Design and develop a General Game Playing System using Answer Set Programming

Zong Hui Guo

Bachelor of Science in Computer Science with Honours
The University of Bath
May 2008

Design and develop a General Game Playing System using Answer Set Programming

Submitted by: Zong Hui Guo

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifcally acknowledged, it is the work of the author.

Signed:

Abstract

General Game Playing (GGP) is the art of designing programs that are capable of playing previously unknown games of a wide variety by being told nothing but the rules of the game. This is in contrast to traditional computer game players like Deep Blue, which are designed for a particular game and can't adapt automatically to modifications of the rules, let alone play completely different games. General Game Playing is intended to foster the development of integrated cognitive information processing technology.

In this project, we build a General Game Playing System in ASP (Answer Set Programming). The system would be able to play arbitrary games, singular or multiplayer, via a intermediate Game Manager. ASP is used as main reasoning and searching module in this project in order to generate the best move for the agent.

Contents

Acknowledgements

I sincerely give my gratitude to the project supervisor Dr Marina De Vos for introducing the basic information about the General Game Playing and Answer Set Programming, and giving useful advices and guidance through out the project.

Chapter 1

Introduction

1.1 General Game Playing (GGP)

General Game Playing (GGP) is an area of research in artificial intelligence that focuses on the representation of games in terms of an abstract language known as Game Description Language (GDL). The abstract nature of that language allows for the development of intelligent agents that without modification can perform competently on games that they have never seen before based on known properties of GDL's uniform and compact syntax.

GGP focuses on discrete, multi-player, deterministic, complete-information games. GDL represents game states by keeping track of the set of propositions that are true of the world (part of the description of tic-tac-toe for instance, might be a proposition used to represent whether or not there is an X in the center cell). GDL represents game dynamics as sets of actions that players can perform. Players are each permitted to make one move per turn, where the legality of a move is a function of the propositions that are true prior to its being made (in tic-tac-toe, a player may place an X in the center square if he is playing the role of X, it is his turn, and the square is currently empty).

Perhaps the most natural representational formalism that might be used to reason about the type of games that can be expressed in GDL is a finite state machine (FSM). However, FSMs have certain properties that make them inappropriate for use by general game playing systems. First, the number of states required to describe an FSM is in general exponential in the number of logical propositions required to encode each of those states. Thus, for all but the simplest games encoded in GDL, the size of the corresponding FSM would be intractably large. Second, FSMs are highly uniform structures; to describe a system, each state must in general associate a truth value to each of the propositions required to exhaustively describe it. Thus, even if a game encoded in GDL were to exhibit substantial independence and decomposability of substructures, it is unclear how that structure might efficiently be uncovered from an exponentially large set of states and a transition function

alone.

Although GDL is an effective tool for communication, it is less useful as a representational framework for reasoning about games. In its place, the Stanford Logic Group has developed a new class of behavioral models, called Propositional Nets (PNs), with which an algorithm has been designed for determining whether games can be decomposed into independent sub-systems that can be reasoned about independently of one another.

Propositional Nets were designed by the Stanford Logic Group(Genesereth and Love, 2006) to address the above shortcomings. A PN is a bipartite graph consisting of nodes used to represent propositions alternating with either logical connectives (AND, OR, or NOT gates) or transitions. The propositions in a PN can be partitioned into three classes: input propositions, whose truth values can be set by agents, base propositions, whose truth values are a function of incoming connections from transitions, and view propositions, whose truth values are a function of incoming connections from logic gates. The state of a PN is the set of truth values associated with its base propositions. Similarly, the dynamics of a PN are defined by the transitions that serve as inputs to those base propositions. State updates are accomplished by setting base propositions to be true if and only if the transitions that they are connected to have an incoming connection from a proposition that is true just prior to that update. Unlike FSMs, PNs are representationally compact; the number of propositions required to encode a system is linear in the number of facts required to represent the state of that system. Furthermore, the graphical structure of PNs makes it a straightforward process to determine the functional relationship between propositions; the consequences of changing the truth value of a proposition can be determined by simply following the outgoing arcs that emanate from it.

If a game expressed in GDL can be translated into a PN, then the topological structure of that PN can be used to perform a computationally efficient analysis of whether or not some of the propositions that it contains are independent of one another. Specifically, if there is no directed path between two propositions, then they can be reasoned about independently of one another, as neither one's truth value can ever affect the other's. Furthermore, if it can be shown that that PN is composed of two or more entirely disjoint subgraphs, then each of those subgraphs can be reasoned about independently of each other, as separate games. The potential computational savings associated with such a discovery are substantial. Consider the case of a game expressed in GDL that involves the simultaneous play of two games, one with m legal moves, and the other with n , both of which last for t moves. The complete search tree for such a game would contain $m^t + n^t$ fringe nodes. However, if such a discovery were made and those games were considered separately from one another, then that search space could be reduced to one of two trees containing a combined fringe nodes.

The results described above suggest that PNs are a representational framework deserving

of continued attention. Future research in the area will include further investigation of the mathematical properties of PNs as well as continued development of algorithms along the lines of the one presented in this summary.

1.2 Answer Set Programming (ASP)

Answer Set Programming (Baral, 2003) has become an appealing tool for declarative problem solving. Unlike the related area of satisfiability checking (Mitchell, 2005), it however lacked a formal framework for describing inferences conducted by ASP solvers, such as the resolution proof theory in SAT (P. Beame and Sabharwal, 2004). This deficiency has led to a great heterogeneity in the description of ASP algorithms and has impeded their formal comparability. We addressed this problem in (Gebser and Schaub, 2006) by introducing a family of tableau calculi (M. D'Agostino and Posegga, 1999) for ASP. The idea is to view answer set computations as derivations in an inference system: A branch in a tableau corresponds to a successful or unsuccessful computation of an answer set, an entire tableau represents a traversal of the search space. This approach provided a uniform proof-theoretic framework for analyzing and comparing different operations, strategies, or even algorithms of ASP solvers. Among others, we related the approaches of existing ASP solvers to appropriate tableau calculi, in the sense that computations of solvers are described by tableaux in corresponding calculi.

ASP is based on Logic programming (LP). Logic programming in the first and wider sense gave rise to a number of implementations, such as those by (Black, 1964), (Slagle, 1965) and (Green, 1969), which were question-answering systems in the spirit of McCarthy's advice-taker. (Foster and Elcock, 1969), on the other hand, was probably the first language to be explicitly developed as an assertional programming language.

Logic programming in the narrower sense can be traced back to debates in the late 1960s and early 1970s about declarative versus procedural representations of knowledge in Artificial Intelligence. Advocates of declarative representations were notably working at Stanford, associated with (McCarthy, 1958), Bertram Raphael and Cordell Green, and in Edinburgh, with J. Alan Robinson (an academic visitor from Syracuse University), (Hayes, 1973)and (Kowalski, 1974). Advocates of procedural representations were mainly centered at MIT, under the leadership of Marvin Minsky and Seymour Papert.

Although it was based on logic, Planner, developed at MIT, was the first language to emerge within this proceduralist paradigm (Hewitt, 1971) The most influential implementation of Planner was the subset of Planner, called Micro-Planner, implemented by Gerry Sussman, Eugene Charniak and Terry Winograd. It was used to implement Winograd's natural-language understanding program SHRDLU, which was a landmark at that time. Planner featured pattern-directed invocation of procedural plans from both assertions and

goals. In order to cope with the very limited memory systems that were available when it was developed, Planner used backtracking control structure so that only one possible computation path had to be stored at a time. From Planner there developed the programming languages QA-4, Popler, Conniver, QLISP, and the concurrent language Ether.

Hayes and Kowalski in Edinburgh tried to reconcile the logic-based declarative approach to knowledge representation with Planner's procedural approach. Hayes developed an equational language in1973, Golux, in which different procedures could be obtained by altering the behavior of the theorem prover. Kowalski, on the other hand, showed how SL-resolution treats implications as goal-reduction procedures. Kowalski collaborated with Colmerauer in Marseille, who developed these ideas in the design and implementation of the programming language Prolog. From Prolog there developed, among others, the programming languages ALF, Fril, G?del, Mercury, Oz, Ciao, Visual Prolog, XSB, and Prolog, as well as a variety of concurrent logic programming languages, (see (Shapiro, 1989) for a survey), constraint logic programming languages and datalog.

In 1997, the Association of Logic Programming bestowed to fifteen recognized researchers in logic programming the title Founders of Logic Programming to recognize them as pioneers in the field. The individuals receiving this honor were: Maurice Bruynooghe (Belgium), Jacques Cohen (USA), Alain Colmerauer (France), Keith Clark (UK), Veronica Dahl (Canada/Argentina), Maarten van Emden (Canada), Herve Gallaire (France), Robert Kowalski (UK), Jack Minker (USA), Fernando Pereira (USA), Luis Moniz Pereira (Portugal), Ray Reiter (Canada), Alan Robinson (USA), Peter Szeredi (Hungary), and David H.D. Warren (UK).

Chapter 2

Literature Survey

Building computer programs that effectively play games, such as chess, has long been a challenge for Artificial Intelligence. Systems developed in the past, such as Deep Blue and Chinook, have showed competitive play against human player. However, they are limited in that the agent can play specified game only. Moreover, it is hard to determine whether their success is depending on the player's game playing technique or due to the programmer's game analysis.

Unlike specified game playing agent, a General Game Playing (GGP) system is one that is capable of playing previously unknown games by being told nothing but the rules of the game. In order to perform well, it requires the integration of several AI technologies, including knowledge representation, theorem proving, reasoning and potentially learning.

2.1 General Game Playing

A General Game Playing System is one that can accept a formal description of an arbitrary game and, without further human interaction, can play the game effectively. This projects seeks to develop a GGP player partially based on answer set programming. All other tools and techniques can be freely chosen. ASP (Answer Set Programming) is an up and coming field in logic programming. Programs consist of a series of logical rules which are then put into a solver, which produces a list of 'possible world views' that are consistent with the information provided by the rules. More information on GGP and support tools can be found on the GGP homepage.

The general game playing scenario adopted for this work is taken from the AAAI General Game Playing competition(Genesereth and Love, 2005). In the competition setup, the Game Manager connects to each player process and sends the game description along with time limits called the start clock and play clock. Players have the duration of the start clock

to analyze the game description before the game begins and the duration of the play clock to choose their moves each turn. The game continues until a terminal state is reached. No human intervention is permitted at any point: the general game player must be a complete and fully autonomous agent.

2.1.1 Game Description Language

For an agent to interpret a game, it must be described in a well-defined language. In the Game Description Language (GDL), used in the competition, games are modeled as state machines. An agent can derive its legal moves, the next state given the moves of all players, and whether or not it has won by applying resolution theorem proving. Part of the description for a game called "Minichess" is shown in Figure 2.1. A GGP agent must be able to play any game, given such a description. We illustrate the features of GDL through this example.

First, GDL declares the game's roles (line 1). "Minichess" has two roles, white and black. Next, the initial state of the game is defined (2-4). Each functional term inside an init relation is true in the initial state. Besides init, none of the tokens in these lines are GDL keywords. The predicates cell, control and step are all game-specific. With the exception of goal values, even the numbers do not have any external meaning. If any of these tokens were to be replaced by a different token throughout, the meaning would not change.

GDL also defines the set of legal moves available to each role through legal rules (5-8). The <= symbol is the reverse implication operator. Tokens beginning with a question mark are variables. The true relation is affirmative if its argument can be satisfied in the current state. The state transition function is defined using the next keyword (9-10). The does predicate is true if the given player selected the given action in the current state. Finally, GDL defines rules to determine when the game state is terminal (15-16). When the game ends, each player receives the reward defined by the game's goal rules.

A game description may define additional relations to simplify the conditions of other rules and support numerical relationships. For instance, the succ relation (11) defines how the game's step counter is incremented, and the nextcol relation (12) orders the columns of the chess board. Identifying such relationships is valuable because they bridge logical and numerical representations.

The language is based on a small set of keywords, that is, symbols which have a predefined meaning. A general game description consists of the following elements.

- The players are described using the keyword *(role ?p)*, e.g., *(role white)*.
- The initial position is axiomatized using the keyword *(init ?f)*, for example, *(init (cell*

a 1 white rook))

- Legal moves are axiomatized with the help of the keywords *(legal ?p ?m)* and *(true ?f)*, where the latter is used to describe features of the current position. An example is given by the following implication:
  (<= (legal white (promote ?u ?x ?p))
  (true (cell ?u 7 white_pawn))
  ...)

- Position updates are axiomatized by a set of formulas which entail all features that hold after a move, relative to the current position and the moves taken by the players. The axioms use the keywords *(next ?f)* and *(does ?p ?m)*, e.g.,
  (<= (next (cell ?x ?y ?p))
  (does ?player (move ?u ?v ?x ?y))
  (true (cell ?u ?v ?p)))

- The end of a game is axiomatized using the keyword terminal, for example:
  (<= terminal checkmate)
  (<= terminal stalemate)
  where checkmate and stalemate are auxiliary, domain-dependent predicates which are defined in terms of the current position, that is, with the help of predicate true.

- Finally, the result of a game is described using the keyword
  (goal ?p ?v), e.g.,
  (<= (goal white 100)
  checkmate (true (control black)))
  (<= (goal black 0)
  checkmate (true (control black)))
  (<= (goal white 50) stalemate)
  (<= (goal black 50) stalemate)
  where the domain-dependent feature *(control ?p)* means that it's player *?p*'s turn.

2.1.2 Automated Domain Analysis

An approach common to most computer game playing systems is heuristic search, in which game states are evaluated based on the contribution of various features. Although these features are typically supplied by the system designer (M. Campbell and Hsu, 2002) and (J. Schaeffer and Treloar, 1992), the general game playing setting prohibits such human involvement, motivating the development of automated methods.

In prior work on automated domain analysis (G. Kuhlmann and Stone, 2006), we developed techniques for generating features automatically from GDL game descriptions. The structures identified during domain analysis are also valuable in domain mapping. One of the most basic structures to look for is a successor relation. This type of relation induces

an ordering over a set of objects. We have already seen two examples in lines 11-12 of Figure2.1 . A major challenge for automated domain analysis in GGP is that, during the competition, the description's non-keyword tokens are scrambled to prevent the use of lexical clues. In a competition setting, the same successor relations may look something like: (tcsh pico cpio) (tcsh cpio grep) ... (tcsh echo ping).

```
1.  (role white) (role black)
2.  (init (cell a 1 b)) (init (cell a 2 b)) (init (cell a 3 b))
3.  (init (cell a 4 bk))  ...  (init (cell d 1 wr))  ...  (init (cell d 4 b))
4.  (init (control white)) (init (step 1))
5.  (<= (legal white (move wk ?u ?v ?x ?y))
6.      (true (control white)) (true (cell ?u ?v wk)) (kingmove ?u ?v ?x ?y)
7.      (true (cell ?x ?y b)) (not (restricted ?x ?y)))
8.  (<= (legal white noop) (true (control black)))
9.  (<= (next (cell ?x ?y ?p)) (does ?player (move ?p ?u ?v ?x ?y)))
10. (<= (next (step ?y)) (true (step ?x)) (succ ?x ?y))
11. (succ 1 2) (succ 2 3) (succ 3 4) (succ 4 5) ... (succ 7 8) (succ 8 9) (succ 9 10)
12. (nextcol a b) (nextcol b c) (nextcol c d)
13. (<= (goal white 100) checkmate)
14. (<= (goal black 100) (not checkmate))
15. (<= terminal (true (step 10)))
16. (<= terminal stuck)
```

Figure 2.1: Game Description Language

Our system can still identify these relations as successors because order is a structural, rather than lexical, property. Based on these relations, the agent identifies additional structures such as a step counter, which is a functional term in the game's state that increments each time step. Our system identifies it by looking for a rule such as the one on line 10 of Figure 2.1.

Another example of a structure that our agent attempts to identify is a board. A board is a two dimensional grid of cells that change state, such as a chess or checkers board. Once a board is identified, the system looks for markers, which are objects that occupy the cells of the board and pieces, special markers that occupy only one cell at a time. In "Minichess", the white rook, wr, and the black king bk are examples of pieces. Games like Othello have only markers.

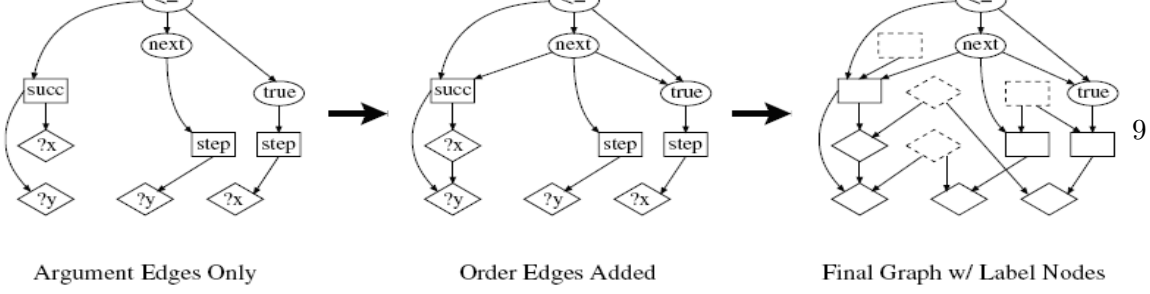


Figure 2.2: Rule graph construction for step counter rule on 10 of Figure 2.1

2.2 Answer Set Programming

Answer Set Programming (ASP) is a new form of declarative programming, oriented towards difficult combinatorial search problems. It has been applied, for instance, to plan generation and product configuration problems in Artificial Intelligence and to graph-theoretic problems arising in VLSI design and in historical linguistics. ASP helps the organizers of computer science conferences assign submissions to referees in accordance with their preferences. Syntactically, ASP programs look like Prolog programs, but the computational mechanisms used in ASP are different: they are based on the ideas that have led to the development of fast satisfiability solvers for propositional logic.

In many applications of Computer Science, especially in combinatorial optimization, hardware design and software correctness checking, complex problems are pervasive. Those problems may be represented in a variety of ways. One technique, which has been used increasingly since the pioneering work of Kautz and Selman on SAT-based planning, is to represent a problem in some logic-based formalism (for instance propositional logic, quantified boolean formulas, or logic programming with the stable semantics) in such a way that intended models correspond to solutions. Subsequently the user applies a solver (dedicated piece of software) to the representation of the problem, computes models and reads off the solutions.

Analogous techniques are used based on a variety of formalisms, such as constraint satisfaction and integer programming, but SAT and ASP are distinguished by being based on logic. While the SAT and ASP research communities apply a similar, logic-based, approach to problem solving, the roots of the two communities are different, and they stress different aspects of the approach. For example, in SAT the emphasis is more toward efficient computation; in ASP more emphasis is placed on knowledge representation. Moreover, there are few researchers who are real experts in both areas.

Platform			Features				Mechanics
Name	OS	Licence	Variables	Function symbols	Explicit sets	Explicit lists	
DLV	Linux,Mac OS,Windows	Freeware	Yes	No	No	No	
Smodels	Linux,Mac OS,Windows	GPL	Yes	No	No	No	

Figure 2.3: Comparison of implementations

2.2.1 History about ASP

The planning method proposed by Dimopoulos, Nebel and Kohler (Y. Dimopoulos and Kohler, 1997) is an early example of answer set programming. Their approach is based on the relationship between plans and stable models described in a paper wrote by (Subrahmanian and Zaniolo, 1995). (Soininen and Niemela, 1998) applied what is now known as answer set programming to the problem of product configuration. The use of answer set solvers for search was identified as a new programming paradigm in (Marek and Truszczynski, 1999)'s paper (the term "answer set programming" was used for the first time as the title of a part of the collection where that paper appeared) and in (Niemela, 1999)'s paper. ASP is a younger field, with approximately 15 years of history, but shows considerable promise on some problems which are poorly handled by SAT. ASP stems of Logic Programming, but changes the paradigm by computing models rather than unifying substitutions. Due to its Logic Programming roots, ASP input languages invariably include schemata, which together with a built-in way to express recursion provides considerable modeling convenience. The built-in recursion mechanism appears particularly useful when encoding problems involving sequences of events, such as hardware and software verification and planning. In contrast to other approaches ASP is the only area of that appears to have taken modeling methodology seriously from the outset.

2.2.2 Answer set programming language Lparse

Lparse is the name of the program that was originally created as a front-end for the answer set solver smodels, and is now used in the same way in many other answer set solvers, including assat, clasp, cmodels, gNt, nomore++ and pbmodels. (dlv is an exception; the syntax of ASP programs written for dlv is somewhat different.)

An Lparse program consists of rules of the form $< head > \leftarrow < body >$. The symbol $\leftarrow$*("if")* is dropped if $< body >$ is empty. The simplest kind of Lparse rules are rules with constraints.

One other useful construct included in this language is choice. For instance, the choice rule $\{p,q,r\}$ says that choose arbitrarily which of the atoms p,q,r to include in the stable model. The Lparse program that contains this choice rule and no other rules has 8 stable models – arbitrary subsets of $\{p,q,r\}$. The definition of a stable model was generalized to programs with choice rules by (P. Simons and Soininen, 2002). According to (Ferraris and Lifschitz, 2005), choice rules can be treated also as abbreviations for propositional formulas under the stable model semantics. For instance, the choice rule above can be viewed as shorthand for the conjunction of three "excluded middle" formulas:

$(p \vee \neg p) \wedge (q \vee \neg q) \wedge (r \vee \neg r)$.

The language of Lparse also allows us to write "constrained" choice rules, such as *1{p,q,r}2*.

This rule says that choose at least 1 of the atoms p,q,r , but not more than 2. The meaning of this rule under the stable model semantics is represented by the propositional formula

$(p \vee \neg p) \wedge (q \vee \neg q) \wedge (r \vee \neg r) \wedge (p \vee q \vee r) \wedge \neg(p \vee q \vee r)$.

Cardinality bounds can be used in the body of a rule as well, for instance:

$\leftarrow 2\{p, q, r\}$.

Adding this constraint to an Lparse program eliminates the stable models that contain at least 2 of the atoms p,q,r. The meaning of this rule can be represented by the propositional formula

$\neg((q \wedge q) \vee (p \wedge r) \vee (q \wedge r))$.

Variables (capitalized, as in Prolog) are used in Lparse to abbreviate collections of rules that follow the same pattern, and also to abbreviate collections of atoms within the same rule. For instance, the Lparse program

$p(a)$.
$p(b)$.
$p(c)$.
$q(X) \leftarrow p(X), X! = a$.

has the same meaning as

$p(a)$.
$p(b)$.
$p(c)$.
$q(b)$.
$q(c)$.

The program

$p(a)$.

$p(b)$.
$p(c)$.
$\{q(X) \leftarrow p(X)\}2$.

is shorthand for

$p(a)$.
$p(b)$.
$p(c)$.
$\{q(a), q(b), q(c)\}2$.

2.2.3 Generating stable models

To find a stable model of the Lparse program stored in file $< filename >$. we use the command $lparse < filename > \| smodels$. Option instructs *smodels* to find all stable models of the program. For instance, if file test contains the rules

$1\{p, q, r\}2$.
$s \leftarrow \neg p$.

then the command *lparse* **test**$\|$ *smodels* produces the output

Answer:1
Stable Model: q,p
Answer:2
Stable Model: p
Answer:3
Stable Model: r,p
Answer:4
Stable Model: q,s
Answer:5
Stable Model: r,s
Answer:6
Stable Model: r,q,s

2.2.4 Examples of ASP programs

Game Setting

An n-setting of a game G is a function Game from its set of vertices to {*1,...,n*} such that *Game(X)* ¬ *Game(Y)* for every pair of adjacent vertices . We would like to use ASP to find an n-setting of a given game (or determine that it does not exist).

This can be accomplished using the following Lparse program:

c(1...n).
1{game(X,I):c(I)}1 ← *v(X).*
← *game(X,I), game(Y,I), e(X,Y), c(I).*

Line 1 defines the numbers *1,...,n* to be colors. According to the choice rule in Line 2, a unique color i should be assigned to each vertex x. The constraint in Line 3 prohibits assigning the same color to vertices x and y if there is an edge connecting them.

If we combine this file with a definition of G, such as

v(1...100). % 1,...,100 are vertices
e(1,55). % there is an edge from 1 to 55

and run smodels on it, with the numeric value of n specified on the command line, then the atoms of the form *Game(...,...)*in the output of smodels will represent an n-coloring of G.

The program in this example illustrates the "generate-and-test" organization that is often found in simple ASP programs. The choice rule describes a set of "potential solutions" – a simple superset of the set of solutions to the given search problem. It is followed by a constraint, which eliminates all potential solutions that are not acceptable. However, the search process employed by smodels and other answer set solvers are not based on trial and error.

Large Clique

A Choosing cycle in a directed game is a cycle that passes through each vertex of the game exactly once. The following Lparse program can be used to find a Choosing cycle in a given directed game if it exists, we assume that is one of the vertices.

$\{in(X,Y) \leftarrow e(X,Y)\}.$
$\leftarrow 2\{in(X,Y) : e(X,Y)\}, v(X).$
$\leftarrow 2\{in(X,Y) : e(X,Y)\}, v(Y).$
$r(X) \leftarrow in(0,x), v(X).$
$r(Y) \leftarrow r(X), int(X,Y), e(X,Y).$
$\leftarrow \neg r(X), v(X).$

The choice rule in Line 1 "generates" all subsets of the set of edges. The three constraints "weed out" the subsets that are not Choosing cycles. The last of them uses the auxiliary predicate *r(X)*("x is reachable from 0") to prohibit the vertices that do not satisfy this condition. This predicate is defined recursively in Lines 4 and 5.

This program is an example of the more general "generate, define and test" organization: it includes the definition of an auxiliary predicate that helps us eliminate all "bad" potential solutions.

2.3 Automatic Heuristic Search Algorithm

General Game Playing is concerned with the development of systems that can play well an arbitrary game solely by being given the rules of the game. This raises a number of issues different from traditional research in game playing, where it is assumed that the rules of a game are known to the programmer. Writing a player for a particular game allows to focus on the design of elaborate, game-specific evaluation functions. But these game computers can't adapt automatically to modifications of the rules, let alone play completely different games.

Systems able to play arbitrary, unknown games can't be given game-specific knowledge. They rather need to be endowed with high-level cognitive abilities such as general strategic thinking and abstract reasoning. This makes General Game Playing a good example of a challenge problem which encompasses a variety of AI research areas including knowledge representation and reasoning, heuristic search, planning, and learning. In this way, General Game Playing also revives some of the hopes that were initially raised for game playing computers as a key to human-level AI.

In this chapter we describe the following functionalities of a complete General Game Player:

1. Determining legal moves and their effects from a formal game description requires reasoning about actions. We use the Fluent Calculus and its Prolog-based implementation FLUX.

2. To search a game tree, we use non-uniform depthfirst search with iterative deepening

and general pruning techniques.

3. Games which cannot be fully searched require automatically constructing evaluation functions from formal ame specifications. We give the details of a method that uses Fuzzy Logic to determine the degree to which a position satisfies the logical description of a winning position.

4. Strategic, goal-oriented play requires to automatically derive game-specific knowledge from the game rules. We present an approach to recognizing structures in game descriptions.

All of these techniques are independent of a particular language for defining games. However, for the examples given in this paper we use the Game Description Language developed by Michael Genesereth and his Stanford Logic Group; the full specification of syntax and semantics can be found at games.stanford.edu. We also refer to a number of different games whose formal rules are available on this website, too. Since 2005 the Stanford Logic Group holds an annual competition for General Game Players. The approach described in this paper has been implemented in the system FLUXPLAYER, which has has won the second AAAI General Game Playing competition.

2.3.1 Survey about Search Algorithm

In our game playing scenario, in which an agent may look ahead by simulating moves, an obvious choice of approach is search. Most existing game playing systems for the types of game that we consider are based upon the Minimax search algorithm. Well-known examples include Chinook for Checkers and Deep Blue for Chess. Even learning-based systems such as TD-gammon incorporate search for action selection. However, unless the state space is small enough to be searched exhaustively, the agent must use a heuristic evaluation function to evaluate non-terminal nodes and thus bound search depth. Such evaluation functions are highly game-specific, and much of the human effort in developing game playing systems is spent on manually tuning them.

In a general game playing setting, the evaluation function cannot be provided by a human. Because each lookahead requires costly theorem proving, exhaustive search is not a viable option. To make reasoning as efficient as we could, our agent uses a Prolog-style interpreter for theorem proving. In early testing, we found that this was significantly faster than other kinds of resolution. Even so, all but the smallest game trees are beyond the reach of exhaustive search.

To overcome these issues, we developed a method for generating heuristic evaluation functions automatically. We construct heuristics from features identified in the game description. Candidate heuristics are evaluated in parallel during action selection to find the best move. We discuss the details of our search algorithm in the next section before moving on

to the heuristic construction algorithm.

2.3.2 Search Algorithm

Players need to look ahead in order to simulate moves, hence we need a good search algorithm to do so. Breadth First Search and Depth First Search are always used for space search, but they are very slow. A more powerful algorithm, Iterative Deepening Depth search, is used more often for complex system due to the following two enhancements: transposition tables (J.Scheaffer, 1989), which cache the value of states encountered previously in the search; and history heuristics (J.Scheaffer, 1989), which re-orders children based on their values found in lower-depth search. (J.Scheaffer, 1989) showed combination of these tow enhancement provides most of the search space cutoff when combined with additional techniques.

(G. Kuhlmann and Stone, 2006) and (Schiffel and Thielscher, 2007) both used Iterative Deepening Search when designing their systems. They combine it with additional pruning technique: alpha-beta-pruning. This algorithm works very well in turn-taking games. In such games, if the move is being made by our player, it is treated as a maximization node; otherwise, it is a minimization node. However, this method does not work in simultaneous moves games. For those games, (G. Kuhlmann and Stone, 2006) assume each player is either teammates or opponents, so the algorithm can still be applied. (Schiffel and Thielscher, 2007) used a paranoid assumption in which they simulate moves of all players and move first.

The search algorithm employed by our player is based on alpha-beta pruning. Without a heuristic, the terminal nodes of the game tree are visited in a depth-first search order. For heuristic search, we use iterative deepening to bound the search depth. We include two general-purpose enhancements: transposition tables, which cache the value of states encountered previously in the search; and the history heuristic, which reorders children based on their values found in lower-depth searches. Other general techniques exist, but the combination of these two enhancements accounts for most of the search space cutoff when combined with additional techniques. We extended this basic minimax search algorithm to allow simultaneous decision games and games with more than two players.

Although our implementation was designed to work for the broadest possible set of games, we made one important simplifying assumption. We assume that each player in the game can be placed into one of two sets: teammates and opponents. Thus, every player is either with us or against us. Our simple but strict definition of a teammate is a player that always receives the same reward that we do. We determine which team a player is on through internal simulation. By treating somewhat cooperative players as opponents, we avoid the need to maintain a separate utility function for each team, which can be expensive since

standard alpha-beta pruning can no longer be used.

Our player uses the same search procedure for turn-taking and simultaneous decision games. The procedure also applies to games with a mix of the two. At a turn-taking node, if the move is being made by a teammate, it is treated as a maximization node. If it is an opponent's turn to move, it is a minimization node. This type of search is an instance of the paranoid algorithm. If it is a simultaneous move node, because of the unpredictability of our opponents, we assume that our opponents choose their moves uniformly at random. We choose our action from the joint action of our team that maximizes our expected reward, based on that assumption. If we have prior knowledge of the opponent, either from past games or earlier moves in the current game, opponent modeling methods could be applied at this point, as could game theoretic reasoning.

2.3.3 Identifying Structures

The first step toward constructing a heuristic evaluation function is to identify useful structures in the game description. Useful structures include time counters, game boards, movable pieces, commodities, etc. Our agent identifies these structures by syntactically matching game description clauses to a set of template patterns. Although the syntax of these templates are specific to GDL, they are conceptually general, and the template matching procedure could be applied to other languages. Still, we include the idiosyncrasies of GDL in our discussion both to make the procedure concrete and to aid future competition participants. Specifically, we describe our agent's identification of five structural game elements: successor relations, counters, boards, markers, pieces, and quantities.

As mentioned above, other than keywords, GDL tokens have no pre-defined meaning to the player. In fact, during the competition, the tokens were scrambled to prevent the use of lexical clues. Therefore, all structures are detected by their syntactic structure alone. For example, one of the most basic structures our agent looks for is a successor relation. This type of relation induces a total ordering over a set of objects.

(G. Kuhlmann and Stone, 2006) identified five structural game element: successor relations, counters, boards, markers and pieces. In their approach, the agent identified the successor relation using structural of the game. A counter is recognized as a function term in the games state that increments each time. Every ternary function in th state is assumed to be a board. Objects that occupy the cells of the board are markers. A piece means that a marker is in only one cell at a time.

(Schiffel and Thielscher, 2007) used a similar approach for their player. They detected successor relation using semantically properties of predicates. The same approach, which

(G. Kuhlmann and Stone, 2006) used to identified the game boards and quantities, was implemented in their player. However, they didn't restrict the boards as two dimensional boards with object on the cell. They consider a game board as a multi-dimensional grid of cells that have a state which can change.

Our system can still identify these relations as successors because order is a structural, rather than lexical, property. When a successor relation is detected, the objects in its domain can be compared to one another and, in some cases, incremented, decremented, added and subtracted. Identifying such structures is very important.

A game description may include several successor relations, which the agent uses to identify additional structures such as a counter. A counter is a functional term in the game's state that increments each time step.

Our game player identifies counters for several reasons. First, if the counter terminates the game in a fixed number of steps, it may be removed during internal simulation to increase the chances of encountering a goal state. More importantly, in some games, leaving the counter in the state representation causes state aliasing, making search inefficient. Therefore, the function is removed before caching a state's value in the transposition table.

Another example of a structure that our player attempts to identify is a board. A board is a two dimensional grid of cells that change state, such as a chess or checkers board. When the agent receives the game description, it assumes every ternary function in the state is a board. However, one of the board's arguments must correspond to the cell's state, which can never have two different values simultaneously -two of its arguments are inputs and the third is an output. We verify this property using internal simulation.

According to our definition of a board, a candidate is rejected if it ever has two objects in the same place. Although such a board may be valid, we choose to eliminate it to prevent false positives. "Minichess" has one board: the cell function. If a board's input arguments are ordered by some successor relation, then we say that the board is ordered.

Once a board is identified, our player looks for additional related structures such as markers, which are objects that occupy the cells of the board. If a marker is in only one cell at a time, we distinguish it as a piece.

Our player also identifies additional structures that do not involve boards or pieces. These structures are functions in the state that may quantify an amount of something, such as money in Monopoly. These are identified as those relations having ordered output arguments. We discuss the distinction between input and output arguments in the next

section.

2.3.4 From Syntactic Structures to Features

The structures identified in the previously described processes suggest several interesting features. We use feature to mean a numerical value, calculated from the game state, that has potential to be correlated with the outcome of the game. If the dimensions of the board are ordered, then our agent computes the x and y coordinates of each piece as the natural numbers associated with the input arguments' indices in their corresponding successor relations. For example, the coordinates of wr in the initial state of the "Minichess" example are (3, 0). From these coordinates, the agent can also calculate the Manhattan distances between pieces.

If a board is not ordered, it may still produce useful features, including the number of occurrences of each marker. In addition, the agent generates features corresponding to the values of each quantifiable amount. (G. Kuhlmann and Stone, 2006) first generated features for identified structures (Table 1). In their approach, a set of candidate heuristics been created, each being either maximization or minimization of a single feature. Finally they linearly mapped the feature's value to an expected reward between $R^{-}+1$ to R^{+}-1, where $R^{-}+1$ and R^{+}-1 are minimum and maximum goal values achievable by the player. In this algorithm, they considered that a definition win is always better then an unknown outcome.

Identified Structure	Generated Features
Ordered Board w/ Piece	Each piece's X coordinate Each piece's Y coordinate Manhattan distance between each pair of pieces Sum of pair-wise Manhattan distance
Board w/o Pieces	Number of markers of each type
Quantity	Amount

Table 2.1: Generated features for identified structures.

2.3.5 From Features to Heuristics

From the set of generated features, our game player creates heuristics to guide search. In traditional single-game playing systems, multiple features are manually weighted and combined to create a single evaluation function. Because the games to be played are not known to the designer, that option is not available to agents in the general game playing scenario. While it may be possible to learn the evaluation function, as is done by TD-gammon, doing this efficiently in the general case remains an open problem.

The idea which (Schiffel and Thielscher, 2007) used for the heuristic evaluation function is to calculate the degree of truth of the goal and terminal formulas in the state to evaluate. They combined the value of goal and terminal in a way such that the terminate state should be avoid as long as the goal is not reached. They used the basic idea of fuzzy logic, i.e. assign values between 0 and 1 to atoms depending on their truth value. Atoms, which correspond to identified structures, will be calculated by using non-binary evaluations. For example, the evaluation of an order relation r is computed as
eval(r(a,b),z)=
t+(1-t)*Δ(a,b)/|dom(r)|, if $<$ r(a,b)
(1-t)-(1-t)*Δ(b,a)/|dom(r)|, if $>$ r(a,b)
where Δ(a,b) is the number of states needed for reaching b from a.
And |dom(r)| denotes the size of the domain of the argument of r.

Instead of constructing a single heuristic function, our agent constructs a set of candidate heuristics, each being the maximization or minimization of a single feature. By including both, the agent can handle games with counterintuitive goal conditions. As it turned out, "Nothello" gave us a great example of such a game during the competition. In "Nothello", the player with the fewest markers at the end of the game wins. Because the agent generates a heuristic to minimize the number of its own markers, it had a candidate heuristic that matched well with the game's goal.

We implement the candidate heuristics as board evaluation functions that linearly map the feature's value to an expected reward between R^-+1 and R^+-1, where R^-+1 and R^+-1 are the minimum and maximum goal values achievable by the player, as described by the goal predicate. The values of the maximizing and minimizing heuristic functions are calculated respectively as follows:
$H(s) = 1 + R^- + (R^+ - R^- - 2) * V(s)$
$H(s) = 1 + R^- + (R^+ - R^- - 2) * (1 - V(s))$
where $H(s)$ is the value of heuristic H in state s and $V(s)$ is the value of the feature, scaled from 0 to 1. We scale the heuristic function in this way so that a definite loss is always worse than any approximated value, and likewise, a definite win is always better than an unknown outcome.

Chapter 3

Requirements

In this section the requirements specification for the General Game Playing System will be discussed. They were derived from interview with supervisor and research on the website.

3.1 Requirements Analysis

The requirement analysis explains how the requirements for each section were considered. Since there was no actual customer for the project, the requirements were derived from the objectives of the project, and the study of similar existing software, together with what the programmer believed would produce a best piece of software.

3.1.1 Function requirements

This section describes the core functionalities of the General Game Playing System. Its main focus is to ensure that the system produced has all required basic functionalities.

Communication

General Game Playing was introduced by the Stanford University Logic Group. They provide a stage so that players can compete with each other. The Game Manager located in the University of Stanford provides the communication channel for all players. Players can only create matches (single player or multi players) via the General Game Playing stage. The description for the match will be sent to player(s) via the game manager.

All messages sent between the game manager and player(s) are using HTTP. Hence the player should be able to support the HTTP, both receive and generating messages.

Understanding GDL

Game description sent by the game manager is in the format of GDL, so as the updated player(s) moves. In order to understand the game, the player is required to understand the GDL first. Or at least would be able to accept the GDL format of game description.

Compiling GDL to ASP

As I mentioned that we are going to use ASP in order to generate the moves. Hence we need a ways to transfer the game description from the GDL into ASP.

Generating Moves

We will generate the move by analysis the game description in ASP. ASP provides a set of model to the given description. We might need to think a heuristic function, which can be applied on the answers in order to get the best move.

3.1.2 Non-function requirements

This section is focused on the general design strategy and usability which have no direct influence on the functionality of the system. The main purpose of this section is to improve the overall quality of the system.

Portability

Windows should not be the only environment in which the player is working. The system should be able to be launched in any operating system (E.g: Windows, Unix).

Robustness

Once the player is started, the process should not be killed without the user's instruction. The System should handle all kind of exception so the player will not be terminated.

Efficiency

In term of efficiency here, I mean the system should not occupy too much disk space. Furthermore, the amount of memory it requires should be as minimum as possible.

3.1.3 Project Constraints

This section outlines the projects constraints. It is very important to mention these in the requirement analysis as these constraints are the most important factor in which whether some features can or can not be implemented.

Time

The Project has to be completed and submitted before 28th Apr 2008.

Human Resources

All tasks are required to be carried out individually, hence only one programmer is available for the entire project, which includes the process of carrying out research of requirements, design and implementation, system testing and documentation write up.

Hardware Resource

A computer is essential in order to carry out the work for this project. The computers in the library at University of Bath, and programmer's own computer will be used as main hardware resources.

Software Resource

Varieties of Java editor programmers are available in the internet. Eclipse is open license software which provides the good interface for Java programming, and plug in to the ASP built by previous University of Bath student projects. Extra software might be used in order to make a good design of the project. Latex provides a good layout when it comes to the documentation write up.

Other Resource

Dr. Marina De Vos will be aside to give advice on any problems that might occur during the project. Magazine and Journal in library is also available for research purpose.

3.2 Requirement Specification

The formal requirement specification is outlined in this section.

The requirements that are essential to the system are identified by the word "must", while the requirements that are not essential are identified as "should". Requirements that are identified by the word "should" might not be implemented in the final system, this might due to the project constraints or any unforeseen difficulties.

3.2.1 Function requirements

1. The system must be able to communicate with the game manager.
   (a) The system must be hosted as a server.
   (b) The system must response when receiving request from the game manager.
2. The system must support HTTP messages.
   (a) The system must be able to accept message using HTTP.
   (b) The system must send all messages using HTTP.
3. The system must accept game description in GDL.
   (a) The system should accept the game description in GDL.
   (b) The system should accept the updated players move in GDL.
4. The system must convert the game description from GDL into ASP.
   (a) The system should be able to parse the GDL.
   (b) The system should be able to generate the game description in ASP from the parsed GDL.
5. The system should use ASP to generate a move.
   (a) The system should support ASP programs.
   (b) The system should get answer from ASP.
   (c) The system should try to apply a heuristic search function on given ASP.

3.2.2 Non-function requirements

1. The system should support cross-platform.
   (a) It should be able to be run both in Windows and Unix.
2. Process should not be terminated without the instruction from user.
   (a) The player should not be killed once started.
3. The system should be efficiency.
   (a) It should not occupy more than 100mb of hard disk space.
   (b) It should use as little memory as possible.

3.2.3 Project Constraints

1. The while project must be complete and submitted by 28th Apr 2008.
2. All tasks must be carried out individually.
   (a) Useful information and advices will be provided by project supervisor Dr. Marina De Vos.
3. Hardware Constraints
   (a) Computers at University of Bath can be used.
   (b) Programmer's own computer can be used.
4. Software Constraints
   (a) Eclipse can be used for programming of Java and ASP.
   (b) Drawing tools(E.g.: Microsoft paint) can be used for design.
   (c) Latex will be used for the documentation write up.

Chapter 4

Design

In this chapter, I will describe the whole system design based on the requirements mentioned in previous chapter. First, I give the frame design in 4.1. Each module will be explained in more details in next few sections.

4.1 Frame Design

Based on the functionalities showed in chapter 3, I am going to use the modularization to break the entire project into three small components.

- Linker
  Linker is used to communicate with the game manager by using HTTP messages.
- Compiler
  Module Compiler acts as an intermediate role between Linker and Reasoner. It translates the Game Description Language into answer set program so the player would be able to recognize the description. Then it translates the generated move into GDL and sends it back to the game manager via the Linker.
- Reasoner
  This module is an important part for the project. It takes the description written in answer set programs, and calls the external stable model (smodels) to calculate all possible moves. A heuristic functions is then be applied on the all generated models in order to get the best move.

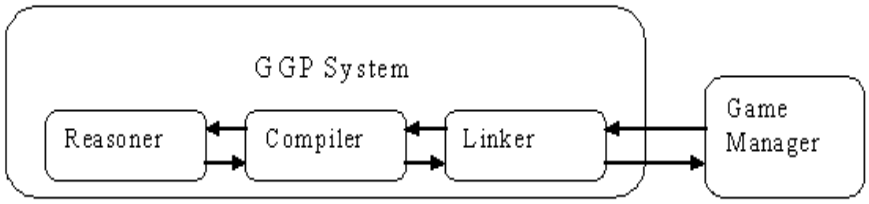


Figure 4.1: Frame of GGP System

4.2 Linker

Linker module is the interface between our player and the game manager. Before we describe how Linker module will be built, I would like to show how the communication is done between the player and the game manager first.

4.2.1 Communication

The game manager communicates with players via message using HTTP. It assumes a player is running on a host listening on a particular port. Message sent to players have the standard HTTP header, with content type text/acl. In the case of games played through the GGP website, the sender will be specified as "Gamemaster", while the receiver will be the remote player name as registered on the web framework. (See Figure 4.2 for an example.)

Figure 4.2: HTTP message example

POST / HTTP/1.0
Accept: text/delim
Sender: GAMEMASTER
Receiver: GAMEPLAYER
Content-type: text/acl
Content-length: 1554

(START MATCH.3316980891 X
(ROLE X) (ROLE O)
...
30 30)

The game manager is the important components in the General Game Playing web framework. Upon receiving a request to a match, a *START* message is sent to each player to initiate the match by the game manager. This message contains the description of the game and other essential information. Once all players are ready for the match or the *startclock* time expires, it sends *PLAY* message to all players to get their moves. The PLAY message includes information about the actions of all players on the previous step. Players should reply within the amount of time specified by the match's *playclcok*. If a player fails to reply within the time limit or the move made is illegal, the game manager selects an arbitrary

legal move for the player. This process repeats until the game is over. Once a game is determined to be finished, a *STOP* message will be sent to each player. This message contains information about the last moves made by all players. This message allows players to clean up any data structure they created for the match.

4.2.2 Linker Design

The Linker module is the interface between the player and the game manager. It receives the request from the game manager and sends the response back to the game manager. The overall view of the Linker's position in the system is shown in (Figure 4.3).

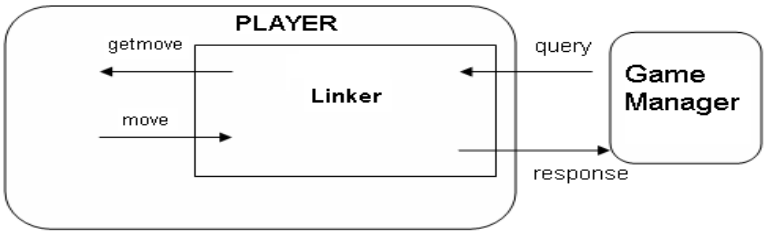


Figure 4.3: Overview of Linker function

We now need to look into the content of the message. The body of the HTTP message consists one of several commands with sets of KIF sentences as arguments. There are three command is used in the KIF sentences: *START, PLAY and STOP.*

- Command START is used to initialize the game. This command provides the information of player's role in the game, the game description and the value of move clocks. It is always in the form:
  $(START < MATCHID >< ROLE >< DESCRIPTION >$
  $< STARTCLOCK >< PLAYCLOCK >)$
- The PLAY command is used to initiate each step of the game. The command takes two arguments, a match ID and the list of actions taken by all players on the previous step. This command is always in this form:
  $(PLAY < MATCHID > (< A1 >< A2 > ... < An >))$
  where each $< Ai >$ is a move made by a player in the last round.
- The STOP command is used to notify the end of the game. It is similar to the PLAY command with the keyword PLAY changes to STOP.

The Linker module is working as this way. Upon receiving the request, it will remove the HTTP header as this is only for communication purpose. It will then work out the command embedded in the body of the HTTP message. Based on the command, different

action will be performed. Figure 4.4 shows insight of the Linker.

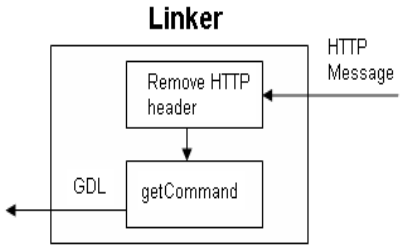


Figure 4.4: Insight into the Linker

4.3 Compiler

As the game description received from the game manager is in GDL format and our objective is to using ASP to generate the models. Hence we need to convent the given description into ASP. This is the main responsibility for the Compiler module.

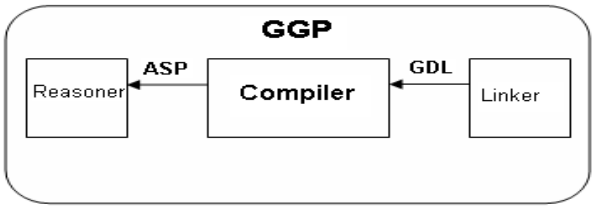


Figure 4.5: Compiler

In the section, we will first look at the syntax of GDL and ASP, then move to introducing a way of converting GDL into ASP.

4.3.1 Game Description Language (GDL)

GDL describes the state of a game world in terms of a set true facts. The transition function between state are described using logical rule in GDL that define the set of facts true in the next state in terms of the current state and the moves of all the players. GDL contains constructs for distinguishing the initial state of the game, as well as the goal states and terminal states. The set of relations used in GDL are: *role, init, true, does, next ,legal, goal* and *terminal*.

GDL sentences are always in this form: *(<= (HEADER) (BODY))*. Where HEADER is an atom or a relation, BODY is a set of rules. Its ground atomic sentence only has the header and looks like this *(HEADER)*.

Players: the *role* relation

The *role* relation defines the players of the game in the game description. This relation is in the form *(role player)*. It only appears in ground atomic sentences. For example, *(role xplayer)* and *(role oplayer)* define two player in Tic-Tac-Toe, referred to in the axioms as *xplayer* and *oplayer*.

Game State: the *true* relation

The current state of a game is represented by facts that take the form of ground functional terms in relational logic. For instance, the ground term *(control xplayer)* in Tic-Tac-Toe is a term. The axioms refer to certain such facts holding in the current state of the game through the true relations: *true(fact)*. The atomic sentences *(true (cell 2 2 b))* indicates the cell at position(2,2) is blank in the current state in Tic-Tac-Toe. This relation is only appearing in the heads of the rules.

Initial State: the *init* relation

The *init* relation indicates all the true facts for the initial state of the game. Therefore, it has the same form as true relation has. For example, *(init (cell 2 2 b))*.

Game State Update: the *next* relation

The *next* relation is an analogue to *true*, referring to facts that will hold in the next state of the game. It appears only in the bodies of the rules.

Moves: the *does* relation

The does relation: *does(player, move)* indicates the moves actually made by players in a particular step of the game. The axioms reference the *does* relation in rules that govern state update. It only appears in the bodyies of the rules like *next* relation.

4.3.2 Answer Set Programming(ASP)

ASP programs are sets of rules of the form:

L ← L1, ... , Lm.
where *Li* are literal.

Body of the rules can be dropped if the L is ground term. For example:

a.
p(a).
q(X) ← p(X).

4.3.3 Compiling GDL into ASP

The game description we receive from the game manager will in GDL, however we would like to generate our move in ASP. Hence we need to work out a method which would be able to translate the given GDL description into ASP. In this section, I will describe how a GDL sentences can be converted into ASP.

We assume GDL is our source program, we intent to generate code for the program in destination language (ASP). Having studied the syntax of both programs, we can easily compile them from on to another.

The ground term in GDL is in the form of *(A1 A2 ... An)*, where As is an atom. We can simple convert it into ASP like this *A1(A2, ... , An)*. This means we take the first argument of the term as a function name in ASP, and put all the remaining atoms as the arguments to the function. Hence, a GDL atomic sentence *(role xplayer)* will become *role(xplayer)* in ASP.

Now we will look at a more complicated example of GDL. As we mentioned sentences in GDL are in the form *(<= H B1 B2 ... Bn)*. Therefore, we can convert it into ASP like this *H ← B1, B2, ..., Bn*. That is, we will take the first term follows the <= symbol as header of our rule, and all remaining terms as body. For example, sentence

(<=
(DIAGONAL ?PLAYER)
(TRUE (CELL 1 3 ?PLAYER))
(TRUE (CELL 2 2 ?PLAYER))
(TRUE (CELL 3 1 ?PLAYER)))

in GDL will be compiled like this in ASP

DIAGONAL(?PLAYER) ←

CELL(1,3,?PLAYER),CELL(2,2,?PLAYER),CELL(3,1,?PLAYER).

4.4 Reasoner

Reasoner is the module we used to generate the move for our players. Module Compiler will convert the game description from GDL into ASP. Hence we can generate the moves in ASP.

In this module, we will call the external programs *lparse* and *smodels* to generate all the models based on the description given. A heuristic search function can then be applied to the returning models in order to generate a move which maximize our benefit.

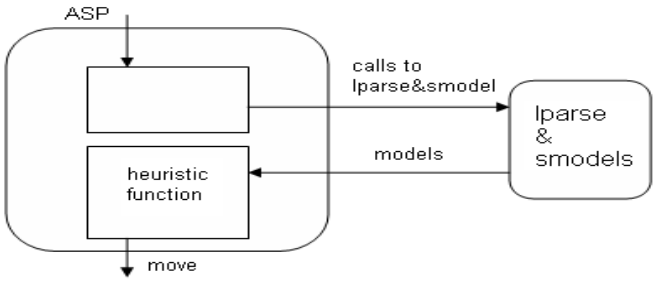


Figure 4.6: Structure of Reasoner Module

Chapter 5

Implementation and Testing

This chapter gives the details of the implementation approaches and algorithms based on the design from the previous chapter. As mentioned in the last chapter, our system will be broken into three components: Linker, Compiler and Reason. I will brief how components are implemented first, component interface and system testing will be discussed in the last section of this chapter.

5.1 Linker Module

Linker module is as the interface between the player and the game manager. The game manager requires the player to be a running HTTP server. Requesting to the player should return the appropriate answers.

This module is implemented as a thread which can be started when receives the command from the user. I created HTTPSession class inside the module. This class is used to handle the HTTP message received and to generate our message in HTTP format. (Figure 1.1 shows the interface of the class)

Listing 5.1: HTTP Class Interface

```
private class HTTPSession implements Runnable
{
        private Socket mySocket;

        public HTTPSession( Socket s);

        public void run();

        private void sendResponse( String status, String mine,
            Properties header, InputStream data);
```

```
};
```

When the module receives the request from the game manager, it will first work what kinds of request is the message. Based on the command of the message, actions will be taken and response will be generated and sent back.

I have mentioned there are three commands will be requested by the game manager, the interface for these command is also listed here. (Figure 5.2)

Listing 5.2: Interface for different command actions

```
private void commandStart(String msg);

private String commandPlay(String msg);

private void commandStop(String msg);
```

CommandPlay function is the only one which requires a return type as String. Because the commandStart function tells the player to setup and get ready for the match. It will pass the game description to the Compiler module and make the player ready for the game. The commandStop function removes all temporary created files so the player would not be affected when the next game is started.

5.2 Compiler Module

The Compiler module is used to compile the given game description in GDL into ASP. This module consists of the parsing phase and code generation. I will describe the implementation of these two phases with the interface the module provides. The testing plan will be briefed in the final section.

5.2.1 Parsing

I broke the GDL program into three categories: des, rules and terms. Term is the small item in the GDL, which can be an atom or a function. Rule is a set of terms or itself a term. I called the entire program as des, which consists of rules and atoms. Hence I created three functions to parse them based on the given statements (Figure 5.3 shows the interface for these functions).

Listing 5.3: Interface for Parse functions

```
static private ArrayList desParser(String des);

static private ArrayList ruleParser(String statement);
```

```
static private ArrayList termsParser(String term);
```

Arraylist instance is used to store the parsed statements, which will then be passed into the code generation for further work. Let's me brief you how these functions are implemented.

The desParser function takes the entire description as input argument. From the syntax analysis of GDL, we know each statement in the description is bounded by the open and close bracket. Hence I used a loop to read each character in the description, increase the bracket count when it is a bracket. I determined that if the number of the open bracket count is the same as the close bracket, a statement is found and stored as a element in the ArrayList. This function returns the ArrayList instance where each element in the list is a statement in GDL.

The ruleParser works similar to the desParser. In ruleParser, the keyword of the statement is generated and stored as the first element of list. All other terms in the statement will be parsed the same way as desParser. Finally, the ArrayList instance, which stores all the terms in the statement, will be sent back to the calling function.

The termsParser function takes an input string as a term. It reads character through the string. Each white space is treated as a break point which separates the atoms. Atoms will be stored in the ArrayList and sent back.

5.2.2 Code Generation

I have created three code generation functions in corresponding to the parsing functions. Figure 5.4 shows the interface of these functions.

Listing 5.4: Interface for Code Generation functions

```
static private String desGenerator(ArrayList statementList);

static private String ruleGenerator(ArrayList ruleList);

static private String termsGenerator(ArrayList atomList);
```

The function takes an ArrayList instance as input argument. The instance might contain the statement or rules based on which function is evoked. I will brief how these functions work in the next few paragraph.

The desGenerator function takes an ArrayList instance which contains the entire game description. It iterates each element in the list in order to work statement by statement. One

of other two functions will be called based on the keyword of the statement. For example, if the keyword is init, the statement will be passed to the termsGenerator function.

Like the parser, the ruleGenerator function works very similar to the desGenerator function. It will first get the keyword of the statement. Different method will be applied based on the keyword. For example, if it gets a <= keyword, it knows the statement is a complicated rule. It tests whether each element (apart from first one) is a term or not. The termsGenerator function will be evoked if the answer is yes, otherwise, it will recursively call itself with the statement as input argument.

The termsGenerator function is the most basic function in code generation phase. It takes the first element in the list as function name, and makes all other elements as the arguments for the function.

5.2.3 Interface

Two interfaces have been created for this module due to the required functionalities of the command action in Linker module (Figure 5.5).

Listing 5.5: Interface of the Compiler module

```
static public void compile(String filename);

static public String getMove(String updateMoves);
```

Function compile is evoked when the Linker receives the START command. Description of the game is stored in a file and the file name is passed to this function for further setup. This function reads the content in the file, and applied parsing and code generation phases on it. The final generated ASP will then be stored in a temporary file.

The getMove function in this module takes the updated player moves as argument. This information will be compiled into ASP before appended to the temporary file. Finally, it calls the reasoner module to generate the move and sends the move back to linker.

5.2.4 Testing

I have created a file which contains rules in GDL(Figure 5.6), the testing result of this case will be showed in next chapter.

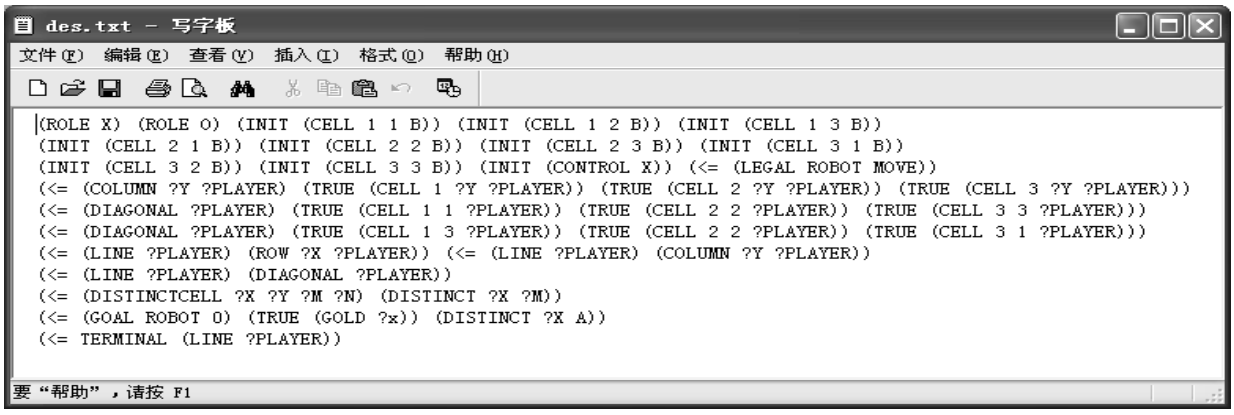


Figure 5.1: Compiler test case

5.3 Reasoner Module

5.3.1 Implementation

The main responsibility of this module is to generate a move for the play. I divided it into two smalls sections, calculating models and searching the best move (See Figure 5.7 for the interface of these functions). Calculating models will take the game description in the ASP and calls the external programs lparse and smodels to generate the models. The output will be collected and a heuristic search function is applied on the models in order to get the best move.

Listing 5.6: Interface of function getModule and generateMove

```
static private String getModels();

static private String generateMove(String models);
```

As I said the compiled file will be stored in the temporary file "model.pl", the getModels function will call the lparse and smodels to generate all answer for the file. I created an Process instance in this function for the external function call, the output of the process will be created and passed to the generateMove function.

The generateMove function will perform a heuristic search algorithm on the returning models, the best move will be selected from the models and sent back to the caller. However, this functionality has not been implemented in this project yet due to the time constraint and the technique issue of the programmer.

Creating instances will occupy the memory from the running environment. If we forget to clear up the memory after the use of the object is finished, we waste the unnecessary

memory. In order to prevent such circumstance, I make all functions in this module as static. The system can easily ask the module to perform certain action without creating an instance of this class. This will certainly reduce the amount of memory used if many matches are played by a launch.

Because the main goal of this module is to produce a move, I provide a function in this module to perform this functionality: static public String getMove(). Many interfaces will always increase the complexity of the system. In order to keep our system clear and easy for future extend, I will only leave this function as the interface to the outside world. However, function getModels and generateMove will be executed in this function as described above.

5.3.2 Testing

Testing result of this module will be shown in this section. Because the generateMove function is not completed at this stage, we can only testing the getModels function.

I created the test main function to make sure the module is working correctly(Figure 5.8). The result of the testing will be displayed in next chapter.

Listing 5.7: Main function for testing the Reasoner module

```
public static void main(String args[]){
        System.out.println(Reasoner.getMove());
}
```

5.4 System Testing

After all components are tested, we would like to test the system as whole to make sure it works as excepted. Let's me brief how the scenario of testing the system looks like. We will first launch our player in any computer which is connected to the internet. Next stage is we go to the General Game Playing web framework to get our player registered. Once our player is registered, we can create match for our player via the web framework.

Chapter 6

Result

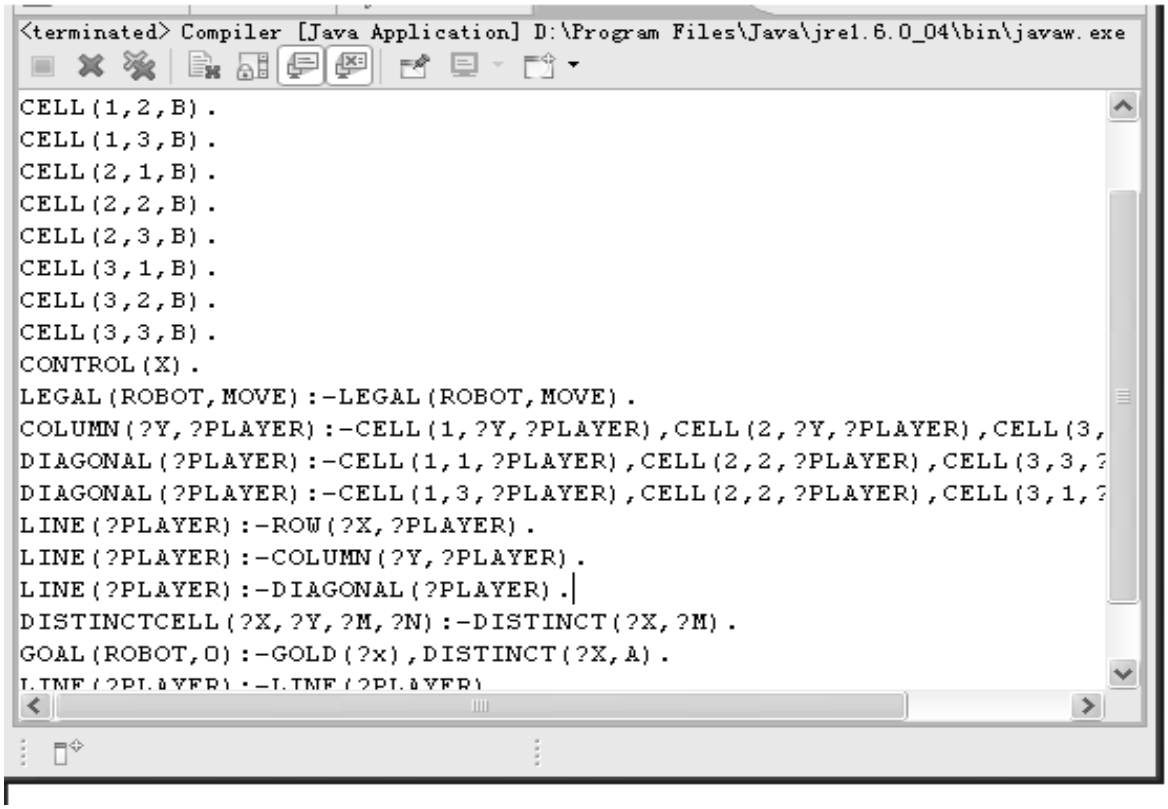
I will show the result for the testing cases mentioned in the chapter 5. The result for each individual component will be briefed before the result of system testing.

6.1 Linker Module

Linker module is an abstract object whose responsibility is to communicate with the game manager.

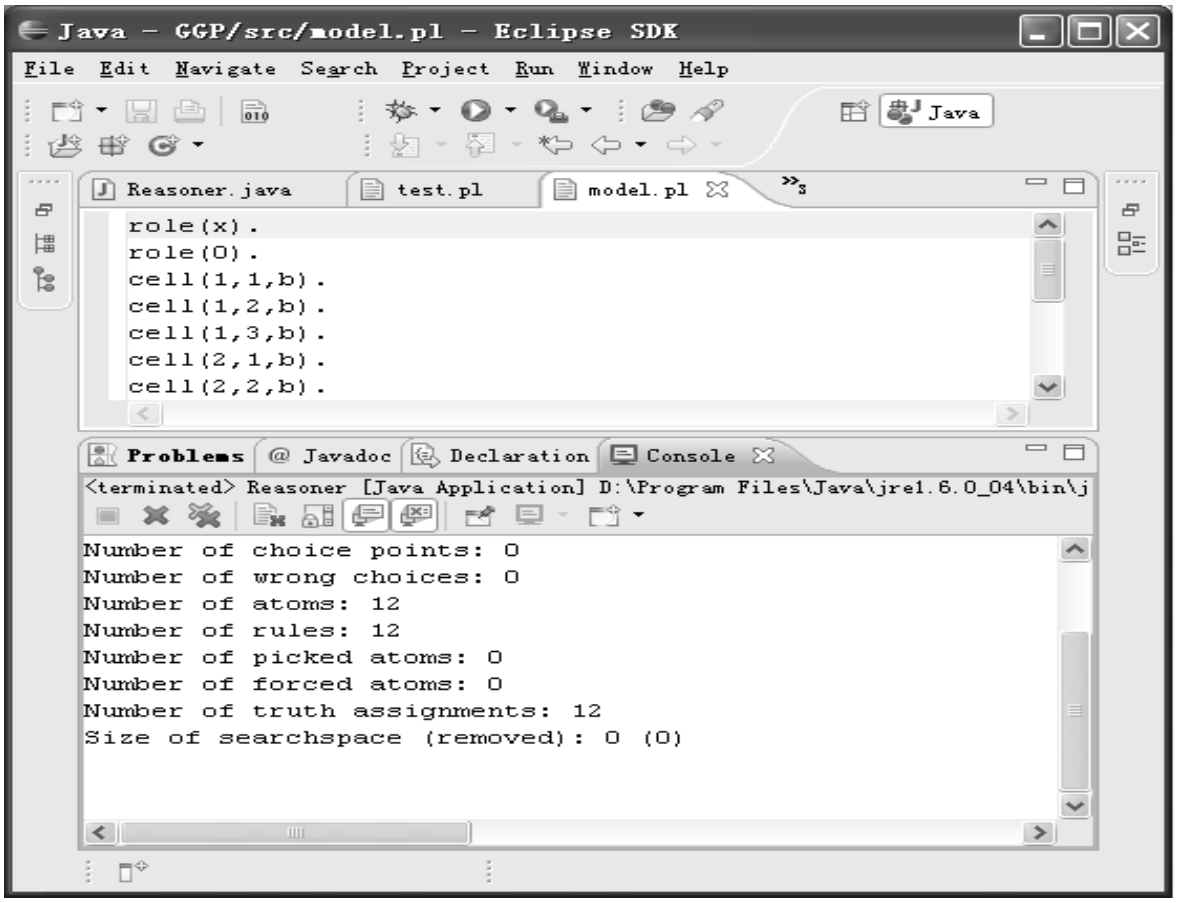
6.2 Compiler Module

I created a file which contains the game description for Tic-Tac-Toe in chapter 5. In order to make sure the component works as we expected, the file is passed into the module and output is printed (Figure 6.1).



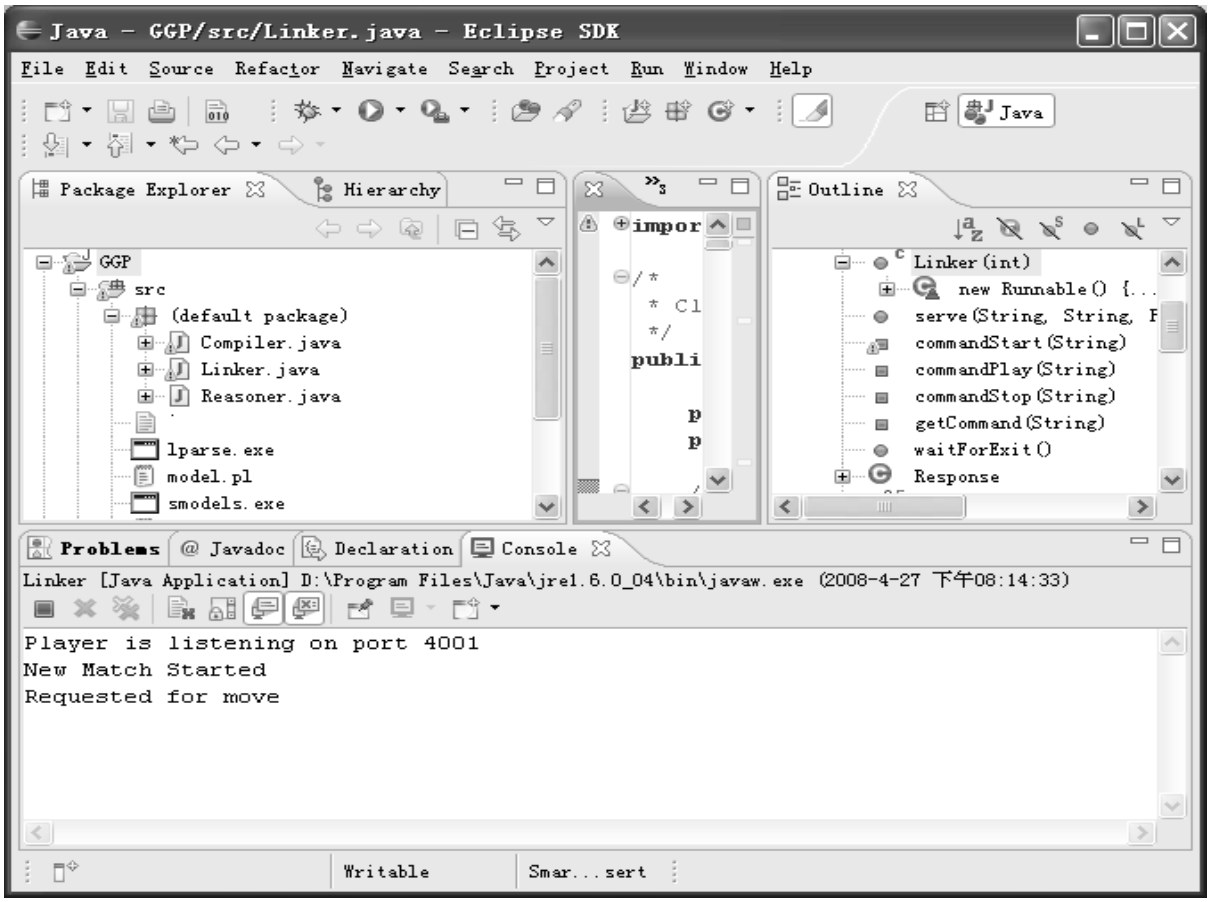
6.3 Reasoner Module

As mentioned in chapter 5, the heuristic search function has not been implemented yet. Therefore, this testing is focus on the first half of the Reasoner module: getModels. We would expect it to execute the command and print out the result (Figure 6.2).



6.4 System Testing

I did not expect the player to win the match as the system is not finished yet. However, we can still get some output from the console by carrying out the scenario described in chapter 5 (Figure 6.3).



The system would be able to receive the START command and indicates the start of new game. System holds on when receiving the request for a move.

Chapter 7

Conclusions

A General Game Playing system has been developed in this project. It answers the request from the game manager and generates moves for game playing. System uses ASP as reasoning module for generating player's move.

7.1 Project Review

7.1.1 Literature Review

Literature review has introduces the general game playing, the game description language that is used by the Stanford University Logic group for describing the games, the history and semantics of the answer set programming and heuristic search algorithms used by other players.

7.1.2 Requirement

In the requirement section, I have introduced the requirement analysis, which has been carried out by research of existing systems. Formal requirements are then concluded.

7.1.3 Design

I have introduced the structure of the system in design chapter, and how the system is broken into three different components. The functionality of each component is described in the section.

7.1.4 Implementation and Testing

In the Implementation and Testing chapter, I have briefed how components are built and the test cases used for the components.

7.2 Critique

The Compiler module does not care whether the given description is in capital or lower cases. It will compile the given description in the right format in ASP. However, lparse requires all atoms should be in lower cases, only variables can be presented as capital. Hence, if the given description is written in capital, the Reasoner module would still not be able to generate the move as program lparse refuses to accept it.

Major problem are found in the Reasoner module, even the game description is written in the format that lparse accepts. The program smodels only return 1 model, which consists all ground term specified in the file.

If we assume the problems mentioned above are working as expected, we still need to find a good heuristic search algorithm. Program smodels might generate the legal moves for the player, but there is no guarantee whether it is the best move.

7.3 Future Work

Future work includes more research on ASP, correcting problem mentioned in section 6.2 and finding a heuristic search algorithm.

Bibliography

Baral, C. (2003), *Knowledge Representation, Reasoning and Declarative Problem Solving*, Cambridge University Press.

Black, F. (1964), 'A deductive question answering system', *Harvard University Thesis* .

Ferraris, P. and Lifschitz, V. (2005), 'Weight constraints as nested expressions', *Theory and Practice of Logic Programming* **1538**(1-2), 47–74.

Foster, J. and Elcock, E. (1969), 'An incremental compiler for assertions: an introduction', *Machine Intelligence 4* pp. 423–429.

G. Kuhlmann, K. D. and Stone, P. (2006), 'Automatic heuristic contruction in a complete general game player', *Twenty-First National Conference on Artificial Intelligence* .

Gebser, M. and Schaub, T. (2006), 'Tableau calculi for answer set programming', *ICLP 2006* **4079**, 11–25.

Genesereth, M. and Love, N. (2005), 'General game playing: Overview of the aaai competition', *AI Magazine* **26**(2).

Genesereth, M. and Love, N. (2006), 'General game playing: Game description language specification', *Technical report, Stanford University* .

Green, C. (1969), 'Application of theorem proving to problem solving', *IJCAI* .

Hayes, P. (1973), 'Computation and deduction', *The 2nd MFCS Symposium* pp. 105–118.

Hewitt, C. (1971), 'Procedural embedding of knowledge in planner', *IJCAI* .

J. Schaeffer, J. C. and Treloar, N. (1992), 'A world championship caliber checkers program', *Artificial Intelligence* **53**(2-3), 273–289.

J.Scheaffer (1989), 'The history heuristic and alpha-beta search enhancements in practice', *IEEE Transactions on Pattern Analysis and Machine Intelligence* **11**(11), 1203–1212.

Kowalski, R. (1974), 'Predicate logic as a programming language', *IFIP Congress* pp. 569–574.

M. Campbell, A. H. and Hsu, F. (2002), 'Deep blue', *Artificial Intelligence* **134**(1-2), 57–3.

M. D'Agostino, D. Gabbay, R. H. and Posegga, J. (1999), *Handbook of Tableau Methods*, Kluwer Academic Publishers, Dordrecht.

Marek, V. and Truszczynski, M. (1999), 'Stable models and an alternative logic programming paradigm', *The Logic Programming Paradigm: a 25-Year Perspective* pp. 169–273.

McCarthy, J. (1958), 'Programs with common sense symposium on mechanization of thought processes', *National Physical Laboratory* .

Mitchell, D. (2005), 'A sat solver primer', *Bulletin of the European Association for Theoretical Computer Science 85* pp. 112–133.

Niemela, I. (1999), 'Logic programs with stable model semantics as a constraint programming paradigm', *Annals of Mathematics and Artificial Intelligence* **25**, 241–273.

P. Beame, H. K. and Sabharwal, A. (2004), 'Towards understanding and harnessing the potential of clause learning', *Journal of Artificial Intelligence Research 22* pp. 319–351.

P. Simons, I. N. and Soininen, T. (2002), 'Extending and implementing the stable model semantics', *Artificial Intelligence* **138**(1-2), 181–234.

Schiffel, S. and Thielscher, M. (2007), 'Automatic construction of a heuristic search function for general game playing', *Seventh IJCAI International Workshop on Nonmontonic Reasoning, Action and Change(NRAC07)* .

Shapiro, E. (1989), 'The family of concurrent logic programming languages', *ACM Computing Surveys* .

Slagle, J. (1965), 'Experiments with a deductive question-answering program', *CACM* .

Soininen, T. and Niemela, I. (1998), 'Formalizing configuration knowledge using rules with choices', *Helsinki University of Technology* .

Subrahmanian, V. and Zaniolo, C. (1995), 'Relating stable models and ai planning domains', *ICLP-95* pp. 233–247.

Y. Dimopoulos, B. N. and Kohler, J. (1997), 'Encoding planning problems in non-monotonic logic programs', *ECP-97, Springer Verlag* pp. 273–285.

Appendix A

Code

- Linker.java
  This is class is the interface between the player and the game manager. It will be started like a server, and response requests from the game manager.

- Compiler.java
  This class compiles the game description in GDL into ASP. Any updated moves in GDL will also be compiled to ASP via this module.

- Reasoner.java
  Reasoner.java is the module which will generate the move.

  This module is not finished, further work can be done this module in order to complete the entire system.

A.1 File: Linker.java

```
import java.io.*;
import java.text.DateFormat;
import java.util.*;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.net.*;

/*
 * Class Linker is the interface between the player and the game manager
 */
public class Linker {

        private Thread server_thread;
        private int myTcpPort;

        /**
         * Starts a HTTP server to given port.<p>
         * Throws an IOException if the socket is already in use
         */
        public Linker( int port) throws IOException
        {
                myTcpPort = port;

                final ServerSocket ss = new ServerSocket( myTcpPort );
                server_thread = new Thread( new Runnable()
                        {
                                public void run()
                                {
                                        try
                                        {
                                                while( true )
                                                        new HTTPSession( ss.accept());
                                        }
                                        catch ( IOException ioe )
                                        {}
                                }
                        });
                server_thread.setDaemon( true );
                server_thread.start();
                System.out.println("Player is listening on port "+port);
        }
```

```
public Response serve( String uri, String method, Properties header, Properties parms, String data )
{
        try{
                String response_string=null;
                if(data!=null){
                        System.out.println("Command: " + data);
                        String command=getCommand(data);
                        if(command==null){
                                throw(new IllegalArgumentException("Unknown message format"));
                        }else if(command.equals("START")){
                                response_string="READY";
                                commandStart(data);
                        }else if(command.equals("PLAY")){
                                response_string=commandPlay(data);
                        }else if(command.equals("STOP")){
                                response_string="DONE";
                                commandStop(data);
                        }else{
                                throw(new IllegalArgumentException("Unknown command:"+command));
                        }
                }else{
                        throw(new IllegalArgumentException("Message is empty!"));
                }
                System.out.println("Response:"+response_string);
                if(response_string!=null && response_string.equals("")) response_string=null;
                return new Response( HTTP_OK, "text/acl", response_string );
        }catch(IllegalArgumentException ex){
                System.err.println(ex);
                ex.printStackTrace();
                return new Response( HTTP_BADREQUEST, "text/acl", "NIL" );
        }
}



/**
 * this method is called when a new match begins
 */
private void commandStart(String msg){
        // msg format: (START MATCH_ID ROLE (GAME_DESCRIPTION) STARTCLOCK PLAYCLOCK)

        // We only interested in the GAME_DESCRIPTION, so only this information is stored.

//Description starts at the second '('
    int beginIndex = msg.indexOf('(',2);
```

```
    int endIndex = msg.lastIndexOf(')');
    String des = msg.substring(beginIndex,endIndex);

        Compiler.compile("des.txt");

}

/**
 * this method is called once for each move
 * @return the move of this player
 */
private String commandPlay(String msg){
        // msg format:(PLAY MATCH_ID (JOINT_MOVE))
        // JOINT_MOVE will be NIL for the first PLAY message and the list of the moves of all the players in
             the previous state
        // e.g. msg="(PLAY tictactoe1 NIL)" for the first PLAY message
        //   or msg="(PLAY tictactoe1 ((MARK 1 2) NOOP))" if white marked cell (1,2) and black did a "noop".

        // Get the updated moves
    int beginIndex = msg.indexOf('(',2);
    int endIndex = msg.lastIndexOf(')');
    String updatedMoves = msg.substring(beginIndex,endIndex);
        //System.out.println(updatedMoves);
    return Compiler.getMove(updatedMoves);

        //return "NIL";
}

/**
 * this method is called if the match is over
 */
private void commandStop(String msg){
        // Cleaning up the temporary created files
        boolean success = (new File("des.txt")).delete();
    if (!success) {
        System.out.println("Error: Could not delete file : des.txt");
    }
}

/*
 * Get the first element (Command) in the message
 */
private String getCommand(String msg){
        String cmd=null;
        try{
```

```
                Matcher m=Pattern.compile("\\(([^\\s]*)\\s").matcher(msg);
                if(m.lookingAt()){
                        cmd=m.group(1);
                }
                cmd.toUpperCase();
        }catch(Exception ex){
                System.err.println("Pattern to extract command did not match!");
                ex.printStackTrace();
        }
        return cmd;
}

public void waitForExit(){
        try {
                server_thread.join(); // wait for server thread to exit
        }catch(Exception ex){
                ex.printStackTrace();
        }
}

/**
 * HTTP response.
 * Return one of these from serve().
 */
public class Response
{
        public String status;
        public String mimeType;
        public InputStream data;
        public Properties header = new Properties();

        /**
         * Default constructor: response = HTTP_OK, data = mime = 'null'
         */
        public Response()
        {
                this.status = HTTP_OK;
        }

        /**
         * Basic constructor.
         */
        public Response( String status, String mimeType, InputStream data )
        {
                this.status = status;
```

```
                this.mimeType = mimeType;
                this.data = data;
        }

        /**
         * Convenience method that makes an InputStream out of
         * given text.
         */
        public Response( String status, String mimeType, String txt )
        {
                this.status = status;
                this.mimeType = mimeType;
                this.data = new ByteArrayInputStream( txt.getBytes());
        }

        /**
         * Adds given line to the header.
         */
        public void addHeader( String name, String value )
        {
                header.put( name, value );
        }
}

/**
 * Some HTTP response status codes
 */
public static final String
        HTTP_OK = "200 OK",
        HTTP_REDIRECT = "301 Moved Permanently",
        HTTP_FORBIDDEN = "403 Forbidden",
        HTTP_NOTFOUND = "404 Not Found",
        HTTP_BADREQUEST = "400 Bad Request",
        HTTP_INTERNALERROR = "500 Internal Server Error",
        HTTP_NOTIMPLEMENTED = "501 Not Implemented";

/**
 * Common mime types for dynamic content
 */
public static final String
        MIME_PLAINTEXT = "text/plain",
        MIME_HTML = "text/html",
        MIME_DEFAULT_BINARY = "application/octet-stream";

/**
```

```
 * Handles one session, i.e. parses the HTTP request
 * and returns the response.
 */
private class HTTPSession implements Runnable
{
        private Socket mySocket;

        public HTTPSession( Socket s )
        {
                mySocket = s;
                Thread t = new Thread( this );
                t.setDaemon( true );
                t.start();
        }

        public void run()
        {
                try
                {
                        String line;
                        InputStream is = mySocket.getInputStream();
                        if ( is == null) return;
                        BufferedReader in = new BufferedReader( new InputStreamReader( is ));

                        // Read the request line
                        line=in.readLine();
                        if(line!=null){
                                //System.out.println("got line:" + line);
                                StringTokenizer st = new StringTokenizer( line );
                                if ( !st.hasMoreTokens())
                                        sendError( HTTP_BADREQUEST, "BAD REQUEST: Syntax error. Usage: GET
                                            /example/file.html" );

                                String method = st.nextToken();

                                if ( !st.hasMoreTokens())
                                        sendError( HTTP_BADREQUEST, "BAD REQUEST: Missing URI. Usage: GET
                                            /example/file.html" );

                                String uri = decodePercent( st.nextToken());

                                // Decode parameters from the URI
                                Properties parms = new Properties();
                                int qmi = uri.indexOf( '?' );
                                if ( qmi >= 0 )
```

```
{
        decodeParms( uri.substring( qmi+1 ), parms );
        uri = decodePercent( uri.substring( 0, qmi ));
}

// If there's another token, it's protocol version,
// followed by HTTP headers. Ignore version but parse headers.
Properties header = new Properties();
if ( st.hasMoreTokens())
{
        line = in.readLine();
        while ( line!=null && line.trim().length() > 0 )
        {
                int p = line.indexOf( ':' );
                header.put( line.substring(0,p).trim(),
                    line.substring(p+1).trim());
                line = in.readLine();
        }
}

String data=null;
int length=0, length_so_far=0, len;
char[] cbuf=new char[1024];
// If the method is POST, there may be parameters
// in data section, too, read another line:
if ( method.equalsIgnoreCase( "POST" )){
        try{
                length=Integer.parseInt(header.getProperty("Content-length"));
        }catch(NumberFormatException ex){};

        while( length_so_far<length &&  (len=in.read(cbuf,0,1023))!=-1){
                line=new String(cbuf,0,len);
                if(data==null) data=line;
                else data += line;
                length_so_far+=len;
        }
}

// Ok, now do the serve()
Response r = serve( uri, method, header, parms, data );
if ( r == null )
        sendError( HTTP_INTERNALERROR, "SERVER_INTERNAL_ERROR: _Serve()_
            returned_a_null_response." );
else
        sendResponse( r.status, r.mimeType, r.header, r.data );
```

```
                        in.close();
                }
        }
        catch ( IOException ioe )
        {
                try
                {
                        sendError( HTTP_INTERNALERROR, "SERVER INTERNAL ERROR: IOException: " +
                            ioe.getMessage());
                }
                catch ( Throwable t ) {}
        }
        catch ( InterruptedException ie )
        {
                // Thrown by sendError, ignore and exit the thread.
        }
}

/**
 * Decodes the percent encoding scheme. <br/>
 * For example: "an+example%20string" -> "an example string"
 */
private String decodePercent( String str ) throws InterruptedException
{
        try
        {
                StringBuffer sb = new StringBuffer();
                for( int i=0; i<str.length(); i++ )
                {
                    char c = str.charAt( i );
                    switch ( c )
                        {
                        case '+':
                            sb.append( ' ' );
                            break;
                        case '%':
                        sb.append((char)Integer.parseInt( str.substring(i+1,i+3), 16 ));
                            i += 2;
                            break;
                        default:
                            sb.append( c );
                            break;
                    }
                }
```

```
                return new String( sb.toString().getBytes());
        }
        catch( Exception e )
        {
                sendError( HTTP_BADREQUEST, "BAD_REQUEST: Bad percent-encoding." );
                return null;
        }
}

/**
 * Decodes parameters in percent-encoded URI-format
 * ( e.g. "name=Jack%20Daniels&pass=Single%20Malt" ) and
 * adds them to given Properties.
 */
private void decodeParms( String parms, Properties p )
        throws InterruptedException
{
        if ( parms == null )
                return;

        StringTokenizer st = new StringTokenizer( parms, "&" );
        while ( st.hasMoreTokens())
        {
                String e = st.nextToken();
                int sep = e.indexOf( '=' );
                if ( sep >= 0 )
                        p.put( decodePercent( e.substring( 0, sep )).trim(),
                                        decodePercent( e.substring( sep+1 )));
        }
}

/**
 * Returns an error message as a HTTP response and
 * throws InterruptedException to stop furhter request processing.
 */
private void sendError( String status, String msg ) throws InterruptedException
{
        sendResponse( status, MIME_PLAINTEXT, null, new ByteArrayInputStream( msg.getBytes()));
        throw new InterruptedException();
}

/**
 * Sends given response to the socket.
 */
private void sendResponse( String status, String mime, Properties header, InputStream data )
```

```
{
        try
        {
                if ( status == null )
                        throw new Error( "sendResponse(): Status can't be null." );

                OutputStream out = mySocket.getOutputStream();
                PrintWriter pw = new PrintWriter( out );
                pw.print("HTTP/1.0 " + status + " \r\n");

                if ( mime != null )
                        pw.print("Content-Type: " + mime + "\r\n");

                if ( header != null )
                {
                        Enumeration e = header.keys();
                        while ( e.hasMoreElements())
                        {
                                String key = (String)e.nextElement();
                                String value = header.getProperty( key );
                                pw.print( key + ": " + value + "\r\n");
                        }
                }

                pw.print("\r\n");
                pw.flush();

                if ( data != null )
                {
                        byte[] buff = new byte[2048];
                        int read = 2048;
                        while ( read == 2048 )
                        {
                                read = data.read( buff, 0, 2048 );
                                if(read>0) out.write( buff, 0, read );
                        }
                }
                out.flush();
                out.close();
                if ( data != null )
                        data.close();
        }
        catch( IOException ioe )
        {
                // Couldn't write? No can do.
```

```
                        try { mySocket.close(); } catch( Throwable t ) {}
                }
        }
};


/**
 * starts the game player and waits for messages from the game master <br>
 * Command line options: [port]
 */
public static void main(String[] args){
        try{
                int port=4001;
                if(args.length>=1){
                        port=Integer.parseInt(args[0]);
                }
                Linker gp=new Linker(port);
                gp.waitForExit();
        }catch(Exception ex){
                ex.printStackTrace();
                System.exit(-1);
        }
}

}
```

A.2 File: Compiler.java

```
import java.io.*;
import java.util.ArrayList;

/*
 * Class Compiler is the intermediate stage which translates GDL into ASP.
 */
public class Compiler {

        /*
         * Interface allows Linker to compiler the given file
         */
        static public void compile(String filename)
        {
                try{
                FileInputStream fstream = new FileInputStream(filename);

                    DataInputStream in = new DataInputStream(fstream);
                    BufferedReader br = new BufferedReader(new InputStreamReader(in));

                    String strLine;
                    String content = "";

                    while ((strLine = br.readLine()) != null){
                        content = content + strLine + "\n";
                        //System.out.println (strLine);
                    }
                    //System.out.println(content);
                    in.close();

                    // After getting the content from the file, we need to parse it first
                    ArrayList list = desParser(content);
                    String target = desGenerator(list);
                    //System.out.println(target);

                    //Writing the targeted code into a file.
                    FileWriter outFile = new FileWriter("model.pl");
                    PrintWriter out = new PrintWriter(outFile);
                    out.println(target);
                    out.close();

        }catch (Exception e){
                //Catch exception if any
```

```
        System.err.println("Error: " + e.getMessage());
        }

}

/*
 * Interface allows Linker to compiler the given file
 */
static public String getMove(String updateMoves)
{
        String move="";

        try{
            // After getting the content from the file, we need to parse it first
            String target = ruleGenerator(ruleParser(updateMoves));
            //System.out.println(target);

            //Appending the targeted code into a file.
            FileWriter outFile = new FileWriter("model.pl",true);
            PrintWriter out = new PrintWriter(outFile);
            out.println(target);
            out.close();

            // Then we call the code_generator to generate the ASP.

            // Generate Moves
            move = Reasoner.getMove();

}catch (Exception e){
        //Catch exception if any
        System.err.println("Error: " + e.getMessage());
        }

return move;

}

/*
 * This function parser the given file (content must be in GDL)
 */
static private ArrayList desParser(String des)
{
        ArrayList list = new ArrayList();
        //Store the number of brackets
        int open_bracket=0, close_bracket =0;
```

```
        //Store the substring index
        int beginIndex=0;
        int index=0;

        while (index < des.length()){
                char ch = des.charAt(index);
                if (ch=='('){
                        if (open_bracket==0)
                                beginIndex = index;
                        open_bracket++;
                }
                if (ch==')')
                        close_bracket++;
                if ((open_bracket==close_bracket) && (open_bracket!=0)){
                        String s = des.substring(beginIndex,index+1);
                        beginIndex=index+1;
                        if (!s.equals(" ")){
                                list.add(s);
                                //System.out.println(s);
                        }
                        open_bracket=0;
                        close_bracket=0;
                }
                index++;
        }

        return list;
}

/*
 * This function parser the given GDL rules.
 */
static private ArrayList ruleParser(String statement)
{
        ArrayList list = new ArrayList();
        //Store the number of brackets
        int open_bracket=0, close_bracket =0;
        //Store the substring index
        int beginIndex=1;
        int index=1;
        char ch;
        String s;

        // Get the keyword and store it in the first element of the list
        ch=statement.charAt(index);
```

```
        while(ch!=' '){
                index++;
                ch=statement.charAt(index);
        }
        s=statement.substring(beginIndex,index);
        list.add(s);
        //System.out.println(s);

        while (index < statement.length()){
                ch = statement.charAt(index);
                if (ch=='('){
                        if (open_bracket==0)
                                beginIndex = index;
                        open_bracket++;
                }
                if (ch==')')
                        close_bracket++;
                if ((open_bracket==close_bracket) && (open_bracket!=0)){
                        s = statement.substring(beginIndex,index+1);
                        beginIndex=index+1;
                        if (!s.equals(" ")){
                                list.add(s);
                                //System.out.println(s);
                        }
                        open_bracket=0;
                        close_bracket=0;
                }
                index++;
        }

        return list;
}

/*
 * This function parser the given GDL terms.
 */
static private ArrayList termsParser(String term)
{
        ArrayList list = new ArrayList();
        int beginIndex = 1;
        int index =1;
        String s = "";

        while(index<term.length()){
                char ch = term.charAt(index);
```

```
                //System.out.println(ch);

                if(ch == ')' || ch == ' '){
                        s = term.substring(beginIndex,index);
                        //System.out.println(s);
                        list.add(s);
                        beginIndex = index+1;
                }
                index++;
        }

        return list;
}



/*
 * This function generates the given description.
 */
static private String desGenerator(ArrayList statementList)
{
        String statement = "";
        String arg;
        String keyword;
        ArrayList ruleList;

        for (int i=0; i<statementList.size();i++){
                arg = (String)statementList.get(i);
                keyword = getKeyword(arg);

                if (keyword.toUpperCase().equals("ROLE")){
                        statement = statement + termsGenerator(termsParser(arg))+ ".\n";
                        //System.out.println(statement);
                };

                if (keyword.toUpperCase().equals("INIT")){
                        statement = statement + ruleGenerator(ruleParser(arg));
                };

                if (keyword.toUpperCase().equals("<=")){
                        statement = statement + ruleGenerator(ruleParser(arg));
                };

        }
```

```
        return statement;
}

/*
 * This function generates the given statements.
 */
static private String ruleGenerator(ArrayList ruleList)
{
        String rule = "";
        String arg;
        String keyword;

        keyword = (String)ruleList.get(0);

        if (keyword.toUpperCase().equals("INIT")){
                arg = (String)ruleList.get(1);
                rule = rule+termsGenerator(termsParser(arg))+ ".\n";
        };

        if (keyword.toUpperCase().equals("<=")){

                // Print out the head
                arg = (String)ruleList.get(1);
                if (isTerm(arg)){
                        rule = rule+termsGenerator(termsParser(arg))+ ":-";
                }else{
                        rule = rule+ruleGenerator(ruleParser(arg))+ ":-";
                }

                // print out the body without the last item
                for (int i=2;i<ruleList.size()-1;i++)
                {
                        arg = (String)ruleList.get(i);
                        if (isTerm(arg)){
                                rule = rule+termsGenerator(termsParser(arg))+ ",";
                        }else{
                                rule = rule+ruleGenerator(ruleParser(arg))+ ",";
                        }
                }

                // Print out the last item in the body
                arg = (String)ruleList.get(ruleList.size()-1);
                if (isTerm(arg)){
                        rule = rule+termsGenerator(termsParser(arg))+ ".\n";
                }else{
```

```
                        rule = rule+ruleGenerator(ruleParser(arg))+ ".\n";
                }
        } ;

        if (keyword.toUpperCase().equals("TRUE")){
                arg = (String)ruleList.get(1);
                rule = rule+termsGenerator(termsParser(arg));
        };

        if (keyword.toUpperCase().equals("TERMINAL")){
                rule = rule + "TERMINAL:-";

                // print out the body without the last item
                for (int i=2;i<ruleList.size()-1;i++)
                {
                        arg = (String)ruleList.get(i);
                        if (isTerm(arg)){
                                rule = rule+termsGenerator(termsParser(arg))+ ",";
                        }else{
                                rule = rule+ruleGenerator(ruleParser(arg))+ ",";
                        }
                }

                // Print out the last item in the body
                arg = (String)ruleList.get(ruleList.size()-1);
                if (isTerm(arg)){
                        rule = rule+termsGenerator(termsParser(arg))+ ".\n";
                }else{
                        rule = rule+ruleGenerator(ruleParser(arg))+ ".\n";
                }
        };

        return rule;
}

/*
 * This function parser the given GDL terms.
 */
static private String termsGenerator(ArrayList atomList)
{
        String term = "";

        //System.out.print((String)atomList.get(0)+"(");
        term = term + (String)atomList.get(0)+"(";
        for (int i=1;i<atomList.size()-1;i++){
```

```
                //System.out.print((String)atomList.get(i)+",");
                term = term + (String)atomList.get(i)+",";
        }
        //System.out.print((String)atomList.get(atomList.size()-1)+").");
        term = term + (String)atomList.get(atomList.size()-1)+")";

        return term;
}

static private String getKeyword(String rule)
{
        int beginIndex;
        int index;
        String keyword;
        char ch;

        if (rule.startsWith("(")){
                beginIndex = 1;
                index = 1;
        }else{
                beginIndex = 0;
                index = 0;
        }

        ch=rule.charAt(index);
        while(ch!=' '){
                index++;
                ch=rule.charAt(index);
        }
        keyword=rule.substring(beginIndex,index);

        return keyword;
}

static private boolean isTerm(String rule)
{
        int open_bracket = 0;
        char ch;

        for (int i=0;i<rule.length();i++){
                ch = rule.charAt(i);
                if (ch == '(')
                        open_bracket++;
        }
```

```
                if (open_bracket<2){
                        return true;
                }else{
                        return false;
                }
        }


    public static void main(String args[]) {
        //ArrayList list = Compiler.termsParser("(cell m n b)");
        //String s = Compiler.termsGenerator(list);

        //ArrayList list = Compiler.ruleParser("(<= (GOAL ?PLAYER 50) (NOT (LINE X)) (NOT (LINE O)) (NOT OPEN))");
        //String s = Compiler.desGenerator(list);

        //System.out.println(Compiler.getKeyword("(goal (GOAL ?PLAYER 50) (NOT (LINE X)) (NOT (LINE O)) (NOT
            OPEN))"));

        Compiler.compile("des.txt");
    }
}
```

A.3 File: Reasoner.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/*
 * Class Reasoner tries to generate the best move.
 */
public class Reasoner {

        /*
         * This function is the interface for other components
         */
        static public String getMove()
        {
                String models = getModels();
                return generateMove(models);
        }

        /*
         * This function calls the external programs lparser and smodels
         * and return the models returned by the programs
         */
        static private String getModels()
        {
        String s = null;
        String answers = "";
        try {
            // The first line should be uncomment if running in Unix
            //Process p = Runtime.getRuntime().exec("cmd /c src/lparse src/test.pl|src/smodels");
            Process p = Runtime.getRuntime().exec("cmd /c src\\lparse src\\test.pl|src\\smodels");

            BufferedReader stdInput = new BufferedReader(new
                 InputStreamReader(p.getInputStream()));

            // read the output from the command
            while ((s = stdInput.readLine()) != null) {
                answers = answers + s;
                System.out.println(s);
            }

        }
        catch (IOException e) {
```

```
            System.out.println("Error message" + e.getMessage());
        }
        return answers;
    }

        /*
         * This function is used to generate the best move
         */
        static private String generateMove(String models)
        {
                return "";

                // TODO:
                // Find a heuristic function algorithm, applied it to the
                // answer given.
        }

    public static void main(String args[]) {
        System.out.println(Reasoner.getMove());

    }
}
```