algorithms that we use come with multiple optimization and pruning techniques, such as the bottom clause saturation, redundant clauses detection, and pruning based on minimum support and confidence. These have been described in Section 2 already. Even so, it is not hard to realize the enormous space that we are searching.

5.5.5 Summary

In this section, we looked at the different input required by the ILP algorithms. Background knowledge specifies the objects to be included, class hierarchy, region discretization, as well as more powerful usage such as conversion of reference frames. Language bias defines the boundary of the hypothesis search space, which shapes what the generated rules shall look like. Next, we look at the output of the ILP algorithms.

5.6 Output of ILP Algorithms

The ILP algorithms generate a list of rules according to the input (ie. background knowledge, training examples, language bias, settings, etc.). Figure 47 list some of the rules generated by ALEPH for the Krislet agent. The rules listed in the figure show some of the fundamental behavior of Krislet. We shall study ILP output in more details in Section 6.

```
action(A,dash,fw,far)
    -
    dirQl(A,B,C,D), instanceOf(C,ball), eq_obj_dirQl_O3(D,front), distQl(A,B,C,E),
    eq_obj_distQl_O3(E,far)

action(A,turn,farright,same)  - not_objectSeen(A,B), instanceOf(B,ball)
action(A,turn,farright,same)  -
    dirQl(A,B,C,D), eq_obj_dirQl_O3(D,farright), instanceOf(C,ball), distQl(A,B,C,E),
    eq_obj_distQl_O3(E,far)
action(A,turn,left,same)      - dirQl(A,B,C,D), eq_obj_dirQl_O3(D,left), instanceOf(C,ball)

action (A,kick,front,far)     -
    dirQl(A,B,C,D), eq_obj_const(C,flb), eq_obj_dirQl_O3(D,farleft), dirQl(A,B,E,D),
    inteam(E,unkn), dirQl(A,B,F,D), instanceOf(F,line), dirQl(A,B,G,H), eq_obj_const(G,fc),
    dirQl(A,B,I,H), instanceOf(I,player), dirQl(A,B,J,K), eq_obj_dirQl_O3(K,front),
    instanceOf(J,goal), dirQl(A,B,L,M), eq_obj_const(L,q1), distQl(A,B,N,O), instanceOf(N,ball),
    eq_obj_distQl_O3(O,same)
```

Figure 47: Sample Generated Rules for Krislet

5.7 Inference Engine

As we know, the generated rules are definite clauses that represent *Situation* $\Rightarrow$ *Response*. The purpose of the inference engine is to take these generated rules, which are now considered as the learned knowledge of the agent, and apply them on any situation to produce a response. These situations can either come from input stimulus provided by the RoboCup Soccer Server at every time cycle (deployment phase), or in a batch mode where they are provided one-by-one to compare the predicted outcome against the actual one (testing phase). In both cases, we have decided to use backward chaining (reasoning) to infer the outcome from the given facts. Basically, backward chaining starts with a list of goals, and works backward

from the consequent to the antecedent to determine if there is facts available that will support any of these consequents [Wik]. The inference engine would continue to search the rules until it finds one with a consequent that matches the desired goal. If the antecedent of the matching rule is not known to be true, it is in turn added to the list of goals. The inference engine must find facts that satisfy all goals in order to determine the root consequent to be true. In Figure 31, we can see that the inference engine also requires background knowledge to be provided. This is because it may contain clauses and facts that are required to allow literals in the rules to be resolved.

Since the Prolog language itself supports backward chaining with its inference engine, it is the perfect candidate to use as our inference engine as well. In the subsequent sections, we shall see Prolog is used to perform both testing and real-time reasoning.

5.7.1 Testing Phase

As mentioned in Section 4, we employ 5-fold cross validation to collect our metrics. The rules generated from each of the 4 players are combined and fed into the inference engine, and the knowledge base from the remaining player will be used as the test set. We wrote a simple Prolog program *imisys_runtest.pl* [NMAb] that loads these rules into the YAP Prolog Compiler, along with the ground facts from the test set, and then runs tests to collect the metrics. The main task of the program is the *testData*/5 predicate, which calls the *test*/4 predicate that is provided in the ALEPH source code. The arguments of *test*/4 are:

- ExampleFile (input): file name containing all examples (positive or negative) for this particular action, as discussed in Section 5.5.3.
- Show (input): whether it should print out all examples covered by each rule.

'noshow' means do not print.

- Covered (output): indicates the number of examples (positive or negative) are covered (ie. true positive or false positive)
- Total (output): indicates the total number of examples (positive or negative).

Basically, *test*/4 takes each of the examples in the ExampleFile, performs a query on it, and Prolog then uses backward chaining to determine whether it succeeds or not. It repeats the process for both positive and negative examples. Notice that when using the *testData*/5 predicate, we must remove the predicates *actionQn*/3 and *actionQn*/4. Otherwise, the positive examples will always be found true by Prolog trivially, without using backward chaining to determine the truth of the example based on the rule itself. The results allow us to calculate the final f-measure metric for each experiment.

Next, we shall see how the inference engine is implemented and used in the deployment phase.

5.7.2 Deployment Phase

The deployment phase is different from testing in that it is time-critical: input stimulus supplied by the RoboCup Server must be processed and an output action must be generated, all within a limited amount of time. Obviously, the inference engine can process the input stimulus faster if there are only a few rules loaded in memory rather than many. Therefore, it is very desirable for the rules generated by the ILP engines to be as concise as possible, but still accurate.

Our basic agent is written in the Java language to communicate with the RoboCup server. Java is easy to use and it is also the language used in the CBR project in our lab [FE08], so code reuse is made possible. We list a few options

below that we considered in the integration of the inference engine with the basic Java agent:

1. **Implement the inference engine in Java:** can take a lot of effort to implement an inference engine that supports backward chaining, and which takes into account background knowledge. We decided not to proceed with this option.

2. **Reuse existing embeddable Java Prolog engine:** there are many Prolog engines in the public domain that are written in Java [Maz,GRH,Winb,Wina]. However, most of them are private projects, and support for standard Prolog cannot be guaranteed. Furthermore, bugs could be very difficult to fix and support could be minimal. Therefore, we decided not to proceed with this option as well.

3. **RoboLog Koblenz:** Murray et al [Mur99] presented a multi-layered architecture that accepts scripts written in Prolog to execute plans or actions for RoboCup agents. The architecture is shown in Figure 48. At first glance, this looks to be a promising platform to use. However, as the figure shows, the basic skills that are performed at the lowest level, such as searching for ball, dashing, kicking, even avoiding obstacles and dribbling, are implemented in the Robo-Log kernel itself. Only high-level abilities and plans are written in Prolog scripts. We require the inference engine to work at the lowest skill level since it is our goal to attempt imitation at this level. Therefore, this option is ruled out as well.

4. **Java interfacing with Prolog Compilers:** An open-source library called *InterProlog* [Dec] is available, which interfaces existing Prolog compilers such as YAP, XSB, SWI with the Java language. It does this by either using sockets to connect between the Java program and the Prolog Compiler, or, for XSB

Prolog [Aut], also through *Java Native Interface* (JNI). We feel that using an existing Prolog inference engine is the safest bet, although we cannot predict whether performance will meet our requirements (ie. 100ms between input from Soccer Server to action). We decide to choose this option nevertheless, since it provides the easiest and safest solution, with the least amount of effort to produce a working agent.

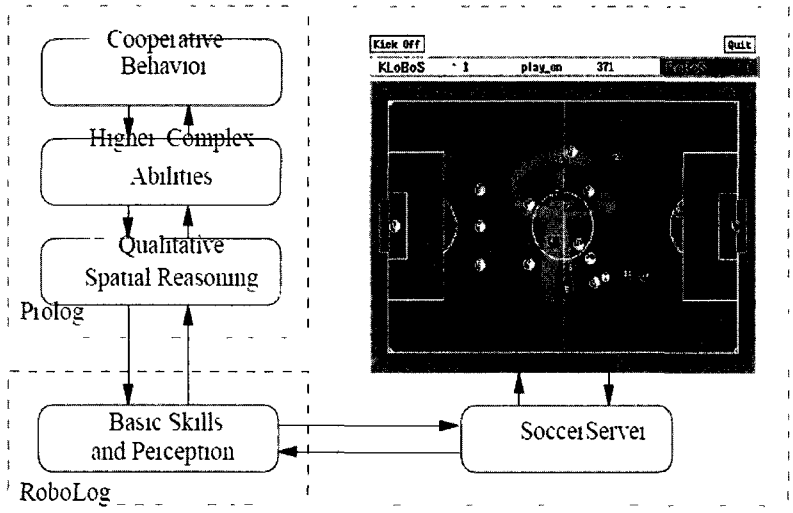

Figure 48: System Architecture of RoboLog [Mur99]

Between the socket and JNI options, we decided to choose the latter, which only works with XSB Prolog, because it is faster than communication through sockets. We also did some preliminary tests using YAP through sockets, but the implementation was unstable, often causing YAP to crash. There was no such issue with XSB/JNI, although initial tests showed that the time it takes to predict an action can vary widely. This can be due to the number of rules loaded in memory, as well as their order. Nevertheless, we shall proceed with this plan. In general, XSB supports the same set of standard Prolog predicates as YAP, so we expect minor differences in terms of input and output formats and their operations.

Backward chaining in the deployment phase works similarly to the testing phase. However, instead of providing a query such as $action(1, turn, left, same)$ to Prolog to determine success or failure, now the following query is sent to XSB Prolog:

$$action(cycle, Action, ActionParam1, ActionParam2).$$

where *cycle* is a constant indicating the current time cycle. The remaining arguments, *Action*, *ActionParam*1, *ActionParam*2 are variables which will be bound with constant terms by the Prolog inference engine upon completion of the backward chaining algorithm. Therefore, the action along with its parameters can be sent toward the RoboCup Soccer Server for execution.

5.8 Agent Execution

Before an action can be sent to the RoboCup server along with its parameters, it must be converted into numerical values for execution since according to the way our language bias is setup the action parameters from the rules are qualitative in nature. For simplicity, we shall simply convert using the values from Figure 29. Ideally, the action predicate may contain variables that appear in the literals (such as *dirQl*/*distQl*) of the rule body. In such cases, a direct mapping straightly using numerical values can be done, without converting to qualitative domain first. However, with the reasons already mentioned in Section 5.5.4, we shall not proceed with this approach.

5.9 Summary

In this section, we first looked at the limitations and assumptions we imposed on our imitation system. Then we looked at the system overview, and studied each of

the component in great details. The entire reasoning system, from training using ILP algorithms to the inference engine, is implemented using standard Prolog first-order logic, making it a highly coherent system that allows high portability to other application domains. Next, we shall use this system to perform some real experiments.

Chapter 6

Experimental Results

6.1 Introduction

We shall present our experimental results in this section. Sections 6.2 and 6.3 discuss results with ALEPH and CLASSIC'CL, respectively. Section 6.4 looks at the run-time performance using the ILP-generated rules. Section 6.5 compares the ILP results with the CBR approach in [FE08]. All experiments were performed on machines with this configuration:

- **Operating System:** RedHat Linux 2.6.9-89.0.11.ELlargesmp
- **CPU:** Intel(R) Xeon(R) CPU X5570 @ 2.93GHz
- **Cache Size:** 8192 KB
- **Total Memory:** 49 GB
- **HardDrive Capacity:** 500 GB

6.2 ALEPH

6.2.1 Introduction

This section is dedicated to experiments performed with the ALEPH software, which we will be using to emulate Progol. ALEPH has many settings to adjust the search process, and the ones we plan to perform experiments with are already mentioned in Section 4. We will first start off by describing the settings in Section 6.2.2. Then, we shall divide the experiments into the egocentric frame of reference (Section 6.2.3) and the multiple frames of reference (ie. egocentric + relative + intrinsic) (Section 6.2.4). Within each of these sections, we shall perform experiments on the three RoboCup Soccer agents: Krislet, Sprinter and CMUnited. Data analysis and comparison shall be performed. Section 6.2.3 describes the usage of a different region discretization, which illustrates the shortcoming of the current approach. In Section 6.2.5, we modified the original Krislet to an agent that uses intrinsic object distance, and perform experiments with it.

6.2.2 ALEPH Settings

Table 14 reiterates the different settings that we shall experiment with in this section, along with descriptions and how they affect the search process.

To emulate Progol, the search strategy is set to "heuristic" and evaluation function is set to "compression". In Section 6.2.3, we shall explain the difference between using the compression and coverage functions.

Setting	Values to test	Descriptions
Max # of nodes	2500, 5000, 10000	The maximum number of candidate clauses that will be generated (visited) for each example, before ALEPH terminates the search and moves on to the next example Obviously, the more nodes visited the higher chance of finding a better clause (if the best has not been found yet), but it will also take more time
Search Strategy	BF, DF, heuristic	The search strategy affects how the clauses are enumerated BF (breadth-first) searches shorter clauses before longer ones, whereas DF performs depth-first traversal on the hypothesis space Heuristic enumerates clauses in a best-first manner, based on their scores computed by a evaluation function (see next)
Evaluation function	compression, coverage	The evaluation function assigns a utility (score) to each candidate clause • compression util = P-N-L+1 • coverage util = P-N where P,N = # of positive and negative examples covered by the clause, L = # of literals in the clause

Table 14: ALEPH settings for Experiments

6.2.3 Ego-centric Frame of Reference

Introduction

In this section, we perform a number of experiments using visual information only based on the egocentric frame of reference (ie. what is supplied by the RoboCup Soccer server). First, we look at how the maximum number of nodes allowed affects the search in terms of accuracy and performance. Then we shall look at the effects, pros and cons of using different search strategies.

Comparison of Maximum Number of Nodes

The purpose of this experiment is to compare the performance (in terms of time) and accuracy of the rules generated, given the number of maximum nodes (clauses) that can be visited per positive example This gives us an overall picture of how the ALEPH algorithm performs, and then we can concentrate on using one fixed number for subsequent experiments. Table 15 shows the results of the average search time and f1-measure, separately for the dash and turn actions of Krislet, and for each of the search strategies. Each number reported is an average taken

from 5 game plays. The results are obtained using the ILP inputs as described in Section 5.5. Note that we exclude the kick action to save computation resources, but it will be included in subsequent experiments.

Search Method = Heuristic	$AvgTime_{dash}$ (s)	$AvgTime_{turn}$ (s)	$f1_{dash}$	$f1_{turn}$
Max Node = 2500	5001	3273	0.8265	0.9633
Max Node = 5000	3748	4662	0.8565	0.9711
Max Node = 10000	7200	11552	0.9106	0.9693
Search Method = BF				
Max Node = 2500	40918	33663	0	0 4842
Max Node = 5000	96226*	64270	0.3332	0 8807
Max Node = 10000	142148*	91180*	0 3332	0.8978
Search Method = DF				
Max Node = 2500	9330*	29581*	0.6815	0 9544
Max Node = 5000	26079*	65287*	0 7384	0 9354
Max Node = 10000	93166*	110396*	0.8342	0 8862

Table 15: Comparison of # Max Nodes

In the case where the search strategy is heuristic, both the average time taken and the f1-measure do not differ significantly for maxima equal to 2500 and 5000. With 10000 maximum nodes, the time taken has increased more or less 2 folds, but the f1-measure does not show a significant increase. This points out that using maximum nodes of 5000 may be good enough, without spending too much computation time.

Using BF and DF search, computation time increases dramatically. For the entries marked by (*), these searches take more than 48 hours before we had to terminate them pre-maturely due to practicality reasons. This applies to the results shown in the rest of this thesis. Hence, these numbers do not represent an accurate value but serve only as a lower bound. The reason for the increased run-time is that BF takes too long to find rules that require a high number of literals. In fact, most of the time it cannot find any rules at all, due to the maximum nodes limit. This is

indicated by the f1-measure value of 0 for the *MaxNode* = 2500 case. For example, a Krislet *dash* action is performed if the ball is in front and at a certain distance (to associate with the dash power). Therefore, the following rule can explain this behavior:

$$action(A,dash,fw,far) \leftarrow dirQl(A,B,C,D), instanceOf(C,ball), eq\_obj\_dirQl\_O3(D,front), distQl(A,B,C,E), eq\_obj\_distQl\_O3(E,far).$$

On the other hand, a Krislet turn action is executed if the ball is at a certain angle from the viewpoint (to associate with the turn angle). The following rule can explain this behavior:

$$action(A,turn,left,same) \leftarrow dirQl(A,B,C,D), eq\_obj\_dirQl\_O3(D,left), instanceOf(C,ball)$$

This time, it is easier for a BF search for find such clause, as indicated by the reduced search time and the higher f1-measure values. Indeed, we found that for the turn left action, a few runs of BF search could find the rule within 300 seconds. For DF search, clauses are searched in the depth-first manner. Thus, the search will quickly start searching from the longest clauses, before it slowly backtracks to the shorter ones, and then repeats. Given the maximum depth is 20 and the expected rule length is around half of that, it is not a surprise that it takes a longer time before it can find the clauses that are relevant. However, it can still produce high accuracy by finding clauses that are more data-fitting.

From Table 15, it suggests that using the heuristic search strategy would provide the best results in the shortest amount of time, and it looks like both BF and DF search methods offer worse performance in terms of time and accuracy. However, more experiments in the next section show that the BF search strategy can generate better quality rules.

Comparison of Search Strategies

Krislet

Table 16 confirms some of the information that is already displayed in Table 15, where we have already learned that using the heuristic search strategy looks to be the most promising. Indeed, Table 16 shows that with heuristic, $f1_{global}$ is statistically better than the other two search methods. However, the last three columns of the table reveal something more interesting.

Search	$f1_{dash}$	$f1_{turn}$	$f1_{kick}$	$f1_{global}$	$Runs_{tot}$	$Time_{avg}$	$Rules_{tot}$	$CovRatio$	$RuleLen_{avg}$
Heuristic	0 856	0 971	0 277	0 653 ±0 128	60/60	2713	485	25 0	11 7
BF	0.333	0 881	0	0 377 ±0 203	58/60	47968*	19	523 5	3 7
DF	0.738	0 887	0 277	0 596 ±0.109	50/60	32487*	228	45.8	16 6

Table 16: Comparisons of Search Methods in Egocentric Reference for Krislet

The heuristic search method generates the highest number of rules (485), with each rule covering on average 25.0 positive examples. The high number of rules generated may not be desirable since if they have low coverage (*CovRatio*), as it implies that these rules may be too data-fitting. With too many rules, it will also take more time to process, which is especially important in real-time applications like RoboCup. $RuleLen_{avg}$ indicates that each rule has 11.7 literals on average. This seems to be quite high, which is another indicator of data-fitting. BF search shows the best coverage ratio of 523.5 examples per rule using only 19 rules, representing about 20 folds that of heuristic search. This illustrates that although the overall performance as indicated by $f1_{global}$ suffers due to reasons given in the previous section (ie. rules were simply not found at all for some of the action parameters), rules that are actually generated have high coverage, hence quality. Indeed, this is also indicated by the very low average number of literals per rule of 3.7. Therefore, we do not want to dismiss the results generated by BF search. As for DF search,

it generates very long rules (16.6 literals) with a coverage not significantly better than heuristic search. With its high run-time, this search method does not seem preferable in this case.

The product of $Rules_{tot}$ and *CovRatio* gives the total number of positive examples covered, which is 12102, 9946, 10434, respectively for heuristic, BF, and DF search. Obviously, more examples being covered will mean a higher f1-measure, which can be correlated in the table.

With this analysis in mind, we can actually look at some of the rules generated by each search method, as shown in Table 17. Clearly, the rules attributes can be correlated to the results shown in Table 16. For all the *turn.left* rules shown here, they subsume the more general clause:

$$action(\text{A,turn,left,same}) \leftarrow dirQl(\text{A,B,C,D}), eq\_obj\_dirQl\_O3(\text{D,left}), instanceOf(\text{C,ball})$$

which states that the action *turn.left* should be performed if the ball is seen on the left side of the observing agent, which is part of the intrinsic behavior of Krislet. Only using the BF search, this exact clause can be found. For others, many literals are included as part of the rules, which makes them more data-fitting. For *turn.farright*, in all three searches, this rule can be found:

$$action(\text{A,turn,farright,same}) \leftarrow not\_objectSeen(\text{A,B}), instanceOf(\text{B,ball})$$

it indicates that short valid clauses can be found regardless the search strategy, if they are high quality. Indeed, this rule alone covers 90% of positive examples of *turn.farright*. This rule again represents Krislet's behavior of turning clockwise to look for the ball when it cannot be seen.

The small dip in $f1_{dash}$ in heuristic search is due to the action *dash.close* alone, which has a f1 score of 0.700, while the f1 score for *dash.far* and *dash.veryfar* is 0.939 and 0.930, respectively. The reason for the relatively low *dash.close* f-measure

Search	Example Rules Generated
Heuristic	• action(A,turn,left,same) - dirQl(A,B,C,D), inteam(C,unkn), dirQl(A,B,E,F), instanceOf(E,ball), dirQl(A,B,G,F), eq_obj_dirQl_O3(F,left), distQl(A,B,E,H), eq_obj_distQl_O3(H,far) • action(A,turn,left,same) - dirQl(A,B,C,D), eq_obj_dirQl_O3(D,left), dirQl(A,B,E,D), instanceOf(E,ball), distQl(A,B,C,F), eq_obj_distQl_O3(F,close), distQl(A,B,E,F), not_objectSeen(A,G), eq_obj_const(G,p1) • action(A,turn,farright,same) - not_objectSeen(A,B), instanceOf(B,ball) • action(A,turn,farright,same) - dirQl(A,B,C,D), eq_obj_const(C,frt), dirQl(A,B,E,F), eq_obj_dirQl_O3(F,farright), instanceOf(E,ball)
BF	• action(A,turn,left,same) - dirQl(A,B,C,D), eq_obj_dirQl_O3(D,left), instanceOf(C,ball) • action(A,turn,left,same) - dirQl(A,B,C,D), eq_obj_dirQl_O3(D,left), instanceOf(C,ball), distQl(A,B,C,E), eq_obj_distQl_O3(E,far) • action(A,turn,farright,same) - not_objectSeen(A,B), instanceOf(B,ball) • action(A,turn,farright,same) - dirQl(A,B,C,D), eq_obj_dirQl_O3(D,farright), instanceOf(C,ball), distQl(A,B,C,E), eq_obj_distQl_O3(E,far)
DF	• action(A,turn,left,same) - dirQl(A,B,C,D), inteam(C,unkn), dirQl(A,B,E,F), instanceOf(E,ball), instanceOf(E,object), instanceOf(B,player), instanceOf(B,object), eq_obj_const(E,ball1), eq_obj_const(B,self), dirQl(A,B,G,F), eq_obj_dirQl_O3(F,left), instanceOf(G,object), dirQl(A,B,H,I), instanceOf(H,object), dirQl(A,B,J,I), instanceOf(J,player), instanceOf(J,object), distQl(A,B,E,K), eq_obj_distQl_O3(K,far) • action(A,turn,left,same) - dirQl(A,B,C,D), eq_obj_const(C,gr), eq_obj_dirQl_O3(D,farleft), dirQl(A,B,E,D), instanceOf(E,object), instanceOf(B,player), instanceOf(B,object), eq_obj_const(B,self), dirQl(A,B,F,D), instanceOf(F,object), dirQl(A,B,G,H), eq_obj_dirQl_O3(H,left), instanceOf(G,ball) • action(A,turn,farright,same) - dirQl(A,B,C,D), not_objectSeen(A,E), instanceOf(E,ball)

Table 17: Comparisons of Rules Generated in different Search Methods for Krislet

is due to the sensitivity of ILP search to the boundary values of discretized regions, which shall be described in Section 6.2.3.

For BF search, *dash.far* alone has a 1.0 f-measure, whereas no rules were found at all for *dash.close* and *dash.veryfar*. We have already discussed that the success of finding rules in BF/DF highly depends on the ordering of the candidate clauses. If the BF search happens to traverse the search lattice in such a way that a rule is discovered that covers a positive example, while meeting the minimum requirements of support and confidence, it is very likely that it covers a lot of other examples as well (as indicated by *CovRatio* and the perfect f-measure score for *dash.far*).

Kick Action At the beginning, we mentioned that we need to use the coverage evaluation function in order to discover rules for the kick action. The problem lies in the fact that there are very few examples (low single digit) for the kick actions. Even after combining multiple game plays into one as mentioned in Section 4, the number is still low after the kick actions are separated into their corresponding regions according to their parameters. Recall that the compression evaluation function assigns each candidate clause with a score using the utility = P-N-L+1, where P,N = # of positive and negative examples covered by the clause, and L = # of literals in the clause. This is the function normally used by Progol. A clause having a higher score is placed at a higher priority to be searched first. The function indicates that a clause with fewer literals is preferred. At the beginning of each iteration, ALEPH will pick the ground positive example as the best candidate clause. L is assigned with the value # of literals + 1, which means the score for any ground positive example is 1-0-2+1 = 0. Candidate clauses will only be placed on the active search list if they have a higher score than the current best, and the one with the best score will become the current best itself. Hence, this means that a candidate clause will not be eligible for search if it has a score ≤ 0.

Now, this is where the problem occurs. With very few examples to cover, P is very small. Even if N is 0 (ie. no negative example is covered), L itself is still a comparable number to P, considering a kick action for Krislet will require several literals to describe. Hence, using the compression function, we cannot find any rules for the kick action at all, simply because all the clauses have a score ≤ 0. With the coverage utility (P-N), the number of literals within a candidate clause is taken out of the equation. Therefore, a clause with simply $P > N$ will carry a non-zero score. Hence, rules can be found. The trade-off is that since L is no longer part of the equation, shorter clauses will not be given preference. Therefore, rule lengths become longer on average. Even with heuristic search, a total number of 145 rules are found for the kick action, each having 15.9 literals on average, which is greater than the average number of 11.7 literals. Below shows an example rule found for *kick.farleft*:

action(A,kick,farleft,far) ← **dirQl(A,B,C,D)**, **eq_obj_const(C,gl)**, dirQl(A,B,E,D), **eq_obj_dirQl_O3(D,farleft)**, instanceOf(E,player), dirQl(A,B,F,G), inteam(F,team2), dirQl(A,B,H,I), eq_obj_dirQl_O3(I,right), dirQl(A,B,J,K), **instanceOf(J,ball)**, dirQl(A,B,L,K), instanceOf(L,player), **distQl(A,B,J,M)**, **eq_obj_distQl_O3(M,same)**.

The literals that are bolded together represent the underlying Krislet behavior, where it kicks the ball toward the left goal when the ball is right beside the player (ie. *same* distance). Even though the rules found are quite over-fitting, applying the rules together on a single test agent still provides a good $f1_{kick}$ value.

The results for kick show a very low score. This is because as mentioned before there are so few kick examples in the test sets themselves (eg. 7 examples of kick.farright; 3 examples of kick.front, out of 5 game plays combined), such that even if one out of the few examples is not covered by the rules, the f1 score is lowered by a significant amount.

Sprinter

Search	$f1_{dash}$	$f1_{turn}$	$f1_{global}$	$Runs_{tot}$	$Time_{avg}$	$Rules_{tot}$	$CovRatio$	$RuleLen_{avg}$
Heuristic	0.986	0.666	0.803 ±0.117	33/33	3007	160	85.6	8.28
BF	0.987	0.680	0.811 ±0.116	31/33	20766*	67	204 7	3 39
DF	0.647	0.663	0.656 ±0.182	24/33	20339*	96	66 1	11.9

Table 18: Comparisons of Search Methods in Egocentric Reference for Sprinter

Table 18 shows the pattern in the results for Sprinter as Krislet. Heuristic search takes the least amount of time to finish and discovers the most rules. BF search produces rules of high quality by covering the most positive examples per rule, with the least number of literals in each one. $f1_{dash}$ for DF search produces a significantly lower score than the other two, because it cannot find any clauses for *dash.veryfar*. The following rule can be found by both BF and heuristic search.

action(A,dash,fw,veryfar) ← dirQl(A,B,C,D), eq_obj_dirQl_O3(D,front), distQl(A,B,C,E), eq_obj_distQl_O3(E,veryfar), instanceOf(C,goal).

The weakness in $f1_{turn}$ is due to the very few examples for *turn.farleft* (1 or 2 examples, and none in some cases), hence no rules could be found using the compression utility. This is the same problem as the Krislet kick action. We shall simply ignore *turn.farleft* since it is rarely used anyway, and does not prohibit us from trying out the agent on the field. Table 19 shows a few of the rules generated by heuristic and BF search, with the relevant literals being bold.

As one can see, BF search produces rules that are very compact and describe the Sprinter's behavior exactly. It produces only 1 reference to a player, which is irrelevant to its behavior, compared to 72 and 115 references using heuristic and DF search, respectively. Also noteworthy is that BF produces rules using the predicate *instanceOf*(C,goal), whereas rules produced by heuristic search use

Search	Example Rules Generated
Heuristic	• action(A,dash,fw,close) ← dirQl(A,B,C,D), inteam(C,unkn), **dirQl(A,B,E,F), eq_obj_const(E,gr), eq_obj_dirQl_O3(F,front), distQl(A,B,E,G), eq_obj_distQl_O3(G,close)** • action(A,turn,farright,same) ← **dirQl(A,B,C,D), eq_obj_const(C,gl), eq_obj_dirQl_O3(D,farleft)** • action(A,turn,farright,same) ← **dirQl(A,B,C,D), eq_obj_const(C,gr), eq_obj_dirQl_O3(D,farleft)** • action(A,turn,left,same) ← dirQl(A,B,C,D), eq_obj_const(C,ll), **dirQl(A,B,E,F), eq_obj_dirQl_O3(F,left), instanceOf(E,goal)**, dirQl(A,B,G,F), instanceOf(G,player)
BF	• action(A,dash,fw,close) ← **dirQl(A,B,C,D), eq_obj_dirQl_O3(D,front), distQl(A,B,C,E), eq_obj_distQl_O3(E,close), instanceOf(C,goal)** • action(A,turn,farright,same) ← **distQl(A,B,C,D), eq_obj_distQl_O3(D,same), instanceOf(C,goal)** • action(A,turn,farright,same) ← **dirQl(A,B,C,D), instanceOf(C,goal), eq_obj_dirQl_O3(D,farleft)** • action(A,turn,left,same) ← **dirQl(A,B,C,D), eq_obj_dirQl_O3(D,left), instanceOf(C,goal)** • action(A,turn,right,same) ← **dirQl(A,B,C,D), eq_obj_dirQl_O3(D,right), instanceOf(C,goal)**

Table 19: Example Rules Generated for Sprinter

mostly *eq_obj_const*(C,gl/gr). This highlights the benefit of using output variables for the predicate template *dirQl*(A,B,C,D), such that the variables can be associated with either a constant term or as an instance of a concept (in this case goal). In this way, BF search produces clauses that can be very general.

CMUnited

We would not expect ILP learning on the CMUnited team to do well, given the sophisticated nature of its behavior, and the fact that we only take into account a limited sets of input objects and attributes for learning (see Section 5.2). The maximum power CMUnited players use is 100, so the action *dash.veryfar* has no positive examples with the current region boundary values used. Also, the kick power "close" is included as part of the learning. Table 20 shows the results, using the same settings as the other two players.

Search	$f1_{dash}$	$f1_{turn}$	$f1_{kick}$	$f1_{global}$	$Runs_{tot}$	$Time_{avg}$	$Rules_{tot}$	$CovRatio$	$RuleLen_{avg}$
Heuristic	0 090	0 006	0 01	0 018 ±0 001	69/90	11835*	130	2.9	11 9
BF	0 082	0	0	0 009 ±0 001	45/90	70645*	4	24 75	5 25
DF	0	0	0	0	19/90	51815*	2	0	13 5

Table 20: Comparisons of Search Methods in Egocentric Reference for CMUnited

The results look very poor indeed. Very few rules could be found compared to the other agents, and the coverage ratio is extremely low. We further looked at the coverage ratio for the training set (*CovRatio* shows the one for test sets), and found that it is 4.84 for heuristic, 11.75 for BF, and 2 for DF. These are very low values indeed, which means the rules found are very over-fitting and provide very low generalization. In the case of DF, it simply meets the minimum support requirement. Hence it is not a surprise that coverage for the test sets are small as well.

As we have already mentioned, the limited set of input objects and attributes are partly responsible for the poor results. As we shall see in the next Section, variations in region discretizations can be a big contributor to the problem as well.

Sensitivity to Region Discretization

Back in Figure 41, we defined the discretized region boundary values through background knowledge. In particular, we define these predicates:

$$dist\_same(\mathrm{A}) \leftarrow \mathrm{A} < 1.0\,.$$
$$dist\_close(\mathrm{A}) \leftarrow \mathrm{A} \geq 1\,0, \mathrm{A} < 5\,0\,.$$

However, Figure 49 shows a snippet from the database *krislet.1.kb* (ie. Krislet game-play #1) to illustrate Krislet's true behavior. The ground facts in bold indicate the cases where the ball is exactly 1.0 unit distance from the observing player. Observe that the actions performed were *turn.farright* and *kick.farright.far*, respectively. The first action is when Krislet turns to look for the goal after the ball gets

```
objectSeen(1151,ball1)
distQn(1151,self,ball1,1.0)
actionQn(1151,turn,40 0)

objectSeen(2350,ball1)
dirQn(2350,self,ball1,0 0)
distQn(2350,self,ball1,1 1)
actionQn(2350,dash,11 0)

objectSeen(1157,ball1)
distQn(1157,self,ball1,0 9)
actionQn(1157,turn,40 0)

objectSeen(20940,gl)
dirQn(20940,self,gl,42 0)
objectSeen(20940,ball1)
distQn(20940,self,ball1,1 0).
actionQn(20940,kick,100 0,42 0)
```

Figure 49: Snippet of ground facts from Krislet game-play #1

close to it, while the second action is when Krislet kicks the ball toward the goal. In fact, out of the 37 cases in the database where the ball distance is 1.0, 3 of them were turn actions and 34 were kick actions. Only if the ball distance is greater than 1.0 is the dash action performed. Now, when ILP search is performed on the action *dash.close*, the following (ideal) candidate clause would be visited:

action(A,dash,fw,close) .- , *distQl*(A,B,C,D), *eq_obj_const*(C,ball1), *eq_obj_distQl_O3*(D,close), .

However, because of the current definition of *dist_close*, the body of this clause will match the 37 negative examples (turn and kick), such that due to low confidence, the clause would not be accepted. This makes the search work harder to find rules that would be more over-fitting, such as the clause below where a player is used to resolve the conflicts:

action(A,dash,fw,close) ← dirQl(A,B,C,D), eq_obj_const(C,p3), dirQl(A,B,E,F), eq_obj_dirQl_O3(F,front), distQl(A,B,E,G), eq_obj_distQl_O3(G,close), instanceOf(E,ball)

Therefore, from the discussion above we now know that *dist_same*(A) should be defined as A $\leq$ 1.0, instead of A < 1.0. However, we only know this because we have advanced knowledge of Krislet's exact behavior. Such human involvement in entering advanced knowledge about the agent itself into the background is obviously undesirable. Hence, the good results we obtained in the previous sections were possible simply because the existing background knowledge happens to coincide with the underlying behavior of Krislet and Sprinter (except for the misalignment for *dash.close*, which we had intentionally allowed to illustrate the problem). This problem applies to any discretized region including direction and power. For example, Krislet performs the dash action toward the ball with a power value that is 10 times that of the distance away from the ball. The same problem will happen if the predicates *power_** do not define ranges that are exactly 10 times the values defined by the predicates *dist_**. To prove our point, Table 21 shows the results of the Krislet experiments being re-run using heuristic search and the boundary values listed in Figure 50, which are intentionally assigned *not* to align with Krislet's true behavior. Notice that the predicate *dir_front* is assigned with a range between ±3, instead of exactly 0. Also, the predicates *power_** are no longer proportional to *dist_**.

$f1_{dash\ close}$	$f1_{dash\ far}$	$f1_{dash\ veryfar}$	$f1_{turn\ farleft}$	$f1_{turn\ left}$	$f1_{turn\ right}$	$f1_{turn\ farright}$
0 008	0 378	0	0 992	0 998	0.999	0.981

Table 21: Results for Krislet with unaligned Boundary Values

The results are broken down per action type and its parameter. We ignore the kick action here as this is for illustration purpose only. For both *dash.close* and *dash.veryfar*, satisfactory rules cannot be found, and results for *dash.far* are much worse than before. The problem with *dash.close* has already been explained previously. For *dash.veryfar*, the predicate *power_veryfar* indicates a distance of over

```
dir_farleft(A) - A ≤ -23 0
dir_left(A) - A > -23 0, A ≤ 3 0
dir_front(A) - A > 3 0, A < 3 0
dir_right(A) - A ≥ 3 0, A < 23 0
dir_farright(A) - A ≥ 23 0

dist_same(A) - A < 1 0
dist_close(A) - A ≥ 1 0, A < 5 0
dist_far(A) - A ≥ 5 0, A < 30 0
dist_veryfar(A) - A ≥ 30 0

power_same(A) - A < 10 0
power_close(A) - A ≥ 10 0, A < 100 0
power_far(A) - A ≥ 100 0, A < 500 0
power_veryfar(A) - A ≥ 500 0
```

Figure 50: Region Boundary Values unaligned with Krislet's true behavior

500.0 units, resulting in very few examples only (single digits). For *dash.far*, the misalignment of *dist_far* and *power_far* causes conflicts with neighboring regions, which results in low confidence rules that cannot be accepted. The results for all the turn actions seem to be unperturbed. The reason is that, unlike the distance-to-power relationship, the angle of the turning action is a direct 1:1 mapping to the angle corresponding to the target object (in this case, the ball). If the agent does not behave like this, the same boundary value problem will surface.

Hence, the key lesson here is that if some arbitrary boundary values were used instead, completely different (and worse) results would have been obtained. This highlights that such sensitivity of ILP techniques to boundary values of discretized regions can become a major limitation: without proper boundary values defined, rules may not be found at all. We can now also see that the current discretization scheme can prevent rule discoveries for CMUnited, since it most likely works on a completely different discretization scheme than Krislet and Sprinter. We offer some potential solutions in Table 22 as future work.

Solution	*Description*
Discretization techniques	Instead of having humans hand-pick boundary values, existing multivariate discretization techniques that are based on statistical analysis ([FI93, Rat03, cLW00, BR97, Cat91, KK99] can be used to discretize continuous data automatically. Although it may not be perfect, better boundary values can lead to fewer negative examples being covered, which in turn leads to candidate clauses being accepted This method can be used along with the lowering of minimum confidence (see next)
Lowering Minimum Confidence	lowering the minimum confidence could allow rules that are otherwise rejected be accepted. However, the obvious disadvantage is that many rules of low quality could be generated where the same facts can be satisfied by multiple rules which point to different actions. This may require more complicated post-processing tasks to filter them.
Numerical Reasoning	The purpose of doing qualitative reasoning in the first place is to avoid doing numerical reasoning, which significantly increases the ILP search space. However, existing work [MIS00] has used ILP for numerical reasoning, although the database itself has only a few hundred ground facts TILDE, which is a first-order counterpart of the popular decision tree algorithm C4 5 [Qui93] can take care of numerical data as well, and we have seen it being used in [DJJ+98], although it was only used to learn high-level behavior The ALEPH software allows emulation of TILDE which can perform regression-like tasks using a technique called lazy evaluation [SCA99] This could help, for example, identity the linear relation between distance and power
Generalize constant terms	Rules with common constant terms in the head and body can be replaced with variables, which generalize the rule. This can help cover more examples that might otherwise be left out For example, if a rule *action*(A,dash,far) ← , *eq_obj_distQl_O3*(D,far), was generated, replacing the term "far" with a variable allow other distances to be covered as well Of course, this depends on the fact that such a rule can be found in the first place, and even so, the rule may already be very over-fitting

Table 22: Potential Solutions for ILP Sensitivity to Boundary Values

Summary

In this section, we have performed experiments using the egocentric spatial representations alone. In general, the rules generated for simple agents like Krislet and Sprinter can track the true behavior quite well, but complicated agents like CMUnited are very hard to learn from. Indeed, the rules are very data-fitting, but the lack of sufficient objects and attributes used in the language bias prevented us from learning any meaningful rules from CMUnited. Last but not least, we also presented a significant limitation of the ILP learning technique, which is that

discovering valid rules depends highly on the correctness of discretized regions boundary values. We offered some solutions as future work.

6.2.4 Ego-centric/Relative/Intrinsic Frames of Reference

Introduction

In Section 3.3.2, we described different frames of reference for representing spatial knowledge. In this section, we perform experiments by adding relative and intrinsic frames to the spatial representation, in order to determine whether they will add any values.

Changes in Input Files

In order to incorporate relative and intrinsic reference into the spatial representation, the background knowledge must contain the appropriate predicates. Figure 44 in Section 5.5.3 has already described the necessary additions to *dirQl* and *distQl* predicate definitions. Using the same predicates as in the egocentric view permits a compact language bias that is easy to understand and allows shorter rules to be generated.

However, due to such changes to the predicate definitions, we also need to make some adjustments to the language bias. Recall from Section 2.2 that Progol creates a bottom clause (ie. most-specific clause) at the beginning of the search for each positive example, so that the search space it traverses is confined between the most general clause (ie. empty clause) and this bottom clause. Obviously, the longer the bottom clause, the exponentially larger the search space. We have compared the two example bottom clauses, before and after the prediction definition changes, as shown in Appendix B. The bottom clause has 128 literals before the change, and 164 literals after. This significant increase is due to the fact that now there

can be *dirQl* and *distQl* predicates assigned for every pair of objects. Before the change, the second argument of *dirQl/distQl* is always fixed, and refers to "self". This dramatically increased the search time and decreased the chance of finding rules. Indeed, we could not generate a single rule after 24 hours of search time, even using the heuristic method.

Therefore, to solve this problem, we made a minor (but significant) change to the language bias. These two lines from the existing bias:

:- modeb(*,dirQl(+k,**-object,-object,-dir**)).
:- modeb(*,distQl(+k,**-object,-object,-dist**)).

are changed to this:

:- modeb(*,dirQl(+k,**#object,#object,#dir**)).
:- modeb(*,distQl(+k,**#object,#object,#dist**)).

The new language bias produces the bottom clause for the same example, as shown in Figure 51. Notice that there are still the same number of pairs of objects, hence the same number of *dirQl* and *distQl* predicates. The difference is that since the predicates no longer create output variables, but only constants, there is no more need for predicates like *instanceOf*, *inteam*, *hasUnum*, etc. to unify with them. This can be seen in Figure 51 where the only time these predicates appear is when they unify with the predicate *not_object*. However, this change removes the ability for Progol to generalize a specific instance of an object to a class (ie. *instanceOf*(X,player)), which can make the rules become more specific and data-fitting. As we shall see in subsequent sections, this small change in the language bias also drastically reduces the search time from before, yet produces the same or better performance in f1-measure.

action(A,turn,left,same) ←
dirQl(A,self,fcb,farleft), dirQl(A,self,p1,farleft), dirQl(A,self,ll,farleft), dirQl(A,self,fc,left), dirQl(A,self,flb,left), dirQl(A,self,ball1,left), dirQl(A,self,gl,right), dirQl(A,self,p6,right), dirQl(A,self,flt,farright), dirQl(A,self,p8,farright), dirQl(A,fc,ll,farleft), dirQl(A,flt,flb,farleft), dirQl(A,flt,gl,farleft), dirQl(A,flt,ll,farleft), dirQl(A,gl,ll,farleft), dirQl(A,ball1,ll,farleft), , dirQl(A,p6,p1,farleft), dirQl(A,p6,ll,farleft), dirQl(A,p8,fc,farleft), dirQl(A,p8,fcb,farleft), dirQl(A,p8,flb,farleft), dirQl(A,p8,ball1,farleft), dirQl(A,p8,p1,farleft), dirQl(A,p8,ll,farleft), dirQl(A,fc,fcb,left), dirQl(A,fcb,ll,left), dirQl(A,gl,flb,left), dirQl(A,ball1,fcb,left), dirQl(A,p1,fcb,left), dirQl(A,p6,flb,left), dirQl(A,p8,gl,left), dirQl(A,p8,p6,left), , distQl(A,self,p6,far), distQl(A,self,p8,far), distQl(A,self,fc,veryfar), distQl(A,self,fcb,veryfar), distQl(A,self,flt,veryfar), distQl(A,self,flb,veryfar), distQl(A,self,gl,veryfar), distQl(A,self,ball1,veryfar), distQl(A,self,ll,veryfar), distQl(A,ball1,fc,same), distQl(A,fcb,flt,far), distQl(A,p1,fc,far), distQl(A,p1,ball1,far), distQl(A,p6,fc,far), distQl(A,p6,ball1,far), distQl(A,p6,p1,far), , distQl(A,fc,flb,veryfar), distQl(A,fc,gl,veryfar), distQl(A,fc,ll,veryfar), distQl(A,fcb,flb,veryfar), distQl(A,fcb,gl,veryfar), distQl(A,fcb,ll,veryfar), distQl(A,flt,flb,veryfar), distQl(A,flt,gl,veryfar), distQl(A,flt,ll,veryfar), distQl(A,gl,flb,veryfar), distQl(A,gl,ll,veryfar), distQl(A,ball1,fcb,veryfar), distQl(A,ball1,flt,veryfar), distQl(A,ball1,flb,veryfar), distQl(A,ball1,gl,veryfar), distQl(A,ball1,ll,veryfar), , distQl(A,p8,ll,veryfar), distQl(A,ll,flb,veryfar), not_objectSeen(A,B), instanceOf(B,player), instanceOf(B,object), eq_obj_const(B,p2), inteam(B,team1), hasUnum(B,2)

Figure 51: New Bottom Clause with Modified Language Bias

Comparison of Search Strategies

The organization of this section is similar to Section 6.2.3, where we shall discuss each individual agent separately, and evaluate their performances using the same metrics as before.

Krislet

Table 23 shows the results for Krislet experiments repeated for the combined frames of reference. First, all the f1 scores for each action have shown various levels of improvement comparing to Table 16, with BF search producing the best overall improvements. Second, each search method requires much less time to search, with BF reduced by over 1000 folds!

Intuitively, we know that this drastic difference cannot be accounted by the addition of reference frames in the spatial representation, since we know that Krislet's behavior itself depends only on egocentric view. The difference must

Search	$f1_{dash}$	$f1_{turn}$	$f1_{kick}$	$f1_{global}$	$Runs_{tot}$	$Time_{avg}$	$Rules_{tot}$	$CovRatio$	$RuleLen_{avg}$
Heuristic	0 881	0.980	0 396	0.712 ±0.149	60/60	179.7	361	34 0	2.75
BF	0 918	0 980	0 789	0.885 ±0 022	60/60	29 0	229	54.1	2.24
DF	0 880	0 980	0 542	0 772 ±0 093	60/60	311.1	448	27.44	2 53

Table 23: Comparisons of Search Methods in Egocentric-Relative-Intrinsic Reference for Krislet

therefore be due to the change of the language bias. In order to prove this, we repeat the experiments from Table 16 with the new language bias. The results, shown in Table 24, confirm our hypothesis.

Search	$f1_{dash}$	$f1_{turn}$	$f1_{kick}$	$f1_{global}$	$Runs_{tot}$	$Time_{avg}$	$Rules_{tot}$	$CovRatio$	$RuleLen_{avg}$
Heuristic	0 954	0.981	0 683	0 850 ±0 074	60/60	30 2	211	59 3	2.86
BF	0.973	0 981	0.893	0.943 ±0 006	60/60	3 09	178	70.5	2.51
DF	0.954	0.981	0 663	0.842 ±0 074	60/60	26.1	211	59.3	3 00

Table 24: Repeated Experiments in Egocentric Reference for Krislet with Constant Language Bias

BF search took a mere 3 seconds on average per experiment to complete, and produces a high $f1_{global}$ score of 0.943. This is a big difference from 0.377 that we saw in Table 16, where rules could not even be found and tests ran for so long that they had to be terminated manually. Even the boundary-value problem that *dash.close* suffers from seems to have been reduced. Notice also that the results in Table 23 are no better than Table 16, proving that the additional relative and intrinsic information plays no particular role in improving Krislet's performance. This is because we know that Krislet does not particularly make use of any relative or intrinsic frame of reference in its behavior. Table 25 compares some of the actual rules (heuristic only) generated by using the two sets of reference frames, which can provide better explanations for the results seen.

With the egocentric frame only, many rules could be found where the ball is in front and is in close range (indicated by the bold literals), with an extra literal

Ref Frames	Example Rules Generated
Egocentric	• action(A,dash,fw,close) ← **dirQl(A,self,ball1,front), distQl(A,self,ball1,close),** distQl(A,self,frb,veryfar) • action(A,dash,fw,close) ← **dirQl(A,self,ball1,front),** distQl(A,self,fc,close), **distQl(A,self,ball1,close),** distQl(A,self,flb,veryfar) • action(A,dash,fw,close) ← dirQl(A,self,p1,left), **dirQl(A,self,ball1,front), distQl(A,self,ball1,close)** • **action(A,kick,farright,far) ← dirQl(A,self,gl,farright), distQl(A,self,ball1,same).** • action(A,kick,farright,far) ← dirQl(A,self,q9,farright), distQl(A,self,ball1,close), distQl(A,self,q10,far), distQl(A,self,q11,far), distQl(A,self,gl,veryfar) • action(A,kick,left,far) ← distQl(A,self,ball1,same), distQl(A,self,q1,close), distQl(A,self,q11,far), distQl(A,self,flt,veryfar) • **action(A,kick,left,far) ← dirQl(A,self,gl,left), distQl(A,self,ball1,same).** • action(A,kick,left,far) ← dirQl(A,self,q5,left), distQl(A,self,ball1,close), distQl(A,self,p1,close) • action(A,kick,left,far) ← dirQl(A,self,gl,left), dirQl(A,self,ball1,front), distQl(A,self,ball1,close), not_objectSeen(A,B), eq_obj_const(B,p1)
Multi	• action(A,dash,fw,close) ← dirQl(A,self,p7,front), **dirQl(A,p7,ball1,front), distQl(A,self,ball1,close)** • action(A,dash,fw,close) ← dirQl(A,q9,q6,farleft), dirQl(A,q1,q8,right), distQl(A,fc,q3,far) • action(A,dash,fw,close) ← dirQl(A,fc,q5,farleft), distQl(A,q1,q7,far), distQl(A,q7,q3,far), distQl(A,q7,q5,far), • action(A,kick,farright,far) ← **dirQl(A,self,gl,farright), distQl(A,self,ball1,close),** distQl(A,gl,flb,far), distQl(A,gl,ll,veryfar) • action(A,kick,farright,far) ← dirQl(A,lb,flb,left), **distQl(A,self,ball1,same)** • action(A,kick,farright,far) ← dirQl(A,q5,lb,right), distQl(A,ball1,fcb,veryfar), distQl(A,ball1,gl,veryfar), not_objectSeen(A,B), instanceOf(B,player), hasUnum(B,1) • action(A,kick,left,far) ← dirQl(A,ball1,q5,farleft), dirQl(A,q9,ll,farright), distQl(A,fc,q5,close) • action(A,kick,left,far) ← dirQl(A,q9,q1,left), dirQl(A,q2,ll,farright), distQl(A,q9,q4,far) • action(A,kick,left,far) ← dirQl(A,ball1,gl,farleft), dirQl(A,gl,ll,farright), distQl(A,self,q9,close) • action(A,kick,left,far) ← distQl(A,q11,q6,close), distQl(A,q2,q3,far)

Table 25: Example Rules Generated in different Reference Frames for Krislet

used to resolve the boundary-value conflict. A similar situation occurs for the kick action as well. The two bold rules (*kick.farright* and *kick.left*) represent Krislet's behavior perfectly. Generalizing these rules as mentioned in Table 22 will allow the *kick.far* action to be perfectly represented for any angle parameters. As for the multiple frames, many generated rules do not include the ball, but instead use relations between individual flags and players. This means the rules can be quite over-fitting, as indicated by the smaller *CovRatio*. The larger search space allows more rules (361 vs 211) to be found, but at the expense of finding lower quality rules because each iteration is still bounded by maximum nodes. That being said, many of the rules still carry the relevant literals (bold).

It is not a surprise that the average rule length is much shorter than before, since all the variables are now constants instantiated within the *dirQl/distQl* predicates, so that the predicates *instanceOf*, *inteam*, *hasUnum*, *eq_obj_const*, *eq_obj_dirQl_O3* and *eq_obj_distQl_O3* are no longer needed to unify the variables. However, it is also because of this that the rules become less general, as one can see that the BF *CovRatio* has dropped significantly from 523.5 before. With regard to the time taken for the searches, we shall explain in Section 6.2.6 how the language bias can affect dramatically the efficiency of the search performed by ALEPH through the hypothesis space.

Sprinter

Search	$f1_{dash}$	$f1_{turn}$	$f1_{global}$	$Runs_{tot}$	$Time_{avg}$	$Rules_{tot}$	$CovRatio$	$RuleLen_{avg}$
Heuristic	0 997	0 681	0 816 ±0 118	33/33	41	109	125 9	1 83
BF	0 997	0 681	0 817 ±0 118	33/33	1 28	74	183 4	1 49
DF	0 997	0 681	0.816 ±0 118	33/33	53 5	108	127 0	1.90

Table 26: Comparisons of Search Methods in Egocentric-Relative-Intrinsic Reference for Sprinter

Search	$f1_{dash}$	$f1_{turn}$	$f1_{global}$	$Runs_{tot}$	$Time_{avg}$	$Rules_{tot}$	$CovRatio$	$RuleLen_{avg}$
Heuristic	0 997	0 681	0.817 ±0 118	33/33	1.09	74	185 4	1 49
BF	0 997	0 681	0 817 ±0 118	33/33	0 38	74	185.4	1 49
DF	0 997	0.681	0 817 ±0 118	33/33	1 10	74	185.4	1 49

Table 27: Repeated Experiments in Egocentric Reference for Sprinter with Constant Language Bias

The results of Sprinter, as shown in Table 26, are in line with what we have already seen with Krislet. We also repeat the experiments with the constant language bias as shown in Table 27, which does not show any statistically significant difference. We list some of the rules generated by BF in Table 28. Except for very few, all of these rules are perfect representations of Sprinter's behavior. Using BF search, only 2 rules out of the 74 use relations between two external objects. This makes sense considering that Sprinter only uses the egocentric point of view to make most of its decisions. Notice that each action is at least represented by two rules, with each rule referring to *gl* or *gr*. This is the effect of using constants instead of variables as the predicate arguments, which could otherwise be represented using a single rule with the literal $instanceOf$(B,goal).

CMUnited

Table 29 shows the results for CMUnited for multiple frames of reference. Again, this uses the modified language bias, and we repeated the same experiments with the egocentric frame only in Table 30. As one can see, the modified language bias provides better results compared to Table 20 and much faster search time. The *CovRatio* for both settings remain very low, where with multiple frames it drops below one, meaning that the rules provide worse generalization than the ground facts themselves. Similar to Krislet, we did not see any improvements with the results by using multiple frames of reference.

action(A,dash,fw,close) ←	dirQl(A,self,gl,front), distQl(A,self,gl,close)
action(A,dash,fw,close) ←	dirQl(A,self,gr,front), distQl(A,self,gr,close)
action(A,dash,fw,close) ←	dirQl(A,self,gl,front), distQl(A,self,gl,far)
action(A,dash,fw,close) ←	dirQl(A,self,gr,front), distQl(A,self,gr,far)
action(A,dash,fw,close) ←	dirQl(A,self,gl,front), distQl(A,self,gl,veryfar)
action(A,dash,fw,close) ←	dirQl(A,self,gr,front), distQl(A,self,gr,veryfar)
action(A,turn,farright,same) ←	distQl(A,self,gr,same)
action(A,turn,farright,same) ←	distQl(A,self,gl,same)
action(A,turn,farright,same) ←	dirQl(A,self,gl,farleft)
action(A,turn,farright,same) ←	distQl(A,self,lr,same)
action(A,turn,farright,same) ←	dirQl(A,self,gr,farleft), dirQl(A,gr,lr,farright)
action(A,turn,farright,same) ←	dirQl(A,self,gl,left)
action(A,turn,farright,same) ←	dirQl(A,self,gr,left)
action(A,turn,farright,same) ←	dirQl(A,self,gl,left)
action(A,turn,farright,same) ←	dirQl(A,self,gr,left)
action(A,turn,farright,same) ←	dirQl(A,self,gr,right), dirQl(A,self,p6,right)

Table 28: Example Rules Generated in Multiple-Frames for Sprinter

Search	$f1_{dash}$	$f1_{turn}$	$f1_{kick}$	$f1_{global}$	$Runs_{tot}$	$Time_{avg}$	$Rules_{tot}$	$CovRatio$	$RuleLen_{avg}$
Heuristic	0.188	0 009	0 005	0 027 ±0 004	88/90	3166*	720	0 86	2 62
BF	0.187	0 010	0 003	0 025 ±0.004	89/90	3957*	934	0.57	2.42
DF	0 136	0 009	0.005	0 021 ±0 003	88/90	3056*	714	0.44	2.67

Table 29: Comparisons of Search Methods in Egocentric-Relative-Intrinsic Reference for CMUnited

Search	$f1_{dash}$	$f1_{turn}$	$f1_{kick}$	$f1_{global}$	$Runs_{tot}$	$Time_{avg}$	$Rules_{tot}$	$CovRatio$	$RuleLen_{avg}$
Heuristic	0 317	0 074	0 012	0.060 ±0 011	90/90	2164	480	4 41	3 12
BF	0 325	0 094	0 013	0 066 ±0 011	90/90	874	542	4 13	3 03
DF	0 315	0.072	0 012	0 059 ±0 010	90/90	2840	475	4 25	3 10

Table 30: Repeated Experiments in Egocentric Reference for CMUnited with Constant Language Bias

Summary

In this section, we performed experiments using spatial representations in egocentric, relative and intrinsic frames of reference. We realized that a change in the language bias was needed to accommodate the exponentially increasing search space, and this was done by fixing the definitions of the *dirQl/distQl* predicates to have constant parameters instead of output variables. Not only this reduced search time dramatically, the results also show better accuracies statistically. However, with the three agents that we tested, we found that the representation in multiple frames did not provide any significant improvement. In fact, many times the rules generated were more over-fitting and took longer to find. The results for CMUnited were too poor to provide any meaningful indications.

6.2.5 Modified Krislet (Krislet2)

Intuitively, multiple frames of reference should provide better results given the richer representation. Hence, in order to obtain a deeper understanding, we modified the original Krislet slightly (we call it *Krislet*2) to make use of intrinsic distance between two objects. Specifically, if any player observes that the distance of a teammate to the ball is within the "far" range, it will stop running toward the ball, but turn far-right instead. The intention is that some players can stay behind if another is close enough to the ball already. The original Krislet's simple and predictable behavior makes it easy for us to examine the relevance of the generated rules, thus modifying it in this manner will also give us the same advantage.

Given the results shown in Table 23 and 24, we decided to test with the BF search strategy only in this case to save computing resources. Table 31 compares the results between the two sets of reference frames. We do not report the results for kick as its behavior did not change from the original Krislet.

Search	$f1_{dash}$	$f1_{turn}$	$Runs_{tot}$	$Time_{avg}$	$Rules_{tot}$	$CovRatio$	$RuleLen_{avg}$
Ego-centric Only	0 771 ±0.004	0 886 ±0.025	35/35	1511	281	47 6	3 09
Multiple Frames	0.779 ±0 002	0 950 ±0.002	35/35	1201	322	43 8	2.71

Table 31: Comparisons between Egocentric-Only and Multiple Frames of Reference for Krislet2

The results for dash are significantly lower than that of the original Krislet for both sets of reference frames, which is mostly due to the lower scores for *dash.far* and *dash.veryfar*. Table 32 shows the rules generated in the multiple frames of reference. For both *dash.far* and *dash.veryfar*, the literal $distQl(A, p2, ball1, veryfar)$ was found (in bold). This represents the precise behavior that we are expecting: only run toward the ball when the teammate is very far from the ball (ie. not within the "far" range). However, it does not seem to find rules that run toward the ball when the teammate is completely absent. Instead, other seemingly irrelevant rules with flags and lines were found to fit that behavior. This over-fittingness most likely contributes to the poor performance for these two action parameters. The same issue is shown in Table 33 for the egocentric only frame. In this case, the teammate $p2$ cannot be directly related to the ball, but rather using seemingly irrelevant relations to $self$. This might explain the slight performance degradation as compared to multiple-frames.

Table 32 also shows the rules captured for *turn.farright*. The three most interesting rules are highlighted in bold, which capture exactly the new behavior: agent turns far-right when a teammate is within the "far" range of the ball. The other rules are the same as the original Krislet. On the other hand, Table 33 shows three rules with $p2$, but in a less accurate manner This contributes to a statistically significant decrease in accuracy from the results with multiple-frames. Hence, this provides concrete evidence that multiple-frames do provide advantage over the egocentric-only frame.

Dash Rules	
action(A,dash,fw,veryfar) ←	dirQl(A,self,ball1,front), distQl(A,self,ball1,veryfar), **distQl(A,p2,ball1,veryfar)**
action(A,dash,fw,veryfar) ←	distQl(A,frt,frb,close)
action(A,dash,fw,veryfar) ←	dirQl(A,self,fc,front), distQl(A,self,ball1,veryfar), distQl(A,self,lt,veryfar)
action(A,dash,fw,veryfar) ←	dirQl(A,fcb,frb,left), distQl(A,frt,lr,veryfar), distQl(A,ball1,fct,veryfar)
action(A,dash,fw,veryfar) ←	distQl(A,fc,ball1,same), distQl(A,fct,lr,veryfar), distQl(A,gr,frt,veryfar)
action(A,dash,fw,veryfar) ←	dirQl(A,self,ball1,front), dirQl(A,fc,ball1,left), dirQl(A,fc,gr,right)
action(A,dash,fw,far) ←	dirQl(A,self,ball1,front), distQl(A,self,ball1,far), distQl(A,ball1,gr,far)
action(A,dash,fw,far) ←	dirQl(A,self,ball1,front), distQl(A,self,ball1,far), **distQl(A,p2,ball1,veryfar).**
action(A,dash,fw,far) ←	dirQl(A,self,ball1,front), dirQl(A,self,flt,right), dirQl(A,self,q1,right)
action(A,dash,fw,far) ←	dirQl(A,ll,gl,farleft), dirQl(A,q1,ball1,left), distQl(A,q1,ball1,far), distQl(A,ll,gl,veryfar)
action(A,dash,fw,far) ←	dirQl(A,self,ball1,front), distQl(A,self,ball1,far), **distQl(A,ball1,p2,veryfar)**
action(A,dash,fw,far) ←	distQl(A,self,q1,close), distQl(A,ball1,ll,far), distQl(A,gl,flt,veryfar)
Turn Rules	
action(A,turn,farright,same) ←	not_objectSeen(A,B), instanceOf(B,ball)
action(A,turn,farright,same) ←	**dirQl(A,self,ball1,front), distQl(A,p2,ball1,far)**
action(A,turn,farright,same) ←	**dirQl(A,self,ball1,front), distQl(A,p2,ball1,close)**
action(A,turn,farright,same) ←	**dirQl(A,self,ball1,front), distQl(A,p2,ball1,same)**
action(A,turn,farright,same) ←	dirQl(A,self,ball1,farright), distQl(A,self,flt,veryfar)
action(A,turn,farright,same) ←	dirQl(A,self,ball1,farright), distQl(A,self,ball1,far)

Table 32: Example Rules Generated in Multiple-Frames for Krislet2

6.2.6 Further Discussions

We have also observed from Section 6.2.4 that the language bias plays a big role in determining whether useful rules can be found, and how long it takes for them to be found. By simply changing the arguments in the predicate templates of *dirQl/distQl* from output variables to constants, up to 1000x speed increase was observed. The problem has to do with how ALEPH handles searches of multiple literals with the same predicate names but instantiated with different variable names. In Appendix B, the bottom clause contains many *dirQl/distQl* literals, each having different variable names which refer to different instances of objects. When ALEPH searches the hypothesis space from the empty clause, it creates a candidate clause with each one of these literals separately as if they were all different. Worse

Dash Rules	
action(A,dash,fw,veryfar) ←	dirQl(A,self,p2,left), dirQl(A,self,gl,front), dirQl(A,self,ball1,front), distQl(A,self,ball1,veryfar)
action(A,dash,fw,veryfar) ←	dirQl(A,self,ball1,front), distQl(A,self,ball1,veryfar), distQl(A,self,lt,veryfar)
action(A,dash,fw,veryfar) ←	dirQl(A,self,ball1,front), dirQl(A,self,p2,right), distQl(A,self,fc,far), distQl(A,self,gr,veryfar), distQl(A,self,ball1,veryfar)
action(A,dash,fw,veryfar) ←	dirQl(A,self,lr,farleft), dirQl(A,self,ball1,front), dirQl(A,self,frb,right), distQl(A,self,p2,far)
action(A,dash,fw,far) ←	dirQl(A,self,fc,farleft), dirQl(A,self,ball1,front), distQl(A,self,ball1,far), distQl(A,self,p2,far)
action(A,dash,fw,far) ←	dirQl(A,self,frt,farleft), dirQl(A,self,ball1,front), dirQl(A,self,gr,right), distQl(A,self,ball1,far)
action(A,dash,fw,far) ←	dirQl(A,self,fct,left), dirQl(A,self,fc,front), dirQl(A,self,q1,farright), distQl(A,self,q1,far)
action(A,dash,fw,far) ←	dirQl(A,self,p2,left), dirQl(A,self,q1,right), distQl(A,self,flb,veryfar)
Turn Rules	
action(A,turn,farright,same) ←	not_objectSeen(A,B), instanceOf(B,ball)
action(A,turn,farright,same) ←	dirQl(A,self,ball1,front), distQl(A,self,p2,close)
action(A,turn,farright,same) ←	dirQl(A,self,ball1,front), dirQl(A,self,p2,front)
action(A,turn,farright,same) ←	dirQl(A,self,ball1,front), dirQl(A,self,p2,right), distQl(A,self,ball1,far), distQl(A,self,gl,veryfar)
action(A,turn,farright,same) ←	dirQl(A,self,ball1,farright), distQl(A,self,flt,veryfar)
action(A,turn,farright,same) ←	dirQl(A,self,ball1,farright), distQl(A,self,ball1,far)

Table 33: Example Rules Generated in Ego-centric Frame for Krislet2

yet, the problem is multiplied exponentially when the clause is progressively being refined! Hence, this increases search time exponentially (and unnecessarily), which decreases chances of finding any good clause. The problem will be avoided if the algorithm checks each refinement as to whether it is subsumed by a clause that has already been visited (although subsumption check also adds cost). While CLASSIC'CL does this check, ALEPH does not. This can be a future improvement for the software.

In Section 5.4 1, we outlined the basic algorithm of ALEPH. The algorithm goes through each example one by one, creates a clause from it, and removes all examples being covered before moving onto the next remaining example. Hence, the set of rules generated and the speed of the search depend on the order in

which these examples are presented. Future work can be done to employ a switch in ALEPH that allows random sampling, where examples are picked randomly from the knowledge base instead of in the order they appear. Experiments can be repeated multiple times to collect a set of rules, which can then be compared.

6.2.7 Conclusion

From the experiments we have performed in this Section, one can appreciate the large amount of computing resources required for searching with ILP, especially with a huge search space as in the RoboCup case, where all 22 players, multiple flags, lines, goals and the ball are included. Given that we perform these experiments for multiple agents, multiple parameters per action, and each repeating for 5 game plays, it takes a great deal of time to obtain valid results, let alone the numerous iterations we spent to refine the experiments. Despite of this, we were able to use the ALEPH software to obtain some interesting and relevant clauses in a reasonable amount of time. We also modified the original Krislet with new behavior that allows us to prove that multiple frames of reference can provide benefits over the egocentric-only one. However, the heavy requirement of having precise region boundary values impacts the usefulness of such imitation techniques. In the next section, we shall use CLASSIC'CL to emulate CARMR, which belongs to a different family of ILP algorithms - characteristic induction, where the algorithm attempts to discover *all* underlying frequent patterns.

6.3 CLASSIC'CL

6.3.1 Introduction

This section is dedicated to experiments performed with the CLASSIC'CL software, which we will be using to emulate CARMR. CLASSIC'CL has many settings to adjust the search process, and the ones we plan to perform experiments already mentioned in Section 4. We will first start off by describing some of the limitations in CLASSIC'CL in Section 6.3.2. Then, we shall perform experiments again on the previous three RoboCup Soccer agents. We shall analyze the data and compare with those obtained previously with ALEPH. All experiments were performed using the same machines as before.

6.3.2 Limitations

In Section 5.4.2, we outlined the CARMR's algorithm in CLASSIC'CL. The algorithm does not perform heuristic search based on any evaluation function like ALEPH. Instead, it performs a breadth-first exhaustive search on the hypothesis space, which makes it significantly slower than ALEPH. Since it does not use any beam search technique or maximum node limit to bound the number of open nodes per level, each level gets exponentially bigger as each candidate frequent clause gets further refined. Because of this, we reduced the knowledge base to 1000 examples instead of using all of them. In addition, we made several modifications to the implementation of CLASSIC'CL to improve the search time and quality of the rules:

1. **The head of each candidate clause is restricted to the action predicate alone**: CARMR allows any predicate to appear as the head of the clause. Since our goal is to predict action based on given situations, we always want the head

to be the action predicate.

2. **Search for each action parameter is made in parallel**: By itself, CARMR is a characteristic induction algorithm where it does not require examples to be classified beforehand. This means searching rules for *all* the various (mutually exclusive) action types and parameters will be done altogether within the same computing thread. Splitting up the database into (positive) examples corresponding to each action type and parameter allows them to be searched in parallel. This idea is borrowed from Progol.

3. **Improved minimum confidence (delta-free) check**: Originally, the CLASSIC'CL software searches for all exceptions of a candidate clause (ie. body = true, but head = false) before determining whether it meets the minimum confidence level. We improved the check by terminating the search as soon as the number of these exceeds the minimum specified. We saw at least a 2x speedup, especially when the minimum confidence is set to a high value (95%).

4. **No inductive leap**: We have fixed a problem that was found during our experimentations where even if the body of a candidate clause was false, it was accepted as a valid rule. This led to an inductive leap where rules that were never found true within the examples got included.

These implemented changes can be found in [NMAb]. Unfortunately, they were still not sufficient to speed up the search process significantly: after 12 hours of search, the maximum level (ie. total # of literals in clause) reached was only 6. For practical reasons, we had to terminate them. Therefore, the experiments we could perform with CLASSIC'CL were severely constrained by this factor. Given the results we have found in the previous section, we also decided to forego

experimentations with using multiple reference frames in this section, as we do not expect to find much improvement with this algorithm.

6.3.3 Language Bias

The language bias, as shown in Figure 52, is very similar to the one defined in Figure 45, except for some syntactical differences. We have added the *class*/2 predicate which defines the class of the action to be searched for. The algorithm is modified to look for this predicate and only considers the candidate clause if the predicate is true. This predicate is defined differently for each action type and parameter. For example, the definition:

```
class(K,+) :- key(K), action(K,turn,left,same).
```

is specified in the background knowledge whenever we want to perform search for the action *turn.left*. CARMR orders the predicates lexicographically using the predicate *rorder*/1, which lists all the predicates allowed within a candidate clause in the order they should appear. This helps eliminate redundant clauses. Care must be taken here that the predicates with output variables must appear before the ones that accept those variables as inputs.

From the experiments in Section 6.2, we found that having arguments for the *dirQl/distQl* predicates as output variables made search time much longer, and that was due to ALEPH revisiting clauses with the same literals multiple times because of its inability to detect whether the current clause (or its generalizations) has already been visited. CLASSIC'CL behaves differently from ALEPH in that it relies heavily on θ-subsumption to determine whether the current candidate clause is subsumed by a clause that has already been visited and determined either being infrequent, or already met the minimum confidence (ie. infrequent delta) so that the current candidate can be eliminated. This is used to significantly prune

```
- consult('$CLASSIC_IMISYS/imisys_classicc1 pl')

rtype(1,class(+k,#class))

rtype(*,key(-k))
rtype(1,not_objectSeen(+k,-object))
rtype(*,dirQl(+k,-object,-object,-dir))
rtype(*,distQl(+k,-object,-object,-dist))

rtype(*,instanceOf(+object,#concept))
rtype(*,eq_obj_const(+object,#object))
rtype(*,eq_obj_dirQl_O3(+dir,#dir))
rtype(*,eq_obj_distQl_O3(+dist,#dist))

rtype(*,inteam(+object,#team))
rtype(*,hasUnum(+object,#unum))

rorder([key/1, dirQl/4, distQl/4, not_objectSeen/2, instanceOf/2, eq_obj_const/2, eq_obj_dirQl_O3/2, eq_obj_distQl_O3/2,
inteam/2, hasUnum/2])
```

Figure 52: Language Bias for RoboCup (CLASSIC'CL version)

the search space. Indeed, we performed some experiments and found that with constant arguments, the candidate clauses are very specific such that they cannot be pruned by visited clauses. Hence they ran much slower than the same experiments performed with variable arguments. Therefore, the following experiments will always be using variable arguments for *dirQl/distQl* predicates.

6.3.4 Results

Since we do not vary any parameter in CLASSIC'CL, we simply show results for all three agents in Table 34. The same metrics that were used for ALEPH are also used here for evaluation. For all experiments, results from 5 game plays were again collected for each agent, using region boundary values as in Figure 41. Each of them was still running after 12 hours and were manually terminated at that point.

The $RuleLen_{avg}$ is always less than 6 since this is the maximum level all searches

Agent	$f1_{dash}$	$f1_{turn}$	$f1_{kick}$	$f1_{global}$	$Runs_{tot}$	$Rules_{tot}$	$CovRatio$	$RuleLen_{avg}$
Krislet	0.210	0.589	0 044	0 267 ±0 071	60/60	8228	0.43	4 96
Sprinter	0 539	0 373	N/A	0 429 ±0 083	33/33	17265	0 14	4 65
CMUnited	0 280	0.169	0 004	0.071 ±0 019	90/90	6133	0.39	4.91

Table 34: Experiment Results using CLASSIC'CL for All three agents

could visit before forced termination. Hence it is not a fair comparison with ALEPH. The *CovRatio* shows a very low value, indicating the rules are highly over-fitting. Indeed, with minimum support of 2, a lot of rules that were accepted have very low coverage. This is different from ALEPH (Progol) where every search was biased toward picking clauses with higher coverage (compression or coverage function), even with the same minimum support requirement. As a result, we found that many negative examples were being covered by the generated rules (ie. false positives), significantly reducing the f1 scores compared to those of ALEPH for all action types on the three agents In fact, we found that the number of false positives are even larger than the true positives (not shown). Table 35 lists some of the rules generated for each agent.

The last column indicates the number of positive examples covered in the training data by the clause. Many rules have a minimum coverage of 2. Such highly over-fitting rules are not useful for generalization. Of course, we can post-process these rules by sorting them according to the compression function, and eliminate the ones with low coverage. The large amount of generated rules also lengthens the time it take to process them, which impacts run-time performance.

For Krislet, no rule was even found for *dash.close*. This is due to the misalignment in boundary values discussed before, which requires more literals in the rule to make it more fitting to avoid the conflict. But since the maximum level reached was 6, that was not enough to generate any valid rules. Hence, since the algorithm searched in breadth-first manner, the problem was magnified here.

Agent	Example Rules Generated	Coverage
Kıslet	action(A,turn,left,same) ← key(A), dırQl(A,B,C,D), dıstQl(A,E,F,G), ınstanceOf(F,ball), eq_obj_const(C,fct), eq_obj_dırQl_O3(D,front)	2
	action(A,turn,left,same) ← key(A), dırQl(A,B,C,D), dıstQl(A,E,F,G), ınstanceOf(C,ball), eq_obj_const(F,p1), eq_obj_dırQl_O3(D,left)	70
	action(A,turn,farrıght,same) ← key(A), not_objectSeen(A,B), eq_obj_const(B,ball1)	278
	action(A,turn,farrıght,same) ← key(A), dıstQl(A,B,C,D), eq_obj_const(C,p3), eq_obj_dıstQl_O3(D,veryfar)	2
	action(A,dash,fw,far) ← key(A) ,dırQl(A,B,C,D), dırQl(A,E,F,G), ınstanceOf(C,ball), eq_obj_const(F,fcb) ,eq_obj_dırQl_O3(D,front)	21
	action(A,dash,fw,veryfar) ← key(A), dırQl(A,B,C,D), dırQl(A,E,F,G), eq_obj_const(C,fcb), eq_obj_const(F,fct)	3
Sprınter	action(A,turn,rıght,same) ← key(A), dırQl(A,B,C,D), ınstanceOf(C,goal), eq_obj_dırQl_O3(D,rıght)	66
	action(A,turn,rıght,same) ← key(A), dırQl(A,B,C,D), eq_obj_const(C,p7), eq_obj_dırQl_O3(D,farrıght)	2
	action(A,turn,rıght,same) ← key(A), dırQl(A,B,C,D), eq_obj_dırQl_O3(D,left), hasUnum(C,10)	2
	action(A,turn,rıght,same) ← key(A), dırQl(A,B,C,D), not_objectSeen(A,E), eq_obj_const(E,fc), eq_obj_dırQl_O3(D,farrıght), hasUnum(C,7)	2
	action(A,dash,fw,close) ← key(A), dırQl(A,B,C,D), dıstQl(A,E,F,G), eq_obj_const(F,gl), eq_obj_dırQl_O3(D,front), eq_obj_dıstQl_O3(G,close)	104
	action(A,dash,fw,close) ← key(A), dırQl(A,B,C,D), dırQl(A,E,F,G), eq_obj_const(C,p2), eq_obj_const(F,p3), eq_obj_dırQl_O3(D,rıght)	4
CMUnıted	action(A,kıck,farleft,far) ← key(A), dırQl(A,B,C,D), dırQl(A,E,F,G), eq_obj_const(C,p8), eq_obj_const(F,frt), eq_obj_dırQl_O3(G,left)	2
	action(A,turn,farrıght,same) ← key(A), dırQl(A,B,C,D), dıstQl(A,E,F,G), eq_obj_const(C,ll), eq_obj_dıstQl_O3(G,veryfar), hasUnum(F,4)	2
	action(A,turn,left,same) ← key(A), dırQl(A,B,C,D), dıstQl(A,E,F,G), eq_obj_const(C,q9), eq_obj_const(F,p2), eq_obj_dırQl_O3(D,left)	2
	action(A,dash,fw,close) ← key(A), dırQl(A,B,C,D), not_objectSeen(A,E), eq_obj_const(C,q7), eq_obj_const(E,ball1), eq_obj_dırQl_O3(D,left)	9

Table 35: Rules Generated by CLASSIC'CL for the three agents

For Sprinter, the first rule matched exactly the behavior of Sprinter's *turn.right*, covering 66 positive examples. Unfortunately, there were also many other low-coverage rules generated as shown, again causing negative examples to be covered. Similarly, for *dash.close*, a rule was found with coverage of 104, but performance again suffered due to other low-coverage rules.

For CMUnited, most of the rules have low coverage, as expected. Many of them rely heavily on players, flags and lines, which makes sense considering the complicated behavior of the agent. We expect that results should improve somewhat

when velocities, accelerations, head directions, sequential predicates and more attributes are added to the language, although the problem with unaligned region boundary values will still limit the improvements.

6.3.5 Further Analysis

Length of Experiments

CARMR takes a lot of time in searching the entire space for the small minimum support of 2, whereas Progol always tries to find the best clauses favoring higher coverage. It is not an easy task to justify what is considered to be an optimal support percentage or value [ZR04], as it highly depends on the number of examples available. A minimum support of 2 means all ground facts themselves are excluded but everything else is included. However, it also allows for too many frequent patterns to be generated, thus increasing run-time. Picking too large a support risks missing out on valid rules that could be useful. We did some experiments with a minimum support of 20, and results showed that while the rules are less over-fitting in general, some could not be found at all for a few action parameters since those number of examples are too small. However, the search was faster as there were fewer frequent patterns to be found.

Although Lattner's *MiTemp* algorithm was based on WARMR, there are several differences that makes the search time for his experiments much shorter. First, the examples are modeled with temporal predicates where an argument specifies the range of cycles in which the predicate is true (see Figure 24). Our approach specifies the truth of each predicate in only a single time cycle, hence even if it is true in multiple cycles, the predicate will appear multiple times in the database. This significantly increase the number of ground facts (over 150k vs 2000 in Lattner's)

to be processed by each query. Indeed, we found that CLASSIC'CL spent the majority of its time in computing the frequencies (coverage) of each candidate clause, which takes more time when there are more facts present. Hence, a sequential representation has the advantage of compacting the knowledge base dramatically, reducing the search time required. This is worth future investigation. Secondly, *MiTemp* randomly picks only 200 candidate clauses at each level above the 3_{rd} instead of searching exhaustively, pruning the search space significantly. It ran the tests multiple times to smooth the effects of randomness. We could add this feature to CLASSIC'CL as a future extension.

Boundary Issues

In this section, we see that CARMR suffers the same problem with unaligned region boundary values. This is expected since the major difference with ALEPH is simply an exhaustive search on the hypothesis space without removing covered examples. With ALEPH, we attempted to cover the search space from three directions by using BF, DF, heuristic search, where we were able to obtain interesting and relevant rules without too much over-fitting as in CARMR. In addition, ALEPH offers a task called *induce_max*, where unlike the task *induce* that we used in our experiments, *induce_max* does not remove the examples already covered per iteration. In a sense, this can be used to mimic the behavior of CARMR.

6.3.6 Conclusion

From the experiments we performed in this section, we can realize that characteristic induction using CARMR takes far more time to search than Progol, if heuristic or random sampling techniques are not used to cut down the hypothesis space. More importantly, because the algorithm simply discovers all clauses that meet the

minimum support and confidence requirement, we obtained a lot of rules that have low coverage, thus highly over-fitting. Progol, on the other hand, looks for the best clauses while taking their coverage into account, thus producing clauses that are more relevant. The over-fitting nature of the rules results in many false positives. Despite this, we did obtain rules that represent the true behavior of the agents, although they were also generated by Progol. Therefore, from the discussions here, we shall only use the rules found by ALEPH for the real-time experiments presented in the next section.

6.4 Real-Time Performance

6.4.1 Introduction

In this section, we shall put the rules that we have generated to the real test by applying them in RoboCup games. In summary, we shall perform the following experiments. Each of them were recorded as videos and can be found on [NMAb]. The discussions in this section can be observed in the videos.

1. Krislet: Ego-centric frame; Ego-centric frame with unaligned region boundary values; Multiple frames.

2. Sprinter: Ego-centric frame; Multiple frames.

3. CMUnited: Ego-centric frame; Multiple frames.

4. Krislet2: Ego-centric frame; Multiple frames

The rules discovered by each search method (ie. BF, DF, heuristic) for each action are first combined into one list, and then sorted according to the compression function, thus taking into account coverage of positive and negative examples, as well

as the rule length. Another scoring function such as that used by Lattner [Lat07] can also be used, but we shall leave that as future work. The reason that the rules need to be sorted is that with Prolog as the inference engine, backward chaining automatically returns the first answer that satisfies a query, regardless of how many rules are present that will satisfy it. Therefore, the rules must be ordered before execution such that the preferred ones appear first.

One of our biggest concerns is the run-time performance of using the rules in real-time. As discussed in Section 5.7.2, our software agent is implemented with Java and connected to the XSB Prolog through JNI. The real-time performance of answering a query by Prolog (or any other means) is a crucial element in determining whether ILP can be a feasible solution for behavior imitation.

In our initial experiment, we used the rules generated by the language bias where *dirQl/distQl* use output variables in their parameters. The result was that each query took a widely varying amount of time, from 10's of milliseconds up to 9 seconds! Obviously, this is hardly acceptable for a real-time application. Such wide variation in query time does not look linear, and this led us to try the language bias where *dirQl/distQl* use constants in their parameters. The results were surprisingly good: tested with around 6000 queries and using *all* rules found, each query took merely 6ms on average to obtain an answer, far below the default 100ms requirement for a RoboCup agent to send a decision back to the server! This immediately allowed us to explain the phenomenon: with variables in the direction and distance predicates, Prolog works much harder (many more backtracking iterations) to find ground facts that satisfy all the predicates within a rule, and the time it took depends on how many literals were present in the rules. With the parameters being constants, it took Prolog much less time to determine whether a predicate succeeds or not, without any backtracking at all. Hence, in the rest of this section, we shall only adopt rules that were obtained with the second language

bias option.

The combined rules from the different search method will have many redundancies, and some may be more general than others. Ideally they should be eliminated to achieve a more compact set of rules, but we shall leave this as future work as well.

Another problem we face is that with the qualitative spatial representation in our rules, we need to convert them back into numerical values for the action parameters before sending to the RoboCup server. Currently, we simply hard-code these to be the minimum values within each discretized region, as listed in Table 36.

Action Qualitative Parameter	*Converted Numerical Value*
angle farleft	-23.0
angle.left	-1.0
angle front	0 0
angle right	1 0
angle farright	23.0
power veryfar	300.0
power far	50.0
power close	10.0
power.same	1.0

Table 36: Discrete Values used for Action Parameters

For each game, each player is placed randomly on the soccer field at the start. Because the rules generated were not perfect representations of the true behavior, there are many instances where no answer can be found for a given situation. When this happens, we randomly pick an action with random parameters, so that at least the agent can still function and does not get stuck forever in some state.

6.4.2 Krislet

In our first attempt, we simply used two (equal) apprentice agents, one on each side to play the game. There are a total of 600 rules for the ego-centric frame. When the game was started, both agents turned to look for the ball, and when seen, they both dashed toward it, just like a Krislet agent. Their movements were a little jerky since, as mentioned, only the values in Table 36 are used. They slowed down significantly once they got closer to the ball because only *power.close* was used, which will give them a disadvantage when playing against other fast moving players. Once they actually reached the ball, they started kicking it toward the direction of the left goal (because the game plays used for learning come from players on the right side).

A problem happened from time to time when the agents got close to the ball: they were stuck where they tried to kick the ball continuously when they were not even close enough yet. Through some debugging, we found that one of these two rules was returned:

- action(A,kick,left,far,c235) ← dirQl(A,self,gl,left), dirQl(A,self,ball1,front), distQl(A,self,ball1,close), not_objectSeen(A,B), eq_obj_const(B,p1).
- action(A,kick,left,far,c252) ← dirQl(A,self,ball1,farright), dirQl(A,self,ll,farright), distQl(A,self,ball1,close), distQl(A,self,gl,veryfar), not_objectSeen(A,B), eq_obj_const(B,p2).

The last argument in each action predicate identifies the clause number. Both rules state that when the ball is close (among other criteria), the kick action should be performed. The scores of these rules are 0 and -1 respectively, indicating that they are very low in the priority list. The best rule for kick with a score of 35 is as follows:

- action(A,kick,left,far) ← dirQl(A,self,gl,left), distQl(A,self,ball1,same).

The problem is that this rule cannot be picked because the agent was still in the "close" region before it can reach "same". However, the close region includes distances from 1.0 to 5.0, and before the agent can reach the distance of 1.0 where it can actually touch the ball, the two rules above were picked already at a further distance where they kept trying to kick the ball aimlessly. This is the effect of having incorrect boundary values in discretization regions as we have seen before in Section 6.2.3. With just one minor misalignment in the boundary values from the true behavior, the effect became very visible. Hence, we anticipate even worse results in the next section where we use unaligned boundary values.

In the second attempt, again we used only two players, and we had a total of 1269 rules generated based on unaligned boundary values. We saw the same issue as previously indicated, but worse. When the game was started, player 1 simply was stuck on the field, while the second one managed to get close to the ball and eventually successfully kicked it after multiple attempts. For player 1, the problem was that it kept turning left toward the ball until the angle of its view to the ball changed from -4.0 to -2.0, which means the region was changed from "close" to "front" according to the unaligned boundary values. There is no rule for *turn.left* that is associated with the situation where the ball is in front. However, multiple rules with ball in front are associated with other actions, where many players are involved in the literals. Since in the game we only had two players, and the scene was not changing for player 1, it could not find any rules that satisfy the current situation, and therefore was stuck.

In the third experiment, we used 908 rules generated based on multiple frames (where the region boundary values are the same as the first). Similar to the first experiment, the agents managed to find the ball and dashed toward it, kicked it a

few times and got stuck again. As expected, the rules with the additional relative and intrinsic frames do not seem to make any difference for Krislet.

Although the generated rules seem to allow the agent to perform fairly well, the success highly depends on the correctness of the discretized region boundary values. Even with a minor misalignment, the kick action was not performed at the appropriate time, hence the ball was not even touched. Nevertheless, the visual results correlate quite well with our quantitative analysis in Section 6.2, where we did expect *dash.close* and *kick* to be performed poorly. These experiments indeed reinforce the importance of having appropriate boundary values.

Since there are many rules within the rule list that involve more objects like players and flags which were generated by ALEPH when it tried to resolve the conflicts due to unaligned boundary values, it might provide better performance if we add more objects on the field, so that these rules can be used to satisfy more queries. We performed one last experiment where we put two original Krislet agents on one team, and 5 apprentice agents on the other. The result again showed that the 5 agents tracked the ball very well, and were even able to keep up with the Krislet agents' pace. From time to time, it can be seen that the players tried to kick the ball, but mostly without success, and hence the Krislet team was able to score all the points.

Next we shall look at how the Sprinter apprentice performs. We do expect similar results as Krislet.

6.4.3 Sprinter

For Sprinter, there are a total of 222 rules combined in ego-centric frame, and 285 rules in multiple frames. The first thing that we noticed was that once the agent found the goal, it simply dashed toward it continuously for a long period of time

(ie. Sprinter's normal behavior). Hence, we must set a large number of cycles to avoid the program from thinking the agent was stuck. In general, both agents using different frames can run back and forth on the field. While the one in ego-centric frame only tracked the goal very well - heading straight toward either goal when seen, the one in multi-frame wandered off a little bit, but eventually found its way back toward the goal. We found that this is due to more rules being found for the latter agent, with the score of the rules being better in general for the first agent. This is in turn because of the larger hypothesis space available to be searched when the predicate definitions of *dirQl/distQl* are enhanced with multiple frames of reference.

Although the results look quite good, we have to emphasize again that this will be made worse if the region boundary values are made arbitrary and unaligned with the teacher's true behavior.

6.4.4 CMUnited

Given that the original CMUnited team consists of sophisticated soccer players, we do not expect the apprentices to perform well at all, with the quantitative results that we have seen from previous sections. There are a total of 1497 rules combined in ego-centric frame, and 2368 rules in multiple frames. Figure 53 lists the top five rules generated by ALEPH for CMUnited. One can see that most of the rules depend on players (q8 to q10). Therefore, in our experiments we included 11 (Krislet) opponents so that some of these rules could hopefully be activated. We also included 2 Krislet agents as teammates (#5 and #6) so the opponents would not score too quickly, which allowed more time for our four CMUnited apprentice agents (#1 to #4) to react.

Since the true behavior of CMUnited is not well-known to us, we ran the game

```
% Score = 34, File = cmunited dash fw close rules
action(A,dash,fw,close,c1) ← dirQl(A,self,fct,right), dirQl(A,self,q10,farright), not_objectSeen(A,B), instanceOf(B,ball)

% Score = 34, File = cmunited dash fw close rules
action(A,dash,fw,close,c2) ← dirQl(A,self,q11,right), dirQl(A,self,q8,farright), distQl(A,self,q9,far)

% Score = 34, File = cmunited dash fw close rules
action(A,dash,fw,close,c3) ← dirQl(A,self,fct,right), dirQl(A,self,q10,farright), not_objectSeen(A,B), instanceOf(B,ball)

% Score = 31, File = cmunited dash fw close rules
action(A,dash,fw,close,c4) ← dirQl(A,self,fct,right), dirQl(A,self,q10,farright), distQl(A,self,q10,far)

% Score = 28, File = cmunited dash fw close rules
action(A,dash,fw,close,c5) ← distQl(A,self,fc,far), distQl(A,self,q8,far), distQl(A,self,frt,veryfar)
```

Figure 53: Top 5 Rules generated for CMUnited

3 times for each set of reference frames to make sure that what we saw was not by chance. For the imitating agents equipped with rules learned from the ego-centric frame only, in general they showed some tendency to head toward the ball, but still a lot of the time they wandered aimlessly around the field. Many times the agents were stuck in an infinite loop of kicking and did not come out from it. Interestingly though, at around time 405 (in game 1), agent #1 changed its dashing direction toward the incoming ball, indicating that at least it recognized the relevance of the ball.

With multiple frames, the CMUnited apprentices seem to perform rationally. First, the agents did not get stuck infinitely for all 3 games, and right off the start (in game 1) all four of them dashed toward the ball. Most interestingly, at around time 240, when the ball was kicked by the opponent team from the right side of the field back toward the left, 3 of the agents immediately turned around and started retreating, even before the ball could have reached them. This looked like a defensive play, which could also be observed in other games as well. Then at around time 300, they turned around again and started to go for the ball again.

However, we noticed that for the games using the egocentric reference only, about 13% of all actions on average are randomly picked due to no rule found, while it was 27% for the games using multiple frames. This correlates with the quantitative results we have observed previously, where the higher percentage of rules not being found could be attributed to the fact that the rules are more over-fitting as shown in Table 29, compared to Table 30. Coincidentally, this also allows the agent not to be stuck, since it performs many random actions to cause its surrounding environment to alter. Therefore, this made it hard to judge whether the qualitative observations above could lead to any definite conclusion. We tried to eliminate random action picking from the games, but the agents from both sets then got stuck very early in the game, making them unusable. Given the poor results, we believe that future works should capture more attributes and sequential representations, in order to see some interesting results.

6.4.5 Krislet2

With the ego-centric only frame, the two Krislet2 agents run toward the ball just like Krislet. However, most of the time they either run very close to the ball regardless of the other teammate's position, or they both get stuck from a far position, even when one of the player is closer to the ball and does not see its teammate. This can be seen at around time 230 and 760. In general, it does not imitate Krislet2 very well.

With the multiple-frame, although the above problems mentioned still exist, it could be observed that one of the agents did pause time and again when it saw its teammate getting closer to the ball, with the most noticeable period from time 850 to 900. We also recorded all actions performed by the agents and found that the rules in Figure 54 have been executed throughout the game. This confirms that

these intrinsic distance rules are indeed useful The agent can be seen imitating Krislet2 better than with the egocentric-only frame.

```
% Score = 197, File = krislet2 turn farright same rules action(A,turn,farright,same,c305) ← dirQl(A,self,ball1,front), distQl(A,p2,ball1,far)

% Score = 49, File = krislet2 turn farright same rules action(A,turn,farright,same,c321) ← dirQl(A,self,ball1,front), distQl(A,p2,ball1,close)

% Score = 79, File = krislet2 turn farright same rules action(A,turn,farright,same,c311) ← dirQl(A,self,ball1,front), dirQl(A,self,p2,front)
```

Figure 54: Intrinsic Distance rules executed by Kriset2 agents

6.4.6 Conclusion

In this Section, we performed real-time experiments using the rules on real agents. Both Krislet and Sprinter performed fairly well, but we emphasized that poor choice of region boundary values could significantly impact its performance. For CMUnited, the quality of the rules learned are simply too low to allow us to have any meaningful conclusions. We also showed that Krislet2 performed better using intrinsic distance rules, suggesting that multiple-frames do provide value. In the next Section, we shall compare both qualitative and quantitative results to our CBR work [FEL08].

6.5 Comparison with CBR

6.5.1 Introduction

In this section, we shall compare our ILP results presented before with the CBR results [FE08], both quantitatively and qualitatively. We will also look at the differences and similarities between CBR and ILP.

6.5.2 Quantitative Comparison

Table 37 compares the best CBR results with the best ILP results we have obtained in this thesis. Strictly speaking, the results by ILP cannot be compared in a one-to-one fashion with those by CBR. This is because, as mentioned before, the ILP experiments use a fixed set of boundary values that are known to match the behavior of Krislet and Sprinter. This means the results are biased in favor of these two agents. As Table 21 shows, the results can get much worse when the boundary values are unaligned with the intrinsic agent's behavior. The results for CMUnited probably represent a better comparison, since it most likely does not use the same discretization as Krislet and Sprinter. The work on CBR [FE08], however, does not require discretization and hence the results are not biased.

Nevertheless, the ILP results do not show a statistically significant (p=0.01) increase over CBR results for Krislet. The low $f1_{turn}$ for Sprinter ILP, as mentioned before, is due to the fact that $turn.farleft$ has too few examples to learn from, resulting in a 0 score for this action parameter. This also differs from the CBR results where action parameters were not taken into account when calculating the f1 scores. If we take $f1_{turn}$ out of the calculation, $f1_{global}$ for Sprinter turns out to be 0.95 ±0.008, again not showing a statistically significant increase over the CBR results. These results show that if the optimal boundary values can be found, learning with ILP in RoboCup can indeed be a very promising solution, as much as CBR.

As for CMUnited, ILP results show a far worse performance than CBR. The reason is that CBR uses similarity measure to find the closest matching case. Even if the match is not a very close one, it nevertheless leads to a solution that will have a chance of success. In contrary, with ILP rules will simply not be generated at all if the given language bias is too limited, or minimum support and confidence cannot

be met. Hence, the results deteriorate rapidly.

Agent	$f1_{dash}$	$f1_{turn}$	$f1_{kick}$	$f1_{global}$	# *Cases/Rules*
Krislet - CBR (Table 8 3 - FR-P)	0 96	0 96	0 84	0.92 ±0 007	32284
Krislet - ILP (Table 24 - BF)	0.97	0.98	0.89	0.94 ±0.006	178
Sprinter - CBR (Table 8.1 - FR-P)	0.95	0 93	N/A	0 94 ±0.003	36897
Sprinter - ILP (Table 27 - BF)	1.0	0.681	N/A	0.817 ±0.118	74
CMUnited - CBR (Table 8.5 - FR-P)	0.80	0.72	0 84	0.92 ±0.007	17773
CMUnited - ILP (Table 30 - BF)	0.33	0 09	0 01	0.07 ±0 011	542

Table 37: Comparisons of CBR and ILP Results

We also compare the number of rules and cases in the table. Obviously, the rules generated using ILP provide significant compression on the examples over CBR. This is a major advantage for ILP over CBR, as it reduces computation time for real-time performance, and we have already seen this in Section 6 4. Of course, that depends on design choices of the actual implementation. The smaller number of rules compared to the cases also allows for less disk space usage, and it makes it easier for humans to examine for deeper understanding.

6.5.3 Qualitative Comparison

In Section 6.4 we observed that Krislet and Sprinter perform quite well with strong resemblance to the original agents (again, given that the discretized region boundary values are aligned). They also perform their actions very quickly without any jitter, thanks to the quick average response time of finding a solution by Prolog. However, they require the assistance of random action generation in cases where they get stuck in an infinite loop, or when no matching rule can be found. In the case of CBR agents, they also tend to get stuck from time to time, until something (eg. the ball) moves in front of them. For Krislet, the CBR agent also tries to kick the ball when it is actually outside the kickable radius, similar to the problem we

saw with the ILP agent due to unaligned boundary values. This happens in the CBR case because the agent simply uses the action parameter found in the closest matching case, but does not try to adapt to the current situation.

CMUnited, on the other hand, performs far more unpredictably, with the agent being stuck in loops at times and occasional dashes toward the ball. A high percentage of the actions performed are generated randomly due to no matching rule found. This kind of unpredictable behavior could also be seen in the CBR case, where the agent sometimes turns in circles or wanders elsewhere. We conclude that more attributes are needed in both approaches to represent CMUnited agents, as well as sequential modeling to include memory states and inter-agent communication.

6.5.4 Conclusion

In this section, we observed that an obvious advantage of ILP over CBR is the high compression of the rules on the training examples, which provide very good generalization. On the other hand, CBR does not have the discretization problem which ILP learning suffers from. This problem can be eliminated or minimized by using numerical reasoning in ILP [SCA99, SC97], or various kinds of auto-discretization techniques [FI93, Rat03, KK99, cLW00]. In addition, CBR is always able to generate a solution, regardless of how close it is to the true behavior. As with ILP, a solution may not be found if no matching rule exists for the given situation. This problem can also be alleviated by customizing the inference engine to perform a fuzzy or closest match on the body of the rules, instead of an exact match as done by Prolog.

Despite the differences, we can see many similarities between CBR and ILP. When the raw cases are used as is in CBR without pre-processing, it can be seen as

having ILP searching for rules with a minimum coverage of 1. Hence, there is no compression at all, with every single case becoming a rule itself. When clustering and prototyping is used, this can also be seen as analogous to the generalization performed by ILP, where a single rule covers multiple examples. Also, with CBR feature extraction is used as a pre-processing step to assign each attribute (object) with a specific weight. This has been found [FE08] to improve the imitation results. The same step can be incorporated within the ILP search, where each object can be assigned a preference value (weight), and the evaluation function can be modified to take this into account, so that it can bias toward candidate rules having attributes with larger weights. These similarities lead us to believe that the two systems can complement each other to overcome the shortcomings of each, and improve on the overall quality of the imitation.

Chapter 7

Conclusions and Future Work

This thesis showed that first-order logic rules learned by an imitative agent in the RoboCup simulated soccer domain using inductive logic programming techniques can provide promising imitative results, superior generalization (high example compression) and excellent real-time response through the Prolog inference engine. The research question *"Can ILP be used to effectively learn the behavior of RoboCup agents?"* was addressed by examining two families of ILP algorithms with various search settings, introducing new spatial reference frames in background knowledge, and by varying the language bias.

The remainder of this chapter will provide a summary of the contributions and results presented in this thesis as well as outlining the limitations of the research and future areas of work.

7.1 Summary of Contributions and Results

The following key contributions and results were presented in this thesis:

- **Literature review:** A description of the state of the art in imitation, learning techniques with ILP, spatial ontology and relation systems was compiled. It was found that although imitation had been successfully used in a variety of

domains, these approaches often used extensive expert knowledge and often focused on specific actions or strategies. Although CBR can be used to imitate the complete behavior of an agent and requires minimal expert knowledge, its real-time performance is limited by the large amount of cases required, even after techniques such as clustering and prototyping are used. In addition, it is difficult to understand the underlying behavior of the agent by examining the cases alone. In order to overcome this, ILP learning techniques were examined to determine if they would present a feasible alternate solution to CBR while providing better real-time performance.

- **Discriminatory ILP:** The ALEPH software was used to emulate Progol, a popular discriminative inductive algorithm. It was found that depending on how the language bias was set up, search time can range from seconds to several days. This also affects how expressive the generated rules are. We compared three different search methods (breadth-first, heuristic, depth-first) and found that while heuristic search provides the best overall results, breadth-first search provides the most compact rules (ie. shortest) with the highest coverage ratio. The rules are found to be highly reflective of how the original agent behaves, at least for Krislet and Sprinter.

- **Characteristic ILP:** The CLASSIC'CL software was used to emulate CARMR, a characteristic inductive algorithm. We found that while many rules generated are very similar to the ones found by ALEPH, due to its characteristic nature the algorithm generates a lot more over-fitting rules that simply cover the minimum support required. This creates a lot of rules that are less useful and adds to the burden of processing them. Therefore, we determined that the discriminatory setting of ILP is preferable in our domain, especially when

the application has real-time constraints. We did make a number of modifications and improvements to CLASSIC'CL, such as eliminating problem with inductive leap and improved check on minimum confidence. The changes can be found in [NMAb].

- **Various teams:** The ILP learning algorithms were applied to three different teams, each of which behaves in a different manner. Sprinter and Krislet were both successfully imitated (with limitations - see next section). However, the imitator of the complex CMUnited team behaved noticeably differently from the original.

- **Background Knowledge:** We used background knowledge to translate direction/distance predicates from the egocentric frame of reference to relative and intrinsic frames, directly using Prolog. This feature is extremely powerful in that such translations take place on-the-fly during learning, without using any external pre-processing jobs. We found that the extra reference frames did not provide any advantage to Sprinter and Krislet, as expected. In general, the new rules were more over-fitting than before as observed through the coverage ratio, since the modified language bias included relational predicates between any object pairs, even though they may be completely irrelevant. However, using the modified Krislet2, it could be observed that intrinsic distance rules provide better results qualitatively and quantitatively, proving our intuition that using the extra reference frames provide better modeling for more complicated players.

- **Improved real-time performance:** Using Prolog as the inference engine, along with a particular setup (constant arguments in *dirQl/distQl* predicates) of the language bias, we discovered that the response time of matching the current situation to any rule within the database was extremely fast (6ms on

average), far below the 100ms default requirement in the RoboCup Simulation League. We tested this with rule base sizes ranging from 200 rules up to over 2000 rules, with no difference in response time seen. It should also be stressed that compared to the large amount of CBR cases required, over 100 times fewer rules are needed to replicate the same behavior as the original agent. This should not be surprising since the original Krislet and Sprinter were implemented with only a handful of rules.

- **Software framework:** The implemented software framework provides the ability to deploy imitative software agents in a variety of domains. The framework only requires the definition of what types of sensory stimuli the agent can receive from the environment as well as the actions the agent can perform through the use of background knowledge. The remainder of the learning process can be performed without any human intervention (with limitations - see next section). Source code for the framework, miscellaneous scripts to perform ILP search and results processing, data sets and videos can be found at [NMAb].

7.2 Limitations and Future Work

Although this research shows some promising results in the ability of an agent to imitate another by using ILP, there are a number of areas that are open to further improvements and exploration. Possible future research topics relate to both overcoming the limitations of this work and addressing topics beyond the scope of this thesis. Several notable areas of future work are listed:

- **Region discretization boundary values:** We have emphasized through the

sections that the choice of boundary values has a huge influence on the imitation results. In order to obtain accurate boundary values, numerical reasoning in ILP can be performed [SCA99, SC97], in which comparison predicates (eg. $>$, $\leq$, etc.) can be used as part of the language bias to produce ranges on variables (eg. $X > 2.0$, $X \leq 5.3$). However, the search time can be significantly increased. Alternatively, discretization techniques such as [FI93, Rat03, cLW00] can be used to estimate boundary values through statistical analysis.

- **More visual/non-visual stimuli:** The current work only considers a subset of visual stimuli that an agent receives, its field of vision, and does not take any other data into account. Future work should include more attributes such as velocity, acceleration, stamina, current game score, inter-communication between agents, to name a few. Further studies are necessary to determine the impact of including such information especially in the case of CMUnited or similarly complex teams that do make use of such stimuli.

- **Temporal Reasoning:** Current framework only supports state-less agent where the action is based only on the current frame. Many agents, such as CMUnited, are memory-based and perform actions taking into account what they have seen or experienced in the past frames. The modeling of each frame should therefore take into account the temporal aspects of the environment, as done in Lattner [Lat07]. This will not only improve imitation for memory-based agents, it will also decrease the number of ground facts required to represent all the frames, as discussed in Section 6.3.

- **Rules post-processing:** The rules can be made more compact by replacing multiple existing rules by a single general rule that θ-subsumes them. The Prolog program *T−Reduce* [TRe] may help with this task. In addition, multiple rules that differ only by some constant parameters can be replaced by variables

instead. Performing this replacement allows direct quantitative mapping from a parameter appearing in body literal to an action parameter. This can reduce the total number of rules required, although it may also cause Prolog's query response time to deteriorate, as discussed in Section 6.4. We want to point out that, these general rules could be directly found by Progol if the language bias is modified accordingly.

- **Action Selection:** Currently, the Prolog inference engine returns the first rule that satisfies the given situation. A different inference engine can be used that would return multiple closest matching rules, similar to what we have done in CBR. The best action can then be selected based on a set of criteria, such as support, confidence, lengths of the rules, predicate preferences, etc. These are some of the criteria employed by Lattner [Lat07].

- **Combination of CBR with ILP:** For each rule found by ILP, we can back-annotate it with all the cases from CBR that satisfy the rule. In execution mode, these rules and cases can be loaded together. The current incoming scene can then be matched in parallel to both the rules generated by ILP, and to the cases by CBR. The best matching cases can then be compared to the back-annotated cases associated with the matching rules, to determine which cases are matched by both techniques. These cases can be considered as the absolute best matches, and selected for execution. On top of this, since the cases contain numerical action parameters, they can be used directly (or adaptively) without converting from qualitative parameters to numerical values.

- **Improvements to CLASSIC'CL and ALEPH:** The problem of over-fitting in CARMR can be reduced by using χ_2 pruning [ZR04] instead of minimum support. ALEPH can be improved by checking redundant candidates during

search as in CARMR.

- **Other domains:** In order to create an imitation framework that is domain independent, a software framework was designed and implemented in such a way that it had no knowledge of the environment it was being used in or what type of behavior it would perform. Also, RoboCup agents with a variety of goals and behaviors were used as test cases. This design and experimentation, however, is no substitute for actually deploying imitation agents in multiple domains. Future work will involve imitating agents in other simulated environments, imitating physical robots and imitating humans. Not only will this test the domain independence of the software framework, but it will also open the door to new challenges such as computer vision and action recognition.

List of References

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[All90] James F. Allen. Maintaining knowledge about temporal intervals. In *Readings in qualitative reasoning about physical systems*, pages 361–372, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[AP94] Agnar Aamodt and Enric Plaza. Case-based reasoning; foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, 1994.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[Aut] Open Source Multiple Authors. Xsb prolog. `http://xsb.sourceforge.net/`, last accessed: Nov 30, 2010.

[BDR98] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101:285–297, May 1998. 10.1016/S0004-3702(98)00034-4.

[BR97] Hendrik Blockeel and Luc De Raedt. Raedt. lookahead and discretization in ilp. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, pages 77–85. Springer-Verlag, 1997.

[BSar] Petrakis E. Batsakis S. Integrated ontologies for spatial scene descriptions. In S. M Hazarika, editor, *Qualitative Spatio-Temporal Representation and Reasoning: Trends and Future Directions*. IGI Global, 2010 (to appear).

[Cat91] Jason Catlett. On changing continuous attributes into ordered discrete attributes. In *EWSL '91. Proceedings of the European Working Session on Machine Learning*, pages 164–178, London, UK, 1991. Springer-Verlag.

[CDFH97] Eliseo Clementini, Paolino Di Felice, and Daniel Hernández. Qualitative representation of positional information. In *Artificial Intelligence*, volume 95, pages 317–356, Essex, UK, September 1997. Elsevier Science Publishers Ltd.

[CFH+02] M. Chen, E. Foroughi, F. Heintz, Z. Huang, S. Kapetanakis, K. Kostiadis, J. Kummeneje, I. Noda, O. Obst, T. P. Riley, Y. Wang Steffens, and X. Yin. Robocup soccer server manual, 2002.

[CH01] A. G. Cohn and S. M. Hazarika. Qualitative spatial representation and reasoning: an overview. *Fundamental Informaticae*, 46(1-2):1–29, 2001.

[cLW00] Marcus christopher Ludl and Gerhard Widmer. Relative unsupervised discretization for association rule mining. In *Principles of Data Mining and Knowledge Discovery, D.A Zighed, H.J. Komorowski and J.M. Zytkow, eds, LNCS 1910*, pages 148–158. Springer-Verlag, 2000.

[cpr] C-progol. `http://www.doc.ic.ac.uk/~shm/Software/`, last accessed: Nov 30, 2010.

[CRCB08] Luiz Celiberto, Carlos Ribeiro, Anna Costa, and Reinaldo Bianchi. Heuristic reinforcement learning applied to robocup simulation agents. In Ubbo Visser, Fernando Ribeiro, Takeshi Ohashi, and Frank Dellaert, editors, *RoboCup 2007: Robot Soccer World Cup XI*, volume 5001 of *Lecture Notes in Computer Science*, pages 220–227. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-68847-1_19.

[DD97] L. Deraedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.

[Dec] Declarativa. Interprolog, a java + prolog interface. `http://www.declarativa.com/interprolog/`, last accessed: Nov 30, 2010.

[Deh99] Luc Dehaspe. Frequent pattern discovery in first-order logic. *AI Communication*, 12:115–117, January 1999.

[DJJ+98] Kurt Driessens, Nico Jacobs, Kurt Driessens Nico Jacobs, Nathalie Cossement, Patrick Monsieurs, and Luc De Raedt. Inductive verification and validation of the kulrot robocup team. In *Proceedings of the second RoboCup Workshop*, pages 135–150. Springer Verlag, 1998.

[DR97] L. Dehaspe and L. De Raedt. Mining association rules in multiple relations. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297, pages 125–132. Springer-Verlag, 1997.

[DRR04] Luc De Raedt and Jan Ramon. Condensed representations for Inductive Logic Programming. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004)*, pages 438–446. AAAI Press, 2004.

[DT98] L. Dehaspe and H. Toivonen. Frequent query discovery: a unifying ilp approach to association rule mining. In *Technical Report CW 258*. K.U.Leuven, March 1998.

[DT99] Luc Dehaspe and Hannu Toivonen. Discovery of frequent datalog patterns. *Data Mining Knowledge Discovery*, 3:7–36, March 1999.

[FDE08] Michael Floyd, Alan Davoust, and Babak Esfandiari. Considerations for real-time spatially-aware case-based reasoning: A case study in robotic soccer imitation. In Klaus-Dieter Althoff, Ralph Bergmann, Mirjam Minor, and Alexandre Hanft, editors, *Advances in Case-Based Reasoning*, volume 5239 of *Lecture Notes in Computer Science*, pages 195–209. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-85502-6_13.

[FE08] Michael W. Floyd and Babak Esfandiari. Improving the performance of a robocup case-based imitation agent through preprocessing of the case base. In *Master's thesis*. Carleton University, December 2008.

[FEL08] Michael W. Floyd, Babak Esfandiari, and Kevin Lam. A case-based reasoning approach to imitating robocup players. In *Proceedings of the Twenty-first International Florida Artificial Intelligence Research Society Conference*, pages 251–256, FLAIRS 2008, Coconut Grove, Florida, USA, May 15-17, 2008.

[FF92] Christian Freksa and Robert Fulton. Temporal reasoning based on semi-intervals. *Artificial Intelligence*, 54:199–227, March 1992.

[FI93] Fayyad and Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the International Joint Conference on Uncertainty in AI*, pages 1022–1027, 1993.

[Fra92] Andrew U. Frank. Qualitative spatial reasoning about distances and directions in geographic space. *Journal of Visual Languages & Computing*, 3(4):343 – 371, 1992.

[Fra96] Andrew U. Frank. Qualitative spatial reasoning: Cardinal directions as an example. *International Journal of Geographical Information Science*, 10:269–290, April 1996.

[Fre92] Christian Freksa. Using orientation information for qualitative spatial reasoning. In *Proceedings of the International Conference GIS - From Space to Territory: Theories and Methods of Spatio-Temporal Reasoning on Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, pages 162–178, London, UK, 1992. Springer-Verlag.

[GRH] GRHolst. Jlog. `http://sourceforge.net/projects/jlogic/`, last accessed: Nov 30, 2010.

[HEF95] Jung-Hong Hong, Max J. Egenhofer, and Andrew U. Frank. On the robustness of qualitative distance- and direction-reasoning. In *Proceedings of Auto-Carto 12*, pages 301–310, 1995.

[HICF95] Daniel Hernndez, Fakultat Fur Informatik, Eliseo Clementini, and Paolino Di Felice. Qualitative distances. In *CiteSeerX - Scientific Literature Digital Library and Search Engine [http://citeseerx.ist.psu.edu/oai2] (United States)*, 1995.

[jav] Javacc tutorial. `http://www.engr.mun.ca/~theo/JavaCC-Tutorial/javacc-tutorial.pdf`, last accessed: Nov 30, 2010.

[Kar] Andreas Karwath. Classic'cl. `http://www.karwath.org/systems/classiccl.html`, last accessed: Nov 30, 2010.

[KK99] Wojciech Kwedlo and Marek Kretowski. An evolutionary algorithm using multivariate discretization for decision rule induction. In *in Principles of Data Mining and Knowledge Discovery*, pages 392–397, 1999.

[Kla98] Roberta L. Klatzky. Allocentric and egocentric spatial representations: Definitions, distinctions, and interconnections. In *Spatial Cognition, An*

Interdisciplinary Approach to Representing and Processing Spatial Knowledge, pages 1–18. Springer, 1998.

[KLS07] Shivaram Kalyanakrishnan, Yaxin Liu, and Peter Stone. Half field offense in robocup soccer: a multiagent reinforcement learning case study. In *Lakemeyer, Sklar, Sorrenti, Takahashi, editors, RoboCup-2006: Robot Soccer World Cup X*, pages 72–85. Springer Verlag, 2007.

[KNS03] Alankar Karol, Bernhard Nebel, and Christopher Stanton. Case based game play in the robocup four-legged league: Part i the theoretical model. In *RoboCup 2003: Robot Soccer World Cup VII*, pages 739–747. Springer Verlag, 2003.

[Lan] Krzysztof Langner. The krislet java client. `http://www.ida.liu.se/~frehe/RoboCup/Libs/libsv5xx.html#Krislet`, last accessed: Nov 30, 2010

[Lat07] Andreas D. Lattner. Temporal pattern mining in dynamic environments. In *Dissertations in Artificial Intelligence*, volume 309, 2007.

[LE05] Kevin Lam and Babak Esfandiari. A scene learning and recognition framework. In *Master's thesis*. Carleton University, August 2005.

[LET06] K. Lam, B. Esfandiari, and D. Tudino. A scene-based imitation framework for robocup clients. In *Proceedings of Modeling Others from Observations (MOO 2006)*. Boston, 2006.

[Lev96] S. C. Levinson. Frames of reference and molyneux's question: Crosslinguistic evidence. In *Language and space, P. Bloom, M. Peterson, L. Nadel, & M. Garrett (Eds.)*, pages 109–169, Cambridge, MA, 1996. MIT press.

[LH04] Andreas D. Lattner and Otthein Herzog. Unsupervised learning of sequential patterns. In *ICDM 2004 Workshop on Temporal Data Mining (TDM'04)*, 2004.

[LH05] Andreas D. Lattner and Otthein Herzog. Mining temporal patterns from relational data. In *Proceedings of LWA'2005*, pages 184–189, 2005.

[Lig93] Grard Ligozat. Qualitative triangulation for spatial reasoning. In Andrew Frank and Irene Campari, editors, *Spatial Information Theory A Theoretical Basis for GIS*, volume 716 of *Lecture Notes in Computer Science*, pages 54–68. Springer Berlin / Heidelberg, 1993.

[LMVH05] Andreas D. Lattner, Andrea Miene, Ubbo Visser, and Otthein Herzog. O.herzog. sequential pattern mining for situation and behavior prediction in simulated robotic soccer. In *9th RoboCup International Symposium*, 2005.

[Lug01] George F. Luger. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 4th edition, 2001.

[Mar04] Paul Marlow. A process and tool-set for the development of an interface agent for use in the robocup environment. In *Master's thesis*. Carleton University, December 2004.

[Maz] Igor A. Maznitsa. Prol. `http://www.igormaznitsa.com/projects/prol/index.html`, last accessed: Nov 30, 2010.

[MB88] Stephen Muggleton and Wray L. Buntine. Machine invention of first order predicates by inverting resolution. In *Machine Learning*, pages 339–352, 1988.

[MC] Raymond J. Mooney and Mary Elaine Califf. The foidl inductive logic programming system. `http://www.cs.utexas.edu/~ml/foidl.html`, last accessed: Nov 30, 2010.

[McA92] David McAllester. First order logic. *CiteSeerX - Scientific Literature Digital Library and Search Engine [http://citeseerx.ist.psu.edu/oai2] (United States)*, 1992.

[MF90] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *Proceedings of the First International Workshop on Algorithmic Learning Theory*, Tokyo, 1990. OHMSHA.

[MF01] Stephen Muggleton and John Firth. Cprogol4.4: a tutorial introduction. In *Inductive Logic Programming and Knowledge Discovery in Databases*, pages 160–188. Springer-Verlag, 2001.

[MIS00] Tohgoroh Matsui, Nobuhiro Inuzuka, and Hirohisa Seki. A proposal for inductive learning agent using first-order logic. In J. Cussens and A. Frisch, editors, *Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming*, pages 180–193, 2000.

[MKKP05] Gerd Mayer, Ulrich Kaufmann, Gerhard Kraetzschmar, and Gnther Palm. Neural robot detection in robocup. In Stefan Wermter, Gnther Palm, and Mark Elshaw, editors, *Biomimetic Neural Learning for Intelligent Robots*, volume 3575 of *Lecture Notes in Computer Science*, pages 349–361. Springer Berlin / Heidelberg, 2005.

[ML01] Donato Malerba and Francesca A. Lisi. An ilp method for spatial association rule mining. In *Working notes of the First Workshop on Multi-Relational Data Mining*, pages 18–29, 2001.

[MLR94] Stephen Muggleton, Luc, and De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19:629–679, 1994.

[MLVH04] Andrea Miene, Andreas D. Lattner, Ubbo Visser, and Otthein Herzog. Dynamic-preserving qualitative motion description for intelligent vehicles. In *Proceedings of the IEEE Intelligent Vehicles Symposium (IV 04)*, pages 642–646, 2004.

[MR08] Reinhard Moratz and Marco Ragni. Qualitative spatial reasoning about relative point position. *Journal of Visual Languages and Computing*, 19(1):75–98, 2008.

[MTBF03] Reinhard Moratz, Thora Tenbrink, John Bateman, and Kerstin Fischer. Spatial knowledge representation for human-robot interaction. In *Spatial cognition III: Routes and navigation, human memory and learning, spatial representation and spatial learning*, pages 263–286, Berlin, Heidelberg, 2003. Springer-Verlag.

[MTG+03] C. Marling, M. Tomko, M. Gillen, D. Alexander, and D. Chelberg. Case-based reasoning for planning and world modeling in the robocup small sized league. *IJCAI Workshop on Issues in Designing Physical Agents for Dynamic Real-Time Environments*, 2003.

[Mug95] Stephen Muggleton. Inverse entailment and progol. *New Generation Computing*, 13:245–286, 1995. 10.1007/BF03037227.

[Mur99] Jan Murray. Robolog koblenz: Complex agent scripts implemented in logic. In *Stefan Sablatng and Stefan Enderle, editors, Proceedings of the Workshop RoboCup during KI'99 in Bonn*, pages 12–25. SFB 527 Report 1999/12, Universitt Ulm, 1999.

[NCW97] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[NMAa] Network management & artificial intelligence lab. `http://sites.google.com/a/nmai.ca/home/research-projects/agent-imitation`, last accessed: Nov 30, 2010.

[NMAb] Network management & artificial intelligence lab - ilp imitation. `http://sites.google.com/a/nmai.ca/home/research-projects/agent-imitation/ILP`, last accessed: Nov 30, 2010.

[PZ87] D. J. Peuquet and Ci-Xiang Zhang. An algorithm to determine the directional relationship between arbitrarily-shaped polygons in the plane. *Pattern Recognition*, 20(1):65–74, 1987.

[QM90] J. R. Quinlan and Jack Mostow. Learning logical definitions from relations. In *Machine Learning*, pages 239–266, 1990.

[Qui86] J. R. Quinlan. Induction of decision trees. In *Machine Learning*, volume 1, pages 81–106, Hingham, MA, USA, March 1986. Kluwer Academic Publishers.

[Qui93] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[Rat03] Chotirat Ratanamahatana. CloNI: Clustering of Sqrt(N)-interval discretization. *4th International Conference on Data Mining Including Building Application for CRM & Competitive Intelligence*, December 2003.

[RCC92] David A. Randell, Zhan Cui, and Anthony G. Cohn. A spatial logic based on regions and connection. In *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (1992)*, pages 165–176, 1992.

[RK06] D.D.M. Ranasinghe and A.S. Karunananda Qualitative reasoning engine for visual scene understanding in cognitive vision systems. In *Information and Automation, 2006. ICIA 2006. International Conference on*, pages 81 –85, December 2006.

[RKR08] D.D.M. Ranasinghe, A.S. Karunananda, and U. Ratnayake. Learning qualitative relations in real world scenes. In *Information and Automation*

for Sustainability, 2008. ICIAFS 2008. 4th International Conference on, pages 413 –418, December 2008.

[RL93] Luc De Raedt and Nada Lavrac. The many faces of inductive logic programming. In *ISMIS '93: Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*, pages 435–449, London, UK, 1993. Springer-Verlag.

[Rob08] Robocup official site, 2008. `http://www.robocup.org`, last accessed: Nov 30, 2010.

[RV00] Patrick Riley and Manuela Veloso. Towards behavior classification: A case study in robotic soccer. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, page 1092. AAAI Press, 2000.

[RVM+06] Raquel Ros, Manuela Veloso, Ramon Lpez De Mntaras, Carles Sierra, and Josep Llus Arcos. Retrieving and reusing game plays for robot soccer. In *Lecture Notes in Artificial Intelligence*, volume 4106, pages 47–61. Springer, 2006.

[SC97] Ashwin Srinivasan and Rui Camacho. Experiments in numerical reasoning with inductive logic programming. In *Journal of Logic Programming*. Oxford University Press, 1997.

[SCA99] Ashwin Srinivasan, Rui Camacho, and R. Campo Alegre. Numerical reasoning with an ilp system capable of lazy evaluation and customised search. *Journal of Logic Programming*, 40:40–185, 1999.

[SNM08] Paulo Santos, Chris Needham, and Derek Magee. Inductive learning spatial attention. *Sba: Controle & Automacao Sociedade Brasileira de Automatica*, 19:316 – 326, September 2008.

[Sri] Ashwin Srinivasan. The aleph manual. `http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/`, last accessed: Nov 30, 2010.

[Ste04] Timo Steffens. Adapting similarity measures to agent types in opponent modelling. In *Proceedings of the Workshop MOO at AAMAS 2004*, pages 125–128, 2004.

[THT02a] R. Thawonmas, J. Hirayama, and F. Takeda. Robocup agent learning from observations with hierarchical multiple decision trees. *Working notes of the FIfth Pacific Rim International Workshop on Multi-Agents (PRIMA2002)*, pages 7–16, August 2002.

[THT02b] Ruck Thawonmas, Junichiro Hirayama, and Fumiaki Takeda. Learning from human decision-making behaviors an application to robocup software agents. In Tim Hendtlass and Moonis Ali, editors, *Developments in Applied Artificial Intelligence*, volume 2358 of *Lecture Notes in Computer Science*, pages 73–78. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-48035-8_14.

[TRe] T-reduce: Theory reduction prolog program. `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/treduce.pl`, last accessed: Nov 30, 2010.

[Uni] Carnegie Mellon University. Cmunited robocup-99 simulator team page. `http://www.cs.cmu.edu/~pstone/RoboCup/CMUnited99-sim.html`, last accessed: Nov 30, 2010.

[VW03] Ubbo Visser and Hans-Georg Weland. Using online learning to analyze the opponents behavior. In Gal A. Kaminka, Pedro U. Lima, and Raul Rojas, editors, *RoboCup 2002: Robot Soccer World Cup VI*, volume 2752 of *Lecture Notes in Computer Science*, pages 78–93. Springer Berlin / Heidelberg, 2003.

[WGL98] Jan Wendler, Pascal Gugenberger, and Mario Lenz. Cbr for dynamic situation assessment in an agent-oriented setting. In *Proceedings of AAAI-98 Workshop on CaseBased Reasoning Integrations, AAAI Technical Report WS-98-15*. AAAI Press, 1998.

[Wie] Jan Wielemaker. Yap prolog user's manual. `http://www.dcc.fc.up.pt/~vsc/Yap/yap.html#index-setof_002f3-2`, last accessed: Nov 30, 2010.

[Wik] Wikipedia. Backward chaining. `http://en.wikipedia.org/wiki/Backward_chaining`, last accessed: Nov 30, 2010.

[Wina] WiniKoff. W-prolog. `http://waitaki.otago.ac.nz/~michael/wp/`, last accessed: Nov 30, 2010.

[Winb] WiniKoff. Yprolog. `http://www.vf.utwente.nl/~schooten/yprolog/`, last accessed: Nov 30, 2010.

[WZ00] Frank Wolter and Michael Zakharyaschev. Spatio-temporal representation and reasoning based on rcc-8. In *Proceedings of the seventh Conference on Principles of Knowledge Representation and Reasoning, KR2000*, pages 3–14. Morgan Kaufmann, 2000.

[ZR04] Albrecht Zimmermann and Luc De Raedt. Corclass: Correlated association rule mining for classification. In *Proceedings DS 2004, 2004*, pages 60–72. Springer, 2004.

Appendix A

ILP Demonstration Example

#	Scene (in FOL)	Objects Seen	Descriptions
1	scene(1) objectSeen(1,p1) dirQuan(1,self,p1,1 0) distQuan(1,self,p1,33 1) actionQuan(1,turn,45 0)	p1	Player turns far-right when no ball is present
2	scene(2) objectSeen(2,p1) dirQuan(2,self,p1,2 0) distQuan(2,self,p1,30 1) objectSeen(2,gr) dirQuan(2,self,gr,-20 0) distQuan(2,self,gr,26 2) objectSeen(2,ball1) dirQuan(2,self,ball1,0 0) distQuan(2,self,ball1,11 0) actionQuan(2,dash,110 0)	p1, gr, ball1	Player dashes towards the ball when the ball is straight in front
3	scene(3) objectSeen(3,p2) dirQuan(3,self,p2,-19 0) distQuan(3,self,p2,4 0) actionQuan(3,turn,45 0)	p2	Player turns far-right when no ball is present
4	scene(4) objectSeen(4,p1) dirQuan(4,self,p1,1 0) distQuan(4,self,p1,24 1) objectSeen(4,ball1) dirQuan(4,self,ball1,0 0) distQuan(4,self,ball1,10 0) actionQuan(4,dash,100 0)	p1, ball1	Player dashes towards the ball when the ball is straight in front
5	scene(5) objectSeen(5,p2) dirQuan(5,self,p2,1 0) distQuan(5,self,p2,33 1) objectSeen(5,gr) dirQuan(5,self,gr,-13 0) distQuan(5,self,gr,23 0) objectSeen(5,ball1) dirQuan(5,self,ball1,-35 0) distQuan(5,self,ball1,15 1) actionQuan(5,turn,-35 0)	p2, gr, ball1	Player turns towards the direction of the ball when it is at an angle
6	scene(6) objectSeen(6,p2) dirQuan(6,self,p2,6 0) distQuan(6,self,p2,18 6) objectSeen(6,ball1) dirQuan(6,self,ball1,-33 0) distQuan(6,self,ball1,20 1) actionQuan(6,turn,-33 0)	p2, ball1	Player turns towards the direction of the ball when it is at an angle
7	scene(7) objectSeen(7,p1) dirQuan(7 self p1,0 0) distQuan(7,self,p1,25 1) objectSeen(7,gr) dirQuan(7,self,gr,10 0) distQuan(7,self,gr,14 2) objectSeen(7,ball1) dirQuan(7,self,ball1,-33 0) distQuan(7,self,ball1,0 5) actionQuan(7,kick,100 0,10 0)	p1, gr, ball1	Player kicks the ball towards the goal when the ball is within kicking distance
8	scene(8) actionQuan(8,turn,45 0)	None	Player turns far-right when no ball is present

Appendix B

Bottom Clause in Egocentric-Only vs. Multiple Frames

Frame of Reference	*Bottom Clause*	*# Literals*
Egocentric	action(A,turn,left,same) ← dirQl(A,B,C,D), dirQl(A,B,E,D), dirQl(A,B,F,D), dirQl(A,B,G,D), dirQl(A,B,H,D), dirQl(A,B,I,D), dirQl(A,B,J,D), , dirQl(A,B,U,V), dirQl(A,B,W,X), dirQl(A,B,Y,X), dirQl(A,B,Z,A1), dirQl(A,B,B1,A1), distQl(A,B,I,C1), distQl(A,B,Y,C1), distQl(A,B,B1,C1), distQl(A,B,K,D1), distQl(A,B,C,D1), distQl(A,B,Z,D1), , distQl(A,B,G,D1), distQl(A,B,R,D1), distQl(A,B,S,D1), distQl(A,B,H,D1), distQl(A,B,T,D1), distQl(A,B,J,D1), not_objectSeen(A,E1), eq_obj_dirQl_O3(L,left), instanceOf(E1,player), instanceOf(E1,object), instanceOf(B1,player), instanceOf(B1,object), instanceOf(Z,flag), instanceOf(Z,object), instanceOf(Y,player), instanceOf(Y,object), instanceOf(W,goal), , eq_obj_const(E1,p2), eq_obj_const(B1,p8), eq_obj_const(Z,flt), eq_obj_const(Y,p6), eq_obj_const(W,gl), eq_obj_const(N,ball1), eq_obj_const(M,flb), , inteam(T,unkn), inteam(S,unkn), inteam(R,unkn), inteam(Q,unkn), , inteam(I,team1), inteam(H,unkn), , hasUnum(E1,2), hasUnum(B1,8), hasUnum(Y,6), hasUnum(I,1)	128
Egocentric + Relative + Intrinsic (Variable)	action(A,turn,left,same) ← dirQl(A,B,C,D), dirQl(A,B,E,D), dirQl(A,B,F,D), dirQl(A,B,G,H), dirQl(A,B,I,H), dirQl(A,B,J,H), dirQl(A,B,K,L), dirQl(A,B,M,L), dirQl(A,B,N,O), dirQl(A,B,P,O), dirQl(A,G,F,D), dirQl(A,N,I,D), . , dirQl(A,P,I,D), dirQl(A,P,J,D), dirQl(A,P,E,D), dirQl(A,P,F,D), dirQl(A,G,C,H), dirQl(A,C,F,H), dirQl(A,K,I,H), dirQl(A,J,C,H), dirQl(A,E,C,H), dirQl(A,M,I,H), dirQl(A,P,K,H), dirQl(A,P,M,H), , dirQl(A,G,K,O), dirQl(A,C,N,O), dirQl(A,C,I,O), dirQl(A,C,K,O), dirQl(A,J,N,O), dirQl(A,J,K,O), dirQl(A,E,N,O), dirQl(A,E,I,O), dirQl(A,E,K,O), dirQl(A,M,N,O), dirQl(A,F,I,O), distQl(A,B,E,R), distQl(A,B,M,R), distQl(A,B,P,R), distQl(A,B,G,S), distQl(A,B,C,S), distQl(A,B,N,S), distQl(A,B,I,S), distQl(A,B,K,S), distQl(A,B,J,S), distQl(A,B,F,S), distQl(A,J,G,T), distQl(A,C,N,R), distQl(A,E,G,R), , distQl(A,G,I,S), distQl(A,G,K,S), distQl(A,G,F,S), distQl(A,C,I,S), distQl(A,C,K,S), distQl(A,C,F,S), distQl(A,N,I,S), distQl(A,N,K,S), distQl(A,N,F,S), distQl(A,K,I,S), distQl(A,K,F,S), distQl(A,J,C,S), , distQl(A,M,K,S), distQl(A,M,F,S), distQl(A,P,G,S), distQl(A,P,C,S), distQl(A,P,N,S), distQl(A,P,I,S), distQl(A,P,K,S), distQl(A,P,J,S), distQl(A,P,F,S), distQl(A,F,I,S), not_objectSeen(A,U), eq_obj_dirQl_O3(H,left), eq_obj_distQl_O3(T,same), instanceOf(U,player), instanceOf(U,object), instanceOf(P,player), , instanceOf(E,object), instanceOf(C,flag), instanceOf(C,object), instanceOf(B,player), instanceOf(B,object), eq_obj_const(U,p2), eq_obj_const(P,p8), eq_obj_const(N,flt), eq_obj_const(M,p6), eq_obj_const(K,gl), eq_obj_const(J,ball1), eq_obj_const(I,flb), , eq_obj_dirQl_O3(D,farleft), eq_obj_distQl_O3(S,veryfar), eq_obj_distQl_O3(R,far), inteam(U,team1), inteam(P,team1), inteam(M,team1), , inteam(E,team1), hasUnum(U,2), hasUnum(P,8), hasUnum(M,6), hasUnum(E,1)	164