České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Ondřej M u s i l

Studijní program: Otevřená informatika (bakalářský)

Obor: Informatika a počítačové vědy

Název tématu: Využitelnost hráčů obecných her jako AI knihoven v praktických aplikacích

Pokyny pro vypracování:

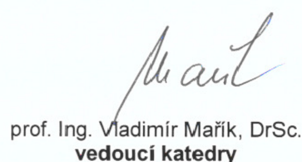
1. Student nastuduje hlavní principy, pravidla a požadavky soutěže v automatickém hraní obecních her (General Game Playing Competition), organizované při konferenci AAAI. Mimo jiné se seznámí s jazyky používanými pro specifikaci her (GDL, GDL-II).
2. Identifikuje vhodné aplikační problémy, na které by GGP mohlo být použitelné jako řešící knihovna a pokusí se je co nejpřesněji definovat (například pronásledování inteligentních cílů v městském prostředí).
3. Vybere si minimálně jednu z navržených domén a vytvoří korektní hru v jazyku GDL (případně GDL-II), která bude co možná nejlépe zachycovat vybraný problém.
4. Vytvoří jednoduchou doménově specifickou implementaci algoritmu řešící vybraný problém.
5. Experimentálně porovná kvalitu řešení her existujícími GGP hráči a jeho doménově specifickým algoritmem.
6. Bude diskutovat možnosti, výhody a nevýhody použití obecných hráčů her jako AI knihovny pro konkrétní problémy.

Seznam odborné literatury:

[1] Genesereth, M., Love, N., & Pell, B. (2005). General game playing: Overview of the AAAI competition. AI Magazine,26(2), 62. http://www.aaai.org/ojs/index.php/aimagazine/article/viewArticle/1813
[2] Thielscher, M. (2010). A General Game Description Language for Incomplete Information Games. AAAI 2010. http://cgi.cse.unsw.edu.au/~mit/Papers/AAAI10a.pdf
[3] Finnsson, H., & Björnsson, Y. (2008). Simulation-based approach to general game playing. Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI (pp. 259-264). http://www.aaai.org/Papers/AAAI/2008/AAAI08-041.pdf

Vedoucí bakalářské práce: Mgr. Viliam Lisý, MSc.

Platnost zadání: do konce zimního semestru 2012/2013



prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 9. 1. 2012

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Ondřej M u s i l

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: Usability of General Game Players as AI Libraries in Practical Applications

Guidelines:

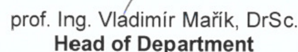
1. The student will study the mail principles, rules and requirements of the General Game Playing (GGP) competition, organized at the AAAI conferences. Primarily, he will study the languages used for specification of the games in the competition (GDL, GDL-II).
2. He will identify suitable applied problems that could be (partially) solved using a general game player as an AI library and he will precisely define the problem (e.g., pursuit of an intelligent target in an urban environment).
3. He will create a simple domain specific implementation for solving the selected problem.
4. He will experimentally evaluate how well the existing GGP players solve the problem in comparison to his domain specific algorithm.
5. He will discuss the possibility, advantages and disadvantages of using general game players as AI libraries for specific problems.

Bibliography/Sources:

[1] Genesereth, M., Love, N., & Pell, B. (2005). General game playing: Overview of the AAAI competition. AI Magazine,26(2), 62. http://www.aaai.org/ojs/index.php/aimagazine/article/viewArticle/1813
[2] Thielscher, M. (2010). A General Game Description Language for Incomplete Information Games. AAAI 2010. http://cgi.cse.unsw.edu.au/~mit/Papers/AAAI10a.pdf
[3] Finnsson, H., & Björnsson, Y. (2008). Simulation-based approach to general game playing. Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI (pp. 259-264). http://www.aaai.org/Papers/AAAI/2008/AAAI08-041.pdf

Bachelor Project Supervisor: Mgr. Viliam Lisý, MSc.

Valid until: the end of the winter semester of academic year 2012/2013

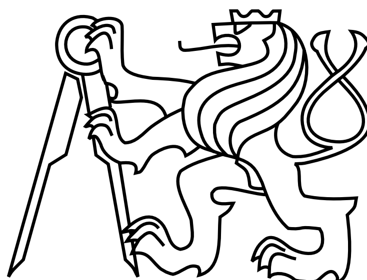


prof. Ing. Vladimír Mařík, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 9, 2012

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics



Bachelor's Thesis

Usability of General Game Players as AI libraries in practical applications

Ondřej Musil

Supervisor: Mgr. Viliam Lisý, MSc.

Study Programme: Open informatics

Field of Study: Computer and Information Science

May 25, 2012

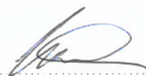
Aknowledgements

I would like to thank my supervisor, Mgr. Viliam Lisý, Msc., for the patient assistance and guidance during the entire time, my family, friends and all the people that supported me, while I was writing this thesis. Without you, most of this would never be possible.

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 25. 5. 2012



Podpis autora práce

Abstract

The goal of General Game Playing (GGP) is to create intelligent agents that are able to learn and solve different problems at an expert level without any human intervention. In this thesis, I will discuss the basis for the use of such agents, and knowledge required for writing problems in description language (GDL). I will show short examples of games in GDL and GDL-II. I will also summarize result of competition organized under AAAI's summer conference and introduce some existing players. We will try how general player handles a practical problem and how it stand up against domain specific player, that can be implemented in the time comparable to the time needed for integration of the general player. We will discuss advantages and disadvantages of application of these agents in practical application such at Pursuit-evasion game.

Abstrakt

Cílem hraní obecných her je vytvořit inteligentní agenty, kteří budou schopni se naučit a řešit různé problémy na úrovni expertů, bez zásahu člověka. V této práci rozeberu základy pro použití takových agentů a pořebné znalosti pro popis problémů v popisném jazyku her (GDL). Ukáži krátké ukázky her v jazyce GDL a GDL-II. Také shrnu výsledky soutěže pořádané během letní konference AAAI a uvedu některé z existujících hráčů. Vyzkouším jak si obecný hráč poradí s konkrétním problémem a jak obstojí proti doménově specifickému agentovi, který je naimplementován za dobu srovnatelnou s dobou potřebnou pro integraci obecného hráče. Také prodiskutujeme výhody a nevýhody použití těchto hráčů v praktickém použití, například u Pursuit-evasion her.

Contents

1 Introduction **1**
1.1 Thesis outline . . . 2

2 General Game Playing **3**
2.1 GGP Competitions and results . . . 3
2.2 Game Manager . . . 4
2.2.1 Handling illegal moves and timeouts . . . 6

3 Game Description Language **7**
3.1 GDL . . . 8
3.1.1 Restrictions . . . 8
3.1.2 Sample implementation . . . 9
3.2 GDL-II . . . 11
3.2.1 Sample implementation . . . 12
3.2.2 Game Checker . . . 13

4 Algorithms **14**
4.1 A* and IDA* . . . 14
4.2 Depth-first search . . . 14
4.3 MiniMax . . . 14
4.4 Monte Carlo method . . . 15
4.5 Upper Confidence Bounds applied to Trees . . . 16

5 Successful players **18**
5.1 Fluxplayer . . . 18
5.2 CADIAplayer . . . 19
5.3 Ary . . . 19

6 Problem solving **20**
6.1 Game descriptions . . . 21
6.2 Domain player implementation . . . 27
6.2.1 Domain players . . . 27
6.3 Experiments . . . 28
6.4 Results . . . 32
6.5 Summary . . . 34

7 Discussion **35**

8 Conclusions **38**
8.1 Future work 39

A List of used acronyms **42**

B GDL descriptions **43**
B.0.1 Attacker and guard (GDL) 43
B.0.2 Attacker and guard (GDL-II) 47
B.0.3 Basic PE game 51

C Installation and user guide **54**
C.1 Installation 54
C.2 Usage 54

D Content of included CD **56**

List of Figures

2.1 (Genesereth2005): Communication diagram 4

4.1 Minimax Game tree 16
4.2 (Chaslot, Bakkes, Szita & Spronck, 2008): Monte Carlo Tree search control loop 17

6.1 World discretization using graph representation. 21
6.2 Grid (11x11) representation with shortest path to escape point highlighted. (E - evader, P - pursuer, X - escape point) 29
6.3 Rubber and police, town representation 31

C.1 Game Controller's window 55

List of Tables

2.1 Annual results of the GGP competitions 3

3.1 (Thielscher, 2010): Main GDL keywords and their functionality. 8
3.2 GDL-II extension 12

6.1 Domain player classes 27
6.2 The experiments overview 28
6.3 Grid 11x11, Game Checker evaluation (rewards are listed in order: evader1, pursuer1, pursuer2) 29
6.4 Grid 31x31, Game Checker evaluation 30
6.5 Rigged map, Game Checker evaluation (rewards are listed in order: evader1, pursuer1, pursuer2) 31
6.6 Experiments results, containing average steps, steps standard deviation (in brackets) and average goal rewards. Roles ids described in Table 6.7. 32
6.7 Player's identifiers description 32

7.1 Summary of advantages and disadvantages of the general players 35

C.1 Player's main classes 54

Chapter 1

Introduction

The goal of General Game Playing is to create intelligent systems that can automatically learn how to play a wide variety of games and problems, given only the description of the game rules. Such system requires a form of general intelligence that enables it to autonomously adapt to new and possibly radically different environments. General game-playing systems are a quintessential example of a new generation of software that end users can customize for their own specific tasks.

In order to perform well, general game players must incorporate various Artificial Intelligence technologies, such as knowledge representation, reasoning, learning, and rational decision making; and these capabilities have to work together in integrated manner.

At first the games with perfect information were studied. And the players in this area were quite successful. Two years ago in 2010 Michael Thielscher introduced an extension of standard Game Description Language (GDL) called Game Description Language for Incomplete Information Games. Which opens new possibilities for solving problems and future improvements of player.

General game playing is a topic with an inherent interests and work in this area has also practical value. The underlying technology can be used in a variety of other application areas, such as business process management, electronic commerce, and military operations. The usage of such players can be very effective for us. When solving some problems, we usually use algorithms designed for those problems, or we have to develop our own heuristics. When the rules are changed the new heuristic needs to be developed. But when the general player is used, the only thing needed is to modify game description and player will adapt to changes alone.

During 7 years since the introduction of General Game playing a number of research groups have been established world-wide, especially the German universities in Berlin, Bremen and Potsdam. Most of these groups developed their own players, but also there is an increasing number of researchers who are interested in specific aspects of general game playing, which does not require them to build a full-fledged, competitive game-playing system.

1.1 Thesis outline

In this section I present the outline of my thesis which should present the basic idea what to expect from this thesis.

In Chapter 2 I will introduce General Game Playing competition and its results and Game communication protocol and Game manager, which controls the matches.

In the following Chapter 3 I will introduce syntax and some restrictions for Game Description Language. Then I will show how can be games and problems written in Game Description Language and its extension Game Description Languages for Imperfect Information.

In Chapter 4 I will summarize algorithms that are commonly used in general playing. Such as Monte Carlo methods and Minimax.

In the following Chapter 5 I will introduce some players and summarize information about them.

A Chapter 6 will be devoted to a specific problem, namely Pursuit-Evasion game, I also present results of particular scenarios and we will discuss them.

In Chapter 7 I will discuss advantages and disadvantages of general player. And we will discuss other suitable problems.

In Chapter 8 I will evaluate this work, revisit goals and also I will discuss future work on this subject.

Chapter 2

General Game Playing

The idea itself, that is to build a system that can learn to play plenty of games, has been around for over 40 years. The French AI pioneer Jacques Pitrat wrote the first ever program that, in principle, could learn to play arbitrary chess-like board games by being given their rules [11]. Later general game-playing programs include [10], but it required AAAI competition to spark broad interest in this problem as an AI Grand Challenge.

2.1 GGP Competitions and results

Main competition in General Game Playing is held under AAAI summer conference. This competition has been organized annually since 2005 by Stanford people. This competition was primarily established to promote work in this research area. The team (or the individual) developer of the winning player obtain financial prize worth of $10000.

In Table 2.1 we can find annual results of AAAI competition. As we can see from the table the names of players are often repeating. Thirteen players was registered to the competition in 2011. And only one new player got to top four. We can see that the most successful player is Cadia with two wins,two second places and one third. Cadia is prosecuted by Ary, which

	2005	**2006**	**2007**
1.	Cluneplayer	Fluxplayer 5.1	CADIAplayer 5.2
2.	Goblin	Cluneplayer	Fluxplayer 5.1
3.		UT-AUSTIN-LARG	Ary 5.3
4.		Ogre	ClunePlayer

	2008	**2009**	**2010**	**2011**
1.	CADIAplayer 5.2	Ary 5.3	Ary 5.3	Turboturtle
2.	ClunePlayer	CADIAplayer 5.2	Maligne	CADIAplayer 5.2
3.	Ary 5.3	TurboTurtle	CADIAplayer 5.2	Ary 5.3
4.	Fluxplayer 5.1	Fluxplayer 5.1	Fluxplayer 5.1	Nexplayer

Table 2.1: Annual results of the GGP competitions

also won twice and three times Ary finished third. In Successful players I will introduce more details of three most successful players.

2.2 Game Manager

A mediator is necessary for playing general games, it distributes game descriptions to players, receives and validates theirs moves and distributes moves to all players, so they can evaluate changes in the game states. Also the mediator recognizes the end of the game and its winner. In GGP this mediator is called Game Manager (GM). For communication between players and GM basic HTTP protocol is used. The communication diagram is shown in Figure 2.1 This ensures the developers choice of language and how they will implement their player. The entire course of the match is controlled by three commands: *start*, *play* and *stop*. In following sections I will introduce each of these actions and response to them.

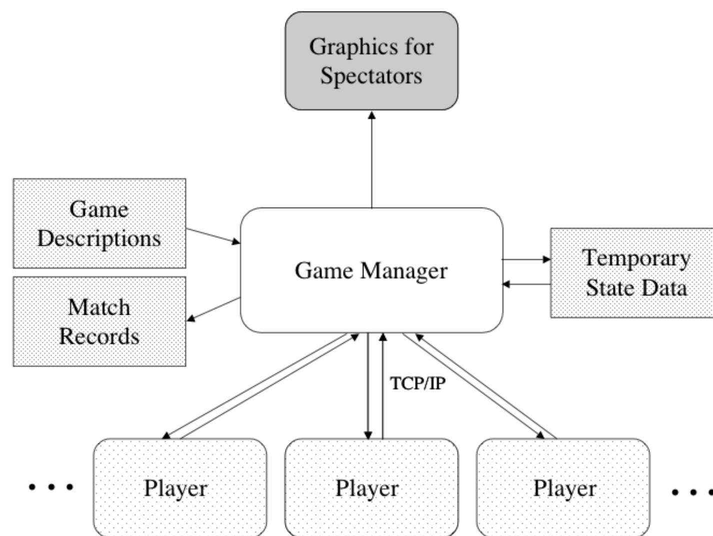


Figure 2.1: (Genesereth2005): Communication diagram

START Command

The Start command is used to initialize new match. The command has five parameters.

```
(START <match_id> <player_role> <game_description> <start_clock> <play_clock>)
```

- **match_id** is the unique identifier of the match. This enables the players to play more than one match at a time. Every subsequent command associated with the match will contain this id.
- **layer_role** is a role name associated to player to which the command is sent.

- **game_description** is a set of Kif sentences, enclosed by an outer set of parentheses.
- **game_clock** is an integer number representing time in seconds before the match begins.
- **play_clock** is an integer number representing time in seconds for every turn.

Reply

Player should reply with **READY** before the *startclock* elapses. However, Game master sends *Play message* to all players when *startclock* expires, regardless of whether they replied. If all players replied with *READY* before *startclock* expires, the match starts immediately.

PLAY Command

The Play command is sent every time when the next move is requested from the player. The *Play command* has 2 parameters.

```
(PLAY <match_id> (<A1> <A2> <A3> ...))
```

- **match_id** tells for what match this command is intended.
- **A1 A2 A3** is a sequence of actions made since the last *Play command* call by each player. The order of <A?> is identical with the order in which the roles are defined.

For example when we have the roles defined as following:

```
(role player1)
(role player2)
...
```

And the player receives the following *play* message from the GM:

```
(PLAY match_test ((MOVE UP) (MOVE DOWN)))
```

Than player1 moves up and player2 moves down.

Reply

The Play command expects a reply containing player's move. For example, if player wants to move upwards, then he sends a reply:

```
(MOVE UP)
```

STOP Command

The Stop command is used to notify player about end of the match. The Stop command is similar to the Play command. The command also sends match_id and player's moves.

```
(STOP <match_id> (<A1> <A2> <A3> ...))
```

The meaning of the variables is the same as in the Start command.

Reply

Players reply for STOP command should be DONE.

2.2.1 Handling illegal moves and timeouts

If the player submits an illegal move (or fails his sending at all) the GM will choose a random legal move for that player. Further play command is send as usual, so the player has chance to submit his own legal move. There is no penalty for submitting illegal moves. One similar situation might occur. When *playclock* is expired and the GM did not receive answer from the player a random move is also made by the GM for that player.

Chapter 3

Game Description Language

The most accepted language for describing games was developed by Michael Genesereth and team around [3] in 2005 and is called Game Description Language (GDL) [8]. GDL is a specification of Knowledge Interchange Format (KIF) [2], a first-order logic based language for describing and communication knowledge. It is a variant of Datalog that allows constants, negations and recursion. A GDL game description uses terms, relations and implications. GDL as well as KIF is usually given in prefix notation.

A term is either a variable or an atom, which is a constant. Variables in GDL start with question mark *?*, followed by string. The atoms are simply strings.

```
atom
?variable
```

The relations specify initial state, as well as every other state. A relation consist of a *relation name* and *n parameters* following the name.

```
(relation_name p1 p2 p3 ...)
```

Each parameter can be a term or other relation. If it is relation, than it must be closed in brackets.

```
(relation_name p1 (relation_name2 p2a p2b) p3)
```

The implications define rules for recognizing and evaluating terminal states, generating and playing legal moves. Implication have a head, which is a conclusion of it, and a body containing the conditions. If all conditions are *true*, than a conclusion becomes true as well. Also the implication symbol $\Leftarrow$ is written in prefix notation.

```
(<= (head p1 p2 ...)
        (body1 p1 p2 ...)
        (body2 p1 p2 ...)
        ...
        (bodyn p1 p2))
```

In GDL is difference between facts which are known globally and facts that are true in current state. The globally known fact can't be changed during game, so global fact should be used only for static elements of game. Everything what is changed during game, such as players position or grasp supplies.

Keyword	Function
role(R)	R is a player
init(F)	F holds in initial state
true(F)	F holds in current state
legal(R, M)	player R can do move M in current position
does(R, M)	player R does move M
next(F)	F holds in next state
terminal	current state is terminals
goal(R, N)	player R gets N points in current position
distinct(P1, P2)	requires that predicate P1 and P2 are syntactically different.

Table 3.1: (Thielscher, 2010): Main GDL keywords and their functionality.

3.1 GDL

Game Description Language is a language used to describe discrete games with perfect information. The GDL distinguishes this relations: *role*, *init*, *true*, *does*, *next*, *legal*, *goal*, *terminal* and *distinct* significance of these relations can be found in table 3.1.

3.1.1 Restrictions

The games described by GDL can't be infinite. We can evade this limitation by adding step counter to game description.

The GDL is limited to first order logic, in the contrast to the Prolog programming language. The variables can't be bound to predicates, so it is impossible to define arithmetic in GDL. So this makes it necessary to define arithmetic in the game description, when it is needed, with an explicit enumeration of the possible values.

Every game described by GDL should met several restrictions [8]:

Definition 1 (Termination): *A game described by GDL terminates if all infinite sequences of legal moves from the initial state of the game reach a terminal state after a finite number of steps.*

Definition 2 (Playability): *A game description in GDL is playable if and only if every role has at least one legal move in every non-terminal state reachable from the initial state.*

Definition 3 (Monotonicity): *A game description in GDL is monotonic if and only if every role has exactly one goal value in every state reachable from the initial state, and goal values never decrease.*

Definition 4 (Winnability): *A game description in GDL is strongly winnable if and only if, for some role, there is a sequence of individual moves of that role that leads to a terminal state of the game where that role's goal value is maximal. A game description in GDL is weakly winnable if and only if, for every role, there is a sequence of joint moves of all roles that leads to a terminal state where that role's goal value is maximal.*

Definition 5 (Well-formed Games): *A game description in GDL is well-formed if it terminates, is monotonic, and is both playable and weakly winnable.*

In addition to these rules, there are also restrictions to the use of keywords:

- **role**: appears only in head of facts.
- **init**: appears only in head of clauses and does not depend on: *true*, *legal*, *does*, *next*, *terminal*, *goal*.
- **true**: only appears in body of clauses.
- **does**: appears only in body of clauses, does not deppend on: *legal*, *term*, *goal*.
- **next**: appears only in head of clauses.

3.1.2 Sample implementation

In this section I will show how to write the world representation. A typical example is the game Tic-tac-toe, but I will explain how to implement a different turn based two-player game. The game has two types of players: an attacker and a guard. As a world we can imagine matrix:

```
c c c c c g
c c c c c g
c c c c c g
c c c c c g
c c c c c g
```

The purpose of the attacker is to reach the rightmost column (in the matrix represented by *g*). And purpose of the guard is to protect the board line. The game is ended when the attacker is captured, the attacker enters the line or the time is up (a specified number of the steps were executed).

Roles

Players are defined through the role relation. For our game the roles are:

```
1 (role attacker)
2 (role guard)
```

This tells us that the game has two players and they will act as the attacker and the guard.

Init state

Init keyword is used to define starting point of the game. In the initial state of our game, we need to define four initial facts.

```
1 (init (position attacker 3 1))
2 (init (position guard 3 6))
3 (init (control attacker))
4 (init (step 1))
```

The first and the second line tells us what the initial position of the attacker and the guard is. The third line tells that in the begin the attacker has control, so he moves first. And the last line initializes the step counter.

Next state

The next keyword is used to define the facts, which will be true in the next state. For example the change of controls can be done like:

```
1 (<= (next (control attacker))
2     (true (control guard)))
3
4 (<= (next (control guard))
5     (true (control attacker)))
```

Legal actions

The rules of the game restricts the actions played in a certain state. By using legal keyword, we can define who and when is able to make a certain action. The attacker can do these actions:

```
1 (<= (legal attacker (move ?dir))
2     (true (control attacker))
3     (true (position attacker ?x ?y))
4     (legalMove ?dir ?x ?y)
5     (distinct ?dir nowhere))
6
7 (<= (legal attacker (move nowhere))
8     (not (true (control attacker))))
```

The first rule enables the attacker to move in the direction *?dir*, if the attacker has control, he is on position *?x*, *?y*, the move is legal (player with this move doesn't leave the playing area) and the direction must be different from *nowhere* (player stays on the same position). The second rule tells us, that the player must stay on the same position, if he doesn't have control.

Moves and does

The does indicates the moves actually done by the player in a particular step of the game. The does relation in rules governs to state update. For example in our case:

```
1   (<= (next (position ?char ?nx ?ny))
2     (does ?char (move ?dir))
3     (true (position ?char ?x ?y))
4     (nextPosition ?dir ?x ?y ?nx ?ny))
```

The second line tells us that the player with the role *?char* did move in the direction *?dir*. So we take his *?x*, *?y* coordinates and update his position to the new one.

Terminal states

The game description contains a set of rules defining terminal states. This indicates, if our game reached terminal state.

```
1 (<= terminal
2   passed)
3
4 (<= terminal
5   timeout)
6
7 (<= terminal
8   captured)
```

Goal

Using the goal keyword we define the reward for the player in final states. The rewards for two players can be inverse (as in zero-sum games), asymmetric or complementary (cooperative games). We define these goals:

```
1   (<= (goal attacker 100)
2     passed)
3   (<= (goal attacker 0)
4     timeout)
5   (<= (goal attacker 0)
6     (captured))
7   (<= (goal guard 0)
8     passed)
9   (<= (goal guard 100)
10    captured)
11  (<= (goal guard 100)
12    timeout)
```

3.2 GDL-II

With GDL one we can describe only finite games with an arbitrary number of players. However GDL can't describe games containing some chance element or games where players have only partial knowledge about the current state of the game. GDL-II, The Game Description Language for incomplete information games [13] is a simple extension of basic GDL. GDL-II has two new keywords. The first is **random** and is introduced as special role, which always chooses random legal move. With this role we can model a chance, such as dice roll or taking a card from the deck. Or for example randomly choose if a road from player's position to shelter is open. The second keyword is **sees**. This relation is specifies when a given player

Keyword	Function
sees(R, P)	defines that player R receives P in next state
random	defines a special player making random moves

Table 3.2: GDL-II extension

sees or percepts any information. For example, the pursuer can see the evader only in the moment when the evader is out from the hiding, or is close enough to the pursuer. The overview of the new keywords is in the table 3.2.

3.2.1 Sample implementation

As an example of game with imperfect information, I will modify game from an example game of GDL 3.1.2. Two small modifications make game much more complex. The first one is that we have marauder that randomly chooses a field and makes it unreachable (this field is blocked). Nothing happens, if that field is already taken by any other player. The player staying on the blocked field must leave it and no other player can access it. The second modification is that players can see each other only if they are in the same row or column (which is not very realistic, but it's sufficient as an example).

Role

Because we have some random behavior in our game, we need to define new role compared to previous example. So now we have 3 roles:

```
1  (role attacker)
2  (role guard)
3  (role random)
```

Next state

In the next state, we have to reflect that our random player have chosen any cell to block.

```
1  (<= (next (blocked ?x ?y))
2     (does random (chooseBlocked ?x ?y)))
```

The second line tells us that a random player has chosen a cell *?x*, *?y* and we have marked this cell as blocked.

Legal

Also we should provide rules how a random player can choose blocked field:

```
1  (<= (legal random (chooseBlocked ?x ?y))
2     (indexM ?x)
3     (indexN ?y))
```

Any cell can be chosen. The second and the third line tell us, that the cell must be in defined ranges.

Goal

Here, only one modification is made and that is definition of the goal value for random player.

```
1  (goal random 100)
```

Sees

Now we define when the player see each other.

```
1  (<= (sees guard (attackerPosition ?x1 ?y1))
2    (true (position attacker ?x1 ?x1))
3    (true (position guard ?x2 ?y2))
4    (or (inRow ?x1 ?y1 ?x2 ?y2)
5      (inCol ?x1 ?y1 ?x2 ?y2)))
```

For guard we define *sees relation* analogously.

3.2.2 Game Checker

Game Checker is a program for checking game descriptions (GDL 3.1 and GDL-II 3.2) for syntax validity (safety, stratification, etc.) and well-formedness (playability, winnability, etc. 3.1.1). The useful functionality of the Game Checker is possibility to traverse the game tree. Two methods are available. The first is simple depth-first search described in Section 4.2 and the Monte Carlo method described in Section 4.4. If the game tree is small, we can run DFS on that game. The Game Checker will calculate calculate average goal rewards and the Game checker also displays minimal and maximal goal values for all roles. This can be very helpful to determine if the game was implemented correctly and how the author thought. If the domain is too big to be explored exhaustively, we can use Monte Carlo method to get at leas particular information about game and its results.

The author of the Game Checker is Stephan Schiffel. The program is available for download at http://www.general-game-playing.de/downloads.html.

Chapter 4

Algorithms

General game players must be able to play every game that can be expressed by GDL. In this chapter I will describe algorithms used in GGP by existing players. At each algorithm I will make brief stop to explain basics of that algorithm.

4.1 A* and IDA*

A* is a widely used algorithm in pathfinding and graph traversal. A* uses a best-first search and finds a least-cost path from initial node to goal node. A* achieves a better performance (according to time) by using heuristics. It uses distance-plus-cost heuristic function $f(x)$ to determine the order in which the search visits nodes in the tree. *Distance-plus-cost* function is sum of two functions. The path cost denoted as $g(x)$ and an admissible heuristic estimate of the distance to the goal denoted as $h(x)$.

Iterative deepening A* (IDA*) is a variant of the A* search algorithm, which uses iterative deepening to keep the memory usage lower than in A*. It is an informed search based on the idea of the uninformed iterative deepening depth-first search. The iteration depth limit is controlled by a cost function. The cost function is evaluated at each node as the sum of actual cost traversed f and heuristic value of reaching a goal h.

4.2 Depth-first search

Depth-first search (DFS) is an uninformed algorithm for traversing or searching a tree or a graph. DFS starts at a given root node and explores as far as possible (reaching goal node, or node without a child). If the goal is not found, the search backtracks, returning to the most recent vertex whose exploration hasn't been entirely completed.

4.3 MiniMax

MiniMax algorithm is widely used in decision making systems. The algorithm is based on assumption that opponent is going to minimize your gain as much as possible. And you are trying to maximize your gain as much as possible. The algorithm is maximizing the

minimal possible gain and it evaluates all states in the game tree and these values are called **Minimax values**. These values are defined as (Russel & Norvig, 2003):

$$Minimax - value(vertex) =$$

$$\begin{cases} utility(vertex) & \text{if } n \text{ is terminate state} \\ max_{s \in Successors(vertex)} Minimax - value(s) & \text{if } n \text{ is max vertex} \\ min_{s \in Successors(vertex)} Minimax - value(s) & \text{if } n \text{ is min vertex} \end{cases}$$

Minimax traverses game tree depth-first 4.2 and search for leaves vertices. From the leaves we obtain utility function for all players (in this case they are equal to Minimax value) and we calculate Minimax values of their parent vertices. Minimax as described here can be used to 2-player, zero-sum games. We can imagine that *Max player* is trying to maximize utility and *Min player* is trying to minimize Max's utility. Minimax can by applied also on n-player games. In its paranoid version all players try to minimize Max's utility. On Figure 4.1 is displayed game tree with evaluated vertices. This game tree is represents 2-player game and its three moves. Max vertices are gray and min vertices are black (filled). All vertices are evaluated according to Minimax definition above. From the example it should be straightforward to see how the MiniMax algorithm works. Minimax algorithm explanation can be found in (Russell & Norvig, 2003) or a more formal description in (Shoham & Leyton-Brown, 2010).

MiniMax is often used in GGP system because it is simple to implement and can be applied to many games. But MiniMax has also several problems for application in GGP. MiniMax can be used only for two (or in his modification for many) players games and zero-sum games with alternating moves. These requirements cannot be always guaranteed.

Unfortunately, its approach for the large game is not sufficient enough, because huge amount of time is required to traverse the whole tree and to calculate the tree values. There are several methods possible to use when reducing the game tree. One of them is to limit the maximal search depth. In which case we obtain smaller tree that can be traversed completely. However, the leaf vertices of this new tree might not be terminal states and thus there is no utility function which we can use to calculate MiniMax tree values. We can deal with this problem by using heuristic function which tells us how good or bad the states are. Another way is to prune game tree. One of the most effective enhancements is *Alpha-Beta pruning*. As the name indicates, this method prunes the game tree so we obtain smaller tree and we can search it deeper. Alpha-Beta pruning identifies vertices that leads to sub-trees which has no chance to change the current game tree value.

4.4 Monte Carlo method

Monte Carlo method is not a specific method, but more a technique. Monte Carlo method relies on repeated random simulations to compute the results. The simplest strategy is to make repeated simulations until the time is up. After that the best move is chosen. Monte Carlo is reweighs vertices to choose better moves.

The most used implementation of Monte Carlo method is Monte Carlo Tree search (MCTS) [4]. It is a best-first search method which uses stochastic simulations. MCTS can be applied to any game of finite length. Its basis is simulation of games where both

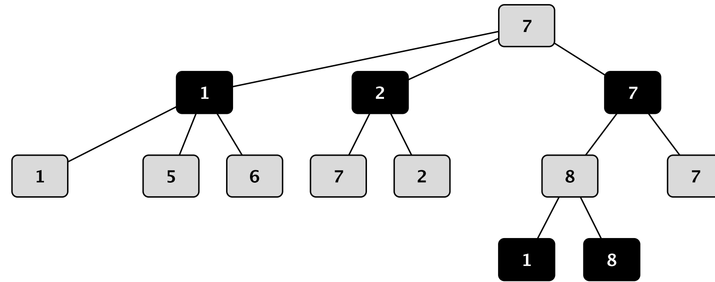


Figure 4.1: Minimax Game tree

AI and its opponents make random move, or better pseudo-random moves. From single game, where player choose their moves randomly, very little can be learnt. But from lots of simulations good strategy can be inferred. The algorithm builds and uses game tree. The game tree is build by repeating these steps:

selection - At the beginning we need to select vertex in simulation tree starting from the root vertex. The selection is repeated until the leaf vertex is reached. The selection of vertex is done according to how much we want to explore the game tree or how much we want to exploit informations that we obtained.

expansion - In this step we expand our simulation tree by adding one or more vertices. These vertices shouldn't have been previously part of the simulation tree. We can add vertices only if it passes some condition (e.g. minimal number of visits of the vertex).

simulation - In this step we simulate the rest of the game from the leaves of the simulation tree. Vertices can be selected randomly or pseudo-randomly (for this we need domain specific heuristic).

backpropagation - After the simulation is finished we have to propagate results back through the simulation tree. Each vertex that was a part of the simulation and leads to terminal state should be updated.

These steps are illustrated in Figure 4.2.

4.5 Upper Confidence Bounds applied to Trees

UCT [7] is a variant of the Upper Confidence Bounds algorithm (UCB1). Simply it is UCB applied to trees. UCB1 is a simple but effective way to balance exploration and exploitation. It solves exploration-exploitation tradeoff. It keeps track of the average returns of all available actions $a \in A$ at time t and samples one with the highest upper confidence bound given as:

$$a_t = v_i + C\sqrt{\frac{\ln N}{n_i}}$$

Where v_i is value of i-th vertex (usually average value of previous simulations), n_i is number of visits of vertex i, N is number of visits of parental vertex if *i-th* vertex and finally

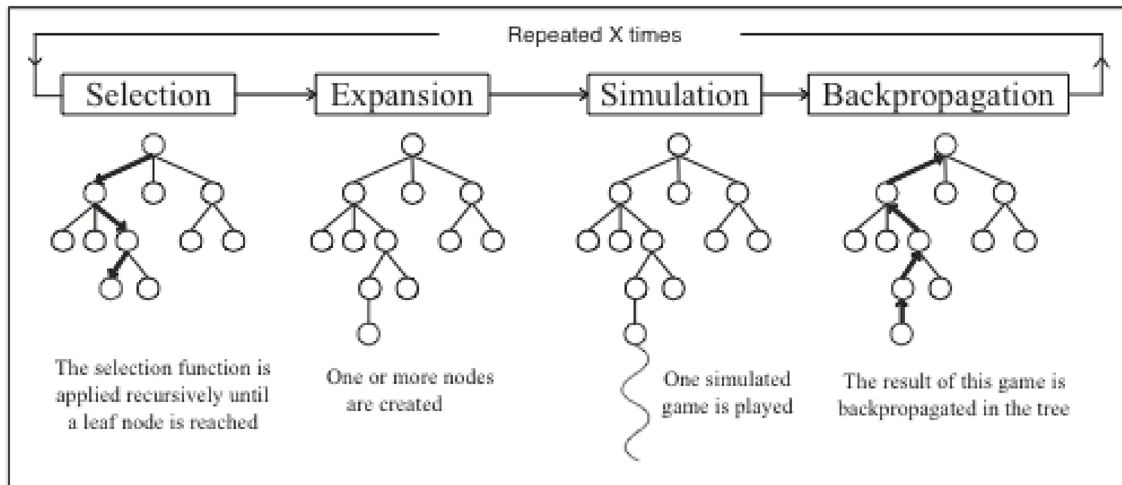


Figure 4.2: (Chaslot, Bakkes, Szita & Spronck, 2008): Monte Carlo Tree search control loop

C is coefficient defining how much we prefer exploration over exploitation. If there exists action that has never been selected, the algorithms default behavior is to prefer this action before any previously sampled. The confidence bound can be described as average estimate value of taking an action plus UCB bonus. The bonus is calculated with respect to vertex visits, the actions gradually build up their bonus when they are not selected and each time they are selected the bonus drops. When the number of visits is low, the action with best estimate value is selected for sampling, but in time the bonus of suboptimal action ensures that this action may be also selected. If the actions continue to look suboptimal, they need to rebuild their bonus value to be considered again. Each time the bonus takes longer to be rebuilt. But if actions average value rises, than they are chosen more frequently for sampling. The selection function chooses action from confidence distribution. We exploit the best action until the number of samples in it generate certain level of confidence in its estimate return value. If the suboptimal action is selected, it means that the confidence of the best action is high enough that is better to lower uncertainty of the estimate value of the suboptimal actions. UCT algorithm is commonly used in the selection step of MCTS algorithm.

Chapter 5

Successful players

5.1 Fluxplayer

The Fluxplayer [12] is one of first successful players. Its authors Stephan Schiffel and Michael Thielscher won with Fluxplayer GGP Competition in 2006 and since than they stands on top places.

The author's focus is on techniques for constructing search heuristics by the automated analysis of game specification. The reason is that state space in most games is too big to be searched exhaustively, it is necessary to bound the depth of the search and to use some heuristic evaluation function for non-terminal states. It is not possible to provide a heuristic function that depends on features specific for the concrete game at hand. Therefore the heuristics function has to be generated automatically at runtime by using the rules given for the game. The main idea of Flux's heuristic is evaluation function calculating the degree of truth of the formulas defining the predicates *goal* and *terminal* in the state to evaluate. The values for *goal* and *terminal* are combined in a way that terminal states are avoided for as long as the goal is not fulfilled. The value of *terminal* state has a negative impact on the evaluation of the state while *goal* has a low value and a positive impact. A Fuzzy-logic is used for implementation. This is part of theory about expressing dynamical domains in first-order logic, that is called Fluent calculus and was introduced by Michael Thielscher (1998).

To search a game tree Flux uses non-uniform depth-first search with iterative deepening. Depth-first search is described in previous Section 4.2. Flux also uses two enhancements, transposition tables for caching the value of states already visited during the search and history heuristics for reordering the moves according to the values found in lower-depth searches. The player is chooses additional pruning technique according to game type (single-player, multi-player, zero-sum game, non-zero-sum game). The player adjusts its game tree according to turn-based or simultaneous moves.

You can see all result of this player in GGP competition in Table 2.1.

5.2 CADIAplayer

CADIA [1] is one of the most successful GGP players. CADIA won competition in years 2007 and 2008. Since that CADIA stands on podium.

The game-playing engine is written in C++ and can be split up into three conceptual layers: *the Game-Agent Interface*, the *Game-Play Interface* and the *Game-Logic Interface*.

The Game-Agent interface handles external communications and manages the flow of the game. It also includes a game parser for building a compact internal representation for referencing atoms and producing KIF strings, both needed by the Game-Play interface. The parser also converts received moves from the GM to the internal form.

Game-Play is the main AI part of the agent responsible for its move decisions. Cadia does not require any a priori domain knowledge nor does it use heuristic evaluation of game positions. Instead, it relies exclusively on Monte-Carlo based simulation search for reasoning about its actions, but guided by an effective search-control learning mechanism. It runs memory-enhanced IDA* search algorithm 4.1 on the *startclock* for single-player games. First solution with points is stored and replaced if better solution found to remember best found solution. If during the *startclock* is any partial solution found CADIA keeps in using IDA* in the *playclock*. If no solution is found on *startclock* CADIA switches to UCT search 4.5 because its Depth-first search 4.2 has chance of hitting some return that might guide the search better.

In the Game-Logic interface the state space of the game is queried and manipulated. The Game-Logic interface encapsulates the state space of the game, provides information about available moves, and tells how a state changes when a move is made and whether the state is terminal and its goal value. It is also called *Game controlled*. Cadia is using YAP [14] (Yet Another Prolog) for reasoning. YAP is a high-performance Prolog compiler. YAP is mainly used because it is free for academic use, reasonably efficient, and provides a convenient interface for accessing the compiled library routines from another host programming language.

5.3 Ary

In years 2009 and 2010 Jean Méhat and Tristan Cazenave won GGP competition with their Ary player [5]. Jean Méhat published infrastructure of Ary player called Subplayer.

The player is using Prolog for interpreting GDL. It translates game rules from GDL into Prolog, and transmits them to the Proglog interpreter. This interpreter is than used as interface for obtaining: *legal moves*, *applying moves*, *detecting end of game*, *determining the score for each player*.

In the past Ary was using simple Monte Carlo method 4.4 to explore the game tree. The player is doing random moves until end of game is reached. In the end of thinging time (Playclock limit), move with best mean reward is played. Since finals 2007 the player uses MC extension called Monte Carlo Tree search 4.4. Ary uses UCT 4.5, to balance between exploration (deepening the search in promising branches) and exploitation (scouting underexplored subtrees).

Chapter 6

Problem solving

In this chapter I will introduce, implement and test a game called Pursuit-evasion game. I chose this game, because it is one of the most common real life problems, which can be solved. Part of this problem is path finding, units positioning and agents cooperation.

Pursuit-evasion (PE) game is family of the games where one group follows other group, which attempts to avoid its capturing. PE game can by modeled geometrically or on graph. Geometrical representation is used for continuous environments and representation on graph for the discrete. Form of euclidian plane is typically used for continuous representation. But for euclidian system we need to define real numbers. But real numbers are problem, because their definition and definition of operations on real numbers in GDL is the complex task. Other problem is efficiency of players with definition of real numbers. This is the main reason why it is better to use representation on graph.

If we want to use the general player for playing some game, we need to define the rules and the entire world representation for the player. Almost every problem has to be simplified, we need to create a game model. We have to consider, which game behaviors are important for our solution. In our case, the pursuers and the evaders weight and size is irrelevant. On other side, in PE game the important factor is speed. The velocity can be modeled as a number of vertices visited during one turn. Other model simplification is distance discretization. I mentioned how velocity can be modeled, but how the vertices models distance?

We have several options how the world can be discretize. For example we can cogitate, that every street is an edge of the graph. Its cost represents the length of the street. The unit can move from one vertex to another, only if it has enough move points. The move points are points, which the unit accumulates, when it waits on the same position. Such model has some disadvantages. The first is that unit must stay on the same position and wait until enough points is accumulated. And the second disadvantage is that the unit cannot change its direction during its transfer from one vertex to other. There is another solution for discretization. We can dismember the streets to some number of pieces. Each of the pieces is represented by graph vertex and the units moves from one piece (vertex) to another, only if these vertices are connected with an edge. An example representation is on Figure 6.1.

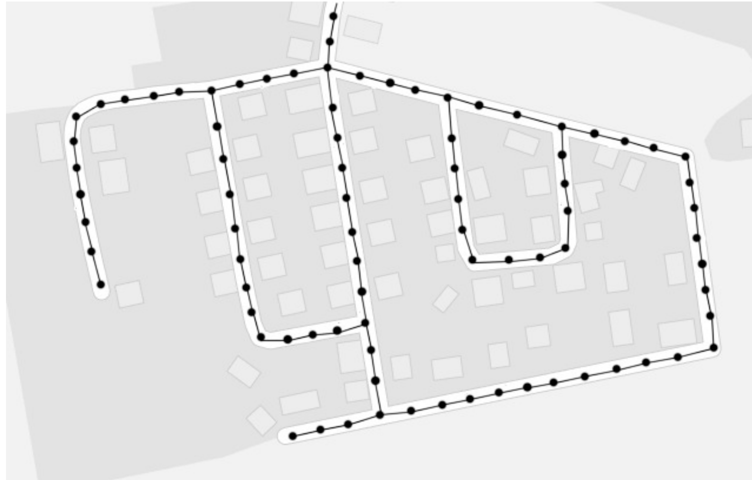


Figure 6.1: World discretization using graph representation.

6.1 Game descriptions

In this section I will describe how the PE game on graph can by implemented in GDL. First I will describe one game and later I will create modifications of this game. All modification will be described and I will discuss why that modification is interesting.

First of all we should assemble our idea of the game model. In introduction of this chapter, I decided I would use graph representation of the world. Next important decision is the unit's velocity. We will consider that all units moves with the same velocity, it means that they can move from one vertex to adjacent one each turn. The movement of units is simultaneous. It means that all units move in same moment. We could use turn taking move, but simultaneous move is more resemble to real life movement. To make the game more symmetric I created point called escape point. If the evader enters this point it escapes from all pursuers and the evader wins the game. The last important thing is adding step counter to ensure that the game is terminal. If several number of steps is made the game ends (the game reached timeout) and we can say that evader was partially successful. The evader didn't escape, but it wasn't caught too. This is the reason why the evader obtain higher points when the game timeouts.

Now that we have thought through how the game model will look like, we can proceed to implementation.

Basic game - Pursuer's with separate goals

The environment of the game is static (no vertices are added or removed during the game) so we can define the vertices and the edges as an atoms. The vertices are represented as:

```
(vertex <id>)
```

Where *<id>* is unique identifier of the vertex. The edges are represented as:

```
(edge <from> <to> <value>)
```

Where *<from>* is represents the vertex from which the oriented edge is leading to the vertex with id *<to>*. And *<value>* is cost of the edge. Next we need to define how player can determine that two vertices are adjacent. Because we are defining game on undirected graph, we can safe many lines in GDL description by defining only one direction of the edge and detect adjacent vertices in the function. The function can look like:

```
1  (<= (adjacentVertex ?V1 ?V2)
2    (vertex ?V1)
3    (vertex ?V2)
4    (or (edge ?V1 ?V2 ?Cost1)
5      (edge ?V2 ?V1 ?Cost2)))
```

Where the second and the third line ensures that *?V1* and *?V2* are the existing vertices. And the fourth and the fifth line says that edge from *?V1* to *?V2* or from *?V2* to *?V1* must exist. The cost parameter is meaningless for now. According to this adjacency definition, we can define only one direction between vertices, but now we must use this function for the adjacency control. Otherwise we obtain directed graph. As we can see the definition of graph in GDL is quite easy and straightforward. Now other aspect of the game need to be defined, the aspects are *roles*, *initial state*, *legal* actions ,*next* relation, *terminal* states and *goal* rewards.

The roles are changing in dependence on the game. But in this basic game I use three roles, one evader and two pursuers, each controlled by one player. So we have:

```
1  (role evader1)
2  (role pursuer1)
3  (role pursuer2)
```

When we have roles, than we can define what actions each role can do. We have two roles playing as pursuer, because these roles behave the same way we can save work by defining a relation *pursuersTeam* and only one implication for pursuers *legal move*. This can be done this way:

```
1  (pursuersTeam pursuer1)
2  (pursuersTeam pursuer2)
3
4  (<= (legal ?purs (move ?purs ?NewV))
5    (pursuersTeam ?purs)
6    (true (position ?purs ?V))
7    (adjacentVertex ?V ?NewV)
8    (not (true (escapePoint ?NewV))))
9
10 (<= (legal ?purs (move ?purs ?V))
11   (pursuersTeam ?purs)
12   (true (position ?purs ?V))
13   (not (true (escapePoint ?V))))
```

From this definition we can see, that the pursuer can move to every adjacent vertex or stay on the same position, only if the vertex where the pursuer wants to move is not the escape point. The movement of the evader is similar, but without restriction for stepping onto escape point and without control if the player is really evader.

```
1  (<= (legal evader1 (move evader1 ?NewV))
2    (true (position evader1 ?V))
3    (adjacentVertex ?V ?NewV))
4
5  (<= (legal evader1 (move evader1 ?V))
6    (true (position evader1 ?V)))
```

When the actions are made, we need to define how the game state changes. This is done by *next* relation. We need to bring three things to the following state: players positions, escape point and state counter. All these three thing can be done by:

```
(<= (next (position ?char ?V))
  (does ?char (move ?char ?V)))

(<= (next (escapePoint ?V))
  (true (escapePoint ?V)))

(<= (next (state ?NewN))
  (true (state ?N))
  (inc ?N ?NewN))
```

The first implication is updates players position. Because the names of the roles and the names of the players identifying their position are identical, we can make position update easily. The second implication just transfers information about escape point from the current state to the next state. And the third implication transfers to the next state the state counter increased by one. The *inc* relation must be defined, otherwise no information about state counter is transferred to the next state. The *inc* relation looks like:

```
(inc 1 2)
(inc 2 3)
...
(int N-1 N)
```

Now we will define how the initial state looks and how the GM and the players can determine that the state is terminal. First we will define the initial positions of players, where the escape point is and we will initialize the state counter.

```
(init (position evader1 0))
(init (position pursuer1 10))
(init (position pursuer2 110))

(init (escapePoint 60))

(init (state 1))
```

For declaring terminal states we need to define some additional functions, such as *captured*, which is true, if one or all pursuers are in same position as the evader, *escaped*, which is true, if the evader escaped (the evaders position is same as position of the escape point) and *timeout*, this is true, if some number of moves were made.

```
(<= (escaped)
  (true (position evader1 ?V))
  (true (escapePoint ?V)))

(<= (captured)
  (true (position evader1 ?V))
  (or (true (position pursuer1 ?V))
    (true (position pursuer2 ?V))))

(<= (timeout)
  (true (state 100)))
```

When we have these functions we can define terminal states. All three functions represent terminal states of the game.

```
(<= terminal
  (captured))

(<= terminal
  (timeout))

(<= terminal
  (escaped))
```

If the state is terminal, the reward for the players is calculated. In this game the evader obtains 100 points if he escapes, 50 points if 100 moves were made (timeout terminal state reached) or 0 points if one or both pursuers caught him. Pursuers have inverse evaluation. The pursuer which caught the evader obtain 100 points, 25 if 100 moves were made or 0 if the evader escaped.

```
(<= (goal evader1 100)
  (escaped)
  (not (timeout)))

(<= (goal evader1 0)
  (captured)
  (not (timeout)))

(<= (goal evader1 50)
  (timeout))

(<= (goal pursuer1 100)
  (true (position pursuer1 ?V))
  (true (position evader1 ?V))
  (not (timeout)))

(<= (goal pursuer1 0)
  (true (position pursuer1 ?V1))
  (true (position evader1 ?V2))
  (distinct ?V1 ?V2)
  (not (timeout)))

(<= (goal pursuer1 25)
  (timeout))
```

The goal relation for the pursuer2 is identical with goal relation the pursuer1.

The complete implementation of the game is in Section B.0.3.

Mod 1 - Pursuer's common goal

In the previous section I introduced PE game where every pursuer played for himself. An interesting modification is to change the goal relation, so the pursuers obtain 100 points when the evader is caught no matter who caught him. With this modification, pursuers should cooperate or at least be more cooperative. We run this modification also for situation where the general player plays as the evader. The reason is that Cadia player is somehow models opponents. And I want to investigate how this modeling will change and what impact it will have on the game. This modification is done by very simple adjustment of the goal relation.

```
(<= (goal pursuer1 100)
  (captured)
  (not (timeout)))

```

```
5  (<= (goal pursuer1 25)
6    (timeout))
7
8  (<= (goal pursuer1 0)
9    (not (captured))
10   (not (timeout)))
```

Whole modification is done by changing lines:

```
(true (position pursuer1 ?V))
(true (position evader1 ?V))
```

onto:

```
(captured)
```

and

```
(true (position pursuer1 ?V1))
(true (position evader1 ?V2))
```

onto:

```
(not (captured))
```

These modifications were made for both pursuers.

Mod 2 - Pursuers controlled by single player

Other interesting modification can be, if all pursuers are controlled by one player. We can imagine the we have a spectator in the helicopter watching ground units and the spectator is navigating friendly units. I will run only games where Cadia is playing for pursuers team. The reason is that we can simulate this behavior by own heuristic.

In previous games the only action for players was *move*. Every player response with the *move* command and the vertex where he wants to move e.g.:

```
(move pursuer1 11)
```

But if one player controls more players we need to modify the move relation. Now it will look like:

```
(move pursuer1 v1 pursuer2 v2)
```

This moves *pursuer1* to the vertex *v1* and *pursuer2* to the vertex *v2*. To obtain this move relation we need to modify legal and next relations.

```
(<= (next (position evader1 ?V))
  (does evader1 (move evader1 ?V)))

(<= (next (position pursuer1 ?V1))
  (does pursuers (move pursuer1 ?V1 pursuer2 ?V2)))

(<= (next (position pursuer2 ?V2))
  (does pursuers (move pursuer1 ?V1 pursuer2 ?V2)))

(<= (legal evader1 (move evader1 ?NewV))
  (true (position evader1 ?V))
  (or (adjacentVertex ?V ?NewV)
    (sameVertex ?V ?NewV)))

(<= (legal pursuers (move pursuer1 ?NewV1 pursuer2 ?NewV2))
  (true (position pursuer1 ?V1))
  (true (position pursuer2 ?V2))
  (or (adjacentVertex ?V1 ?NewV1)
    (sameVertex ?V1 ?NewV1))
  (or (adjacentVertex ?V2 ?NewV2)
    (sameVertex ?V2 ?NewV2))
  (not (true (escapePoint ?NewV1)))
  (not (true (escapePoint ?NewV2))))

(<= (sameVertex ?V ?V)
  (vertex ?V))
```

The next relation for the evader is unchanged, but for pursuers we have to fork implication into two new implications. From the code above should be straightforward how the position of the pursuers is updated. The legal relation is modified for both. We are not able to define legal move for pursuers in the same way as in previous games. It is possible for evader, but to keep the description more consistent, we use the same rules for both. The only thing which can be unclear is *or* relation, this just says that the player can move to any adjacent vertex or stay in the same position. The last modification is about the goal rewards for the players.

```
(<= (goal evader1 100)
  (escaped)
  (not (timeout)))

(<= (goal evader1 0)
  (captured)
  (not (timeout)))

(<= (goal evader1 50)
  (timeout))

(<= (goal pursuers 100)
  (captured)
  (not (timeout)))

(<= (goal pursuers 0)
  (not (captured))
  (not (timeout)))

(<= (goal pursuers 25)
  (timeout))
```

6.2 Domain player implementation

To be able to compare general player with domain specific player, we need to secure communication between these players. The easiest way is to use already existing structure of the general player. I decided to use BasicPlayer a part of the Palamedes IDE [6]. The main reason is already implemented communication and move parser. I created basic environment for playing PE game against the general players. It consists of 5 main classes: *Game*, *PEGPlayer*, *PEGPlayer2*, *PEGStrategy* and *Graph*. The overview and description is summarized in the Table 6.1.

Class	Description
Game	This class provides all informations about game, updates its state.
PEGPlayer	Player for *basic* and *pursuers common goal* games.
PEGPlayer2	Player for games where pursuers are controlled by one player.
PEGStrategy	Abstract class, all strategies should inherit this class.
Graph	Data structure to hold graph representation.

Table 6.1: Domain player classes

6.2.1 Domain players

It is not so much about the players, but rather about their strategies. I have created 3 simple heuristic strategies, one for the evader and two for the pursuers. They are not intended to be used in real application. Their purpose is to show, that working strategy can be written very quickly and without knowledge of sophisticated algorithms. All strategies during *startclock* calculate distance between all vertices. This is not the best way, but how we will see during the experiments time needed to calculate all distances is still much shorter than time, which Cadia player needs for enough sufficient playing. Also during the *startclock* the map is loaded to the Game class and this game is associated to the player or better to the strategy.

- **Evader**: It's strategy is very simple. Every time the player is requested for the next move, it takes the position of all pursuers and it marks all vertices in distance of 2 as dangerous. Two is chosen because evader can get close to pursuer (sneak around him to the target, usually to the escape point), but still stay far enough. When dangerous vertices are marked, it takes all possible and safe moves. From these moves evader takes that one, which moves him closer to the escape point.

- **Follower**: This strategy follows one evader. It chooses the closest one and it keeps following him until the end of the game. In the beginning of each move it calculates where the evader would move. This is done by taking all possible moves of the evader and choosing one with the smallest distance to the escape point. This point, we can call WhereToMove. The next step is the move chosen. This is done the same way except the target is not the escape point, but WhereToMove. As next move vertex with the smallest distance to the vertex called WhereToMove is selected. The strategy makes also random move. The reason for it is that the strategies are not communicating between each other. If two players with same strategy are in the same position, they

will do the same moves. In this situation we will actually loose one player. For that reason the pursuers move randomly, if they detect that two of them are in the same position.

- **Blocker**: This strategy tries to prevent the evader to escape. It takes all the adjacent vertices from the escape point - lets call then access points. Than the distance to the closest evader is calculated from each access point. The access point with the smallest distance is chosen, because it is the most vulnerable. We will call this vertex as WhereToMove. As next move is selected vertex with the smallest distance to the vertex called WhereToMove.

6.3 Experiments

In this section I will introduce specific scenarios of PE game, I will run matches for those scenarios with the roles occupied by different players. I will discuss each scenario and its results.

Tested general player is Cadia 5.2 in version *2.0.1* from June 8th 2001. Cadia player is running on i7 3.2 GHz processor. The domain players are running on i5 2.3 GHz processor and memory limited to 512 Mb.

Each game has both clocks (*startclock* and *playclock*) set to 10 seconds. The reason why I chose 10 seconds will be discussed in Section 6.5.

I used standalone server called *GameController* to run test matches. To be exact I used *gamecontroller-gui-r495.jar* and *gamecontroller-cli-r495.jar*. Both programs can be downloaded as jar file or source code from: http://sourceforge.net/projects/ggpserver/.

The Table 6.2 shows overview of experiments which I will run .

	Environment	**Game type**
Scenario 1 (6.3)	Grid 11x11	Separate goal (6.1) Common goal (6.1) 2-player (6.1)
Scenario 2 (6.3)	Grid 31x31	Separate goal (6.1) Common goal (6.1) 2-player (6.1)
Scenario 3 (6.3)	Rugged map	Separate goal (6.1) Common goal (6.1) 2-player (6.1)

Table 6.2: The experiments overview

Scenario 1 - Small Grid

In this scenario environment is represented as a grid. The vertices of grid are numbered from zero to 121. The numbering begins on top left corner and continues line by line (6.2). The evader starts at the vertex labeled with the number **0**, one pursuer is at the position **10** and the second is at the position **110**. The escape point is in the middle of the grid (the vertex

60) so each player has the same distance to the escape point. When we run Game Checker 3.2.2 with Monte Carlo method limited to 60 seconds we obtain average rewards.

	States checked	Terminal states	avg. rewards
Separate goal	276912	2914	38.6, 30.39, 31.16
Common goal	275350	3856	38.35, 52, 52
2-player	164271	2346	37.63, 52.77

Table 6.3: Grid 11x11, Game Checker evaluation (rewards are listed in order: evader1, pursuer1, pursuer2)

From numbers in Table 6.3 we can see that the game goal rewards are fairly balanced for the *separate goals*. In the game with *common goal* evaders average points are similar to point earn in the game with *separate goal*. But this is not surprising, because evaluation of the terminal states is not changed for the evader. The other thing is how this change takes effect during the game play. The last row shows numbers for game where all pursuers are controlled by one player. Here it look that the game is more optimistic for the player controlling the pursuers. But it can be caused by less explored states and typical behavior of MC method 4.4, which is random sampling.

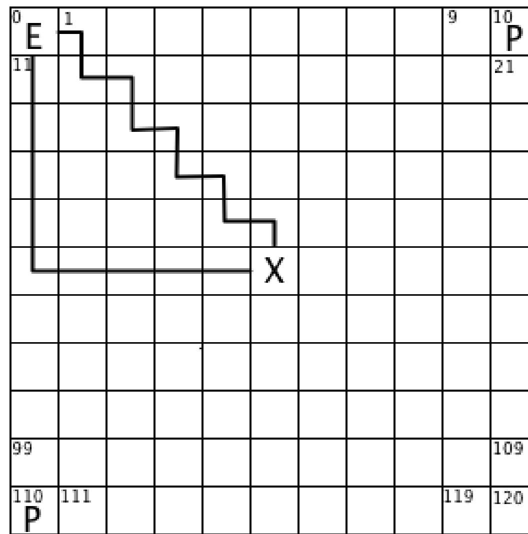


Figure 6.2: Grid (11x11) representation with shortest path to escape point highlighted. (E - evader, P - pursuer, X - escape point)

The shortest way from the evaders initial position to the escape point is 10 steps long, so evader can escape in state 11. But this can happen only if pursuers are playing really bad and don't cut this shortest path. The shortest way is displayed in Figure 6.2. Only two paths are displayed, but other two are symmetrical along the main axis.

We can say that the general player should be able to play this game sufficiently. The domain is simple and its game tree is not big. But in the concrete implementation, the general player can have problem to explore dangerous situations, which occurs when the

evader is moving along the shortest path. For the pursuers it means that they must cut the shortest path to escape point.

Scenario 2 - Big Grid

This scenario is similar to the previous Scenario 1 - Small Grid. The only difference is that the matrix dimensions are 31 rows and 31 columns. The evader begins at the position **0**, one pursuer at the vertex with id **30** and the second at the vertex **930**. Also the escape point is in the middle of the matrix (the vertex **480**).

Game Checker evaluation in Scenario 1 - Small Grid was running MC method for 60 seconds, but this game is much bigger so I let MC method run 10 times longer. It is 10 minutes and we obtained numbers listed in Table 6.4. From the average rewards we may assume, that the players playing for evader's role will be winning.

The shortest path is of course longer than in previous scenario. Now the path is 30 steps long. The shortest path is identical to the paths in Figure 6.2, but scaled for dimension 31x31.

I'am testing this scenario, because I want to see how the general player will handle bigger environments.

	States checked	**Terminal states**	**avg. rewards**
Separate goal	692342	6994	49.95, 24.98, 25.06
Common goal	663347	7185	41.62, 37.57, 37.57
2-player	293933	2969	49.97, 25.05

Table 6.4: Grid 31x31, Game Checker evaluation

Scenario 3 - Rugged map

I created this scenario to test players in environment more similar to real problems. There is only very small probability that we will solve PE game on open space as it is in previous scenarios. We can imagine that in one part of a town, a thief robbed the house and got into the car (on position **54**), but the house alarm notified the police station (on position **0**). From the town leads only one path and if the thief reaches this point (on position **57**), he will escapes to the police. The police of course know this as well and their goal is to catch or at least avoid escaping of the rubber. The initial positions are displayed in Figure 6.3.

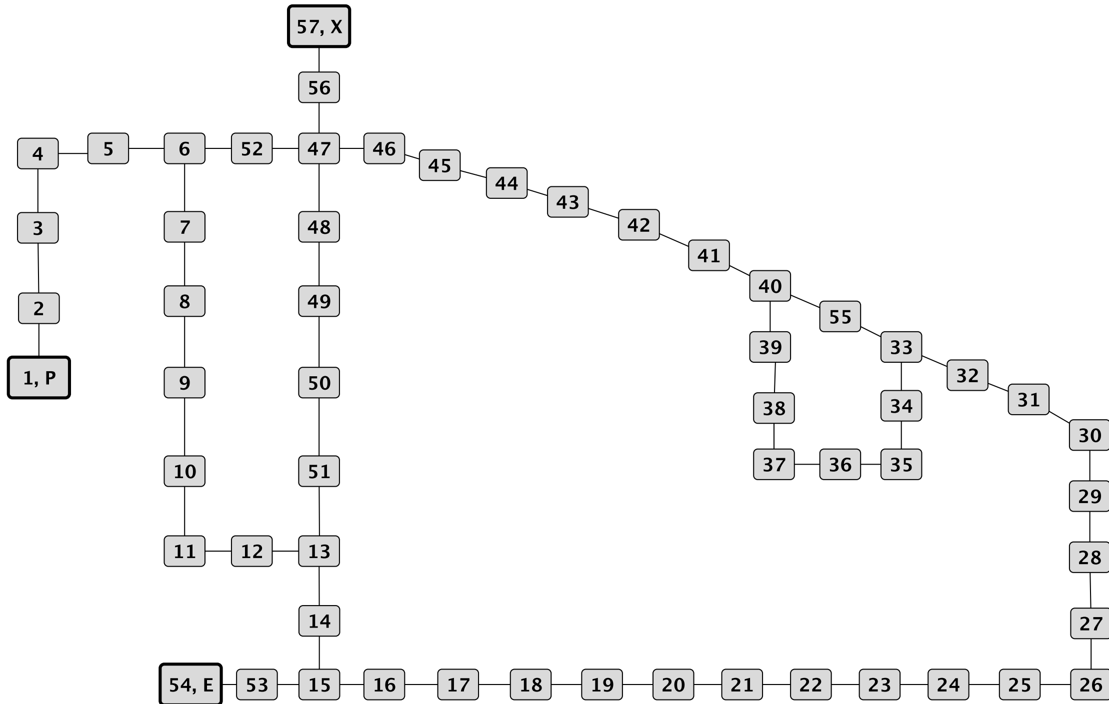


Figure 6.3: Rubber and police, town representation

As in previous scenarios, I will discuss Game Checker test results. Game Checker was running Monte Carlo method for 60 seconds and he provided result in Table 6.5. For the game with *separate goals* the results look balanced. But for the game with *common goal* it looks better for pursuers. But the reason is that the pursuer take reward even when the evader is caught by the second pursuer. And the third case is similar to the previous, because player controlling pursuers is rewarded in a similar way. But other thing should be noticed in game with 2-players. And it is a number of explored states. The number is smaller by almost 150000 states. From this we can consider that this game will be more difficult for players.

	States checked	**Terminal states**	**avg. rewards**
Separate goal	563510	6974	32.26, 34.61, 33.54
Common goal	561762	6924	32.85, 53.4, 53.4
2-player	424474	5248	32.03, 54.57

Table 6.5: Rigged map, Game Checker evaluation (rewards are listed in order: evader1, pursuer1, pursuer2)

6.4 Results

As a first thing in this section I will comment the Table 6.6 and what we can find in that table. The table is divided by the game type (Separate goals, Common goal, 2-player), game environment (Grid 11x11, Grid 31x31, Town map) and the roles assigned to the players. The abbreviations of player's names are described in Table 6.7. The columns with average steps also contains information about standard deviation of the samples.

The average values were calculated from 5 samples for each game. And the players had 10 seconds for thinking before the match begun and 10 seconds each turn to make their decision (*startclock* and *playclock*).

		Separate goals		**Common goal**	
		avg. steps	**avg. score**	**avg. steps**	**avg. score**
Grid 11x11	**C, F, B**	32 (32.6)	80, 20, 0	40 (19.9)	80, 20, 20
	C, F, F	22.4 (13.7)	80, 20, 0	16.2 (2.3)	80, 20, 20
	E, F, B	100 (0)	50, 25, 25	-	-
	E, F, F	35.8 (11.8)	60, 40, 20	-	-
	E, C, C	13 (2.8)	40, 60, 0	19 (7.4)	20, 80, 80
Grid 31x31	**C, F, B**	89.6 (14.5)	50, 35, 15	92 (17.9)	60, 20, 20
	C, F, F	65.2 (21.3)	90, 5, 5	100 (0)	50, 25, 25
	E, F, B	31 (0)	100, 0, 0	-	-
	E, F, F	31 (0)	100, 0, 0	-	-
	E, C, C	31 (0)	100, 0, 0	100 (0)	50, 25, 25
Town map	**C, F, B**	48 (13.5)	0, 100, 0	42.2 (5.2)	0, 100, 100
	C, F, F	33.2 (5.9)	100, 0, 0	36 (1)	100, 0, 0
	E, F, B	100 (0)	50, 25, 25	-	-
	E, F, F	20.2 (4.6)	100, 0, 0	-	-
	E, C, C	24.2 (2.9)	0, 80, 60	21.4 (12.8)	20, 80, 80

		2-player	
		avg. steps	**avg. score**
Grid 11x11	**E, C**	15 (8.4)	80, 20
Grid 31x31	**E, C**	78 (31.7)	50, 35
Town map	**E, C**	17.6 (1.1)	0, 100

Table 6.6: Experiments results, containing average steps, steps standard deviation (in brackets) and average goal rewards. Roles ids described in Table 6.7.

Identifier	**Player**
C	Cadia player (GGP)
E	Evader (General player)
F	Follower (Domain player)
B	Blocker (Domain player)

Table 6.7: Player's identifiers description

Separate goals

In the Scenario 1 - Small Grid Cadia is very successful in evader's role. If we compare my evader playing against follower and blocker or two followers, we find that Cadia has better results, but our player has more consistent results. In Scenario 2 - Big Grid has little worse results. But it was expected, because the domain was bigger. And the players had the same time to make theirs moves, so they were not able to evaluate game states, as well as in smaller grid, where the players reach to terminal states much sooner. Despite the fact that the game tree is bigger, Cadia was playing very well against the followers in bigger grid. The reason why Cadia is playing worse against follower and blocker in bigger grid than in smaller is domain size again. Cadia has simply problem to find the best way how to bypass blocking player and enter the escape point. My evader in bigger grid is playing much better than Cadia, but it is not surprising. Because for domain player is actually everything same. The only thing which we can observer is, that there is also longer time needed to calculate distances between all vertices.

An interesting results are obtained on the Town map. Cadia playing against follower and blocker behaves unreasonable. The blocker blocks the entrance to the escape point. So Cadia should evade the follower. But Cadia in some moment stays on the same position until it gets caught by the follower. The reason for this behavior may be wrong evaluation of the reward, when the game timeouts (in our case an 100 moves were made).

Common goal

If we compare the results of this game with the game with separate goals, the goal modification did not bring that big changes of the results and of the rewards. The only exception is game where Cadia was playing against two followers in the bigger grid environment. Cadia was in game with separate goals diametrically better. The reason can be unpropriate modeling of opponents. But it can be caused also by my heuristic players. Because their heuristic can by illogical and this will confuse Cadia player.

The reason why the results are missing in result table for the games where my domains players are facing each other is that the heuristic will be the same as in previous game and they wont consider the change of the goal values.

2-player

This game should test, if Cadia takes an advantage of controlling both pursuers. This is also reason why I did not simulate the case where my domain player controls all pursuers.

Little surprising is that Cadia has problems to play sufficiently with both players. The results in grid environments are not good at all. And the interesting thing is that Cadia was playing better on the bigger grid than on the smaller. But if we look to results on Town map, we can see that the results are very good. From this we can consider, that the problem will be the branching factor on the grid. [1] If we look to process of the games on grids, we will notice that often is moving only one pursuer. The reason can be under evaluated moves where both pursuer will change their position, or even not evaluated these moves at all.

[1] All game logs are available on enclosed CD in folder *game_logs*.

6.5 Summary

In this section I want to evaluate all the way from idea to play PE game to the results. I will summarize complexity of each step and as last think in the section I will discuss obtained results.

In Section 6.1 we created model of the problem. As it can be seen, that wasn't tough task. But we should not forget at the design time of the problem model to properties, limitations and requirements of GDL. Such is playability, termination, etc. The implementation of the problem in GDL is not usually difficult, but we should be very careful how we implement the problem. Before starting the game with the general player it is very handy to test the game using for example Game Checker mentioned in Section 3.2.2 or online utility for checking games created by Michael Genesereth for GGP course on Stanford University http://logic.stanford.edu/classes/cs227/2012/index.html (Class cs227). On those site, there can also be found GDL stepper. The stepper is very useful, if we want to check some special cases of the game and if we defined its behavior correctly. Although, it is very useful utility, I don't recommend it's use for bigger environments.

The next step was to create the domain players. Here I saved some work using existing general player. I used it's communication protocol. But to be able to play the PE game, I had to create game environments interpreter, which holds, updates and provides information about the game state to the player. It was also necessary to create own environments loading, because parsing from GDL description would take much more time. Many information in the game description is useless for domain player, such as legal moves definition, goal reward etc. All these thinks were ensured directly in the domain player.

One of the difficult tasks was to obtain and make running the general player. Only one general player was freely available. Luckily it was Cadia player which is one of the best players at all.

When I was preparing the experiments, I ran many pretest to find out the best configuration. This mainly concerned the setting *startclock* and *playclock*.At first I run the game with 5 seconds for *startclock* and 1 for *playclock*, As I supposed the results was totally unsatisfactory. After a few tests, I decided to use 10 seconds for both clocks. The reason is that the player was already playing sufficiently and the game finished in reasonable time (approximately 10 mites for one game).

When we take in account time needed to learn GDL and procure the general player, it would be faster to create all PE game. But on the other side, if we already know GDL and have working player, than usage of this player as support act can give us minimally partial results. This results may tell us how difficult it would be to create own domain player and how difficult the problem is for solution. This is the reason why I think that the general player in this test succeeded.

Chapter 7

Discussion

The reason why we should want to use GGP as libraries for solving problems is saving of time, Now we will discuss if the GGP system is already sufficient enough and if it provides easy way to implement the problem. First of all I will summarize advantages and disadvantages of general players and comment them, then I will try to identify improvements which could help to better usage of the general players. As the last thing I'll try identify some more problem which could be solved by the general players.

Advantages	Disadvantages
No algorithmic knowledge needed	Unavailable public players
Quick definition of the problem	Big time demands
Relatively simple integration	Unstable results
Complex algorithms and methods implemented	Needed big simplification
	Unprepared for production

Table 7.1: Summary of advantages and disadvantages of the general players

No algorithms knowledge needed

In this I see the biggest advantage of general game playing. The users without any Artificial Intelligence knowledge would solve the problems. Not many expertise are required, but at least knowledge of the first-order logic can be very beneficial. learning curve is very steep for GDL. If we begin with defining simple games such as Simple path finding from start position to goal position or the game Tower of Hanoi[1] we can learn GDL techniques very quick. I think that user with good mathematical knowledge, but without bigger knowledge of programming languages or even algorithmic knowledge may be able to solve some problems.

Quick definition of the problem

When we want to test some game/problem, such as PE game. The long way before first run is before us. Because we need to define environment, players and mediator to control the

[1] description of Towers of Hanoi: <http://en.wikipedia.org/wiki/Tower_of_Hano>

game flow. But if we use GGP a lot of work is done. At least we have already implemented players, the mediator, which will control the game (it checks legality of moves, it distributes information to all participants of the game, etc.). So for the first test we just need to develop the rules of the game. But before we jump into implementation of problems in GDL we have to stop and we should think out, if we are able to define the problem in GDL at all. The games or problems played by the general players need to be simplified. Usually we don't need all information about the environment to obtain desired results. But it can happen that amount of the informations needed for solution is too big to be handled by the general player. For example use the general player for controlling a car would probably be largely unsuccessful.

Relatively simple integration

Another thing that speaks in favor of the general player is chosen communication protocol. Because the player is receiving only three commands 2.2 to be controlled during the gameplay. Someone might say, that http is too slow for communication between environment and the agent. It is right argument, but if we take in account time consumed by the general player for playing is latency in http protocol meaning less. The other thing is that not every player is communicating only over the http protocol. For example Cadia player 5.2 is communicating through in/out pipes and the http server is only extension.

Complex algorithms and methods implemented

The thing, which can also speak in favor of the usage of the general players is lots of advanced algorithms and theirs improvements implemented in these players. For example a lot of research was made in reasoning or improving UCT 4.5 such as in [9, 15]. This can give us an advantage in using the general player over implementing this features alone.

Big time demands

Demands on time is the biggest disadvantage of the general players. In Section 6.5 we were discussing reason why I chose *startclock* and *playclock* to be 10 seconds. Not every domain is that complete that the general player ould need a lot of time. But common time for *startclock* in in tens of seconds. This is not that big problem, but for *playclock* 5 and more seconds per one move is usually selected. So for simulations the general players can be still useful, but for real application is this time often too long.

Unstable results

As we have observed in Table 6.6 the results if Cadia player were very unstable. In table it is represented as standard deviation (in bracket in column avg. steps). So the question occurs how much we can rely on the general players.

Unavailable public players

One of the reasons why it is difficult to explore the possibilities of general players is their unavailability. At least two web portals specialized on Genera Game Playing are existing. Probably most popular is The Dresden GGP Server (now running 130.208.241.192/ggpserver/). On this server we can find **228** registered players. Some of them are only modification or updates and some of them are just test of the user. But still we can find some familiar names of the player such as Cadia, Ary, Fluxplayer, TurboTurtle, etc.). Unfortunately none of them is usually running. The reason is clear. The players must run on its own server, which is administrated by each user (developer). Also binaries or even source code is not available in common. The only exception is Cadia player. We can find many basic player, such as BasicPlayer, Jocular, Common Prolog Player, etc. But these players are usually use less for application.

Chapter 8

Conclusions

This thesis was focused on application of General Game Playing systems in real problems. After becoming familiar with the concepts of General Game Playing and some previous works in this research field, I summarized results of annual competition held under summer conference of AAAI. Based on competition results I presented some successful players in this competition and their game solving techniques. I also described the most used techniques and algorithms used on GGP.

The important part of GGP is Game Description Language for games definition. To clarify how the problems can be implemented in GDL (Chapter 2) and GDL-II (Chapter 3) I made two simple games and precisely explained them.

In Chapter 6 I implemented domain specific players for Pursuit-evasion game. I created three different heuristic strategies (one for evader and two for pursuers) and tested them. The comparison of their results against the general player's results showed that the general player is capable to achieve similar values and to be even more successful on smaller domains. On the other hand the general player's results weren't so reliable because other test runs showed that unlike other players the general player tends to fail unexpectedly. We also tried to discover whether the general player can take advantage from controlling all pursuers and make them to cooperate. The results of this modification were surprising, because even on a small environment the general player received worse rewards than in the case when the players were controlled separately. In conclusion, the general player wasn't competent to solve all problems properly despite previous expectations.

I also commented how difficult it is to create game for GGP compare to complete domain implementation and what advantages can the general player bring. On of the possible ways to use the generic player is to investigate solvability of given problems and to obtain information how difficult it would be to create domain player.

In Chapter 7 I discussed main advantages and disadvantages of the general players. To summarize, we can say that in smaller tasks, general players are useable only as guidance for complete problem solving. The general players are still too limited and incompetent to be applied as agents systems. Despite their relatively good results during solving various problems and despite containing advanced algorithms and techniques, they can't be compared to agents developed primarily for concrete problems.

Evaluation

1. **The first task was to get familiarize with main competition organized under AAAI conference and study requirements needed to play General Games.** This task was done in Chapter 2 where the basis of GGP were described and in Chapter 3 the GDL and GDL-II are introduced. I also created two examples of the games implementation. In Chapter 5 some of the successful players are presented. In Chapter 4 I descried algorithms used commonly by the general players.

2. **The next task was to implement the domain player for specific problem, precisely define the problem and experimentally evaluate efficiency of general players against domain specific player.** I examined this task in Chapter 6, giving an explanation how the testing games were implemented and in what order those games run. I also described the behavior of each domain player. In the conclusion of the chapter I presented the results of the previously tested games and commented them.

3. **The last task was to discuss advantages and disadvantages of using general game players as AI libraries for solving problems.** I summarized and discussed advantages and disadvantages of general players in Chapter 7.

8.1 Future work

- Not many real life problems are with perfect information. People involved in the general game playing were aware about this limitation. Michael Thielscher published an extension of standard GDL the GDL-II 3.2. This is opening new opportunities for solving more problems using the general players. The problems will be also more similar to reality.

- Integration of the general player into existing environment, such as AgentScout2 developed by Agent Technology Center.

Bibliography

[1] Hilmar Finnsson and Yngvi Björnsson. Cadiaplayer: A simulation-based general game player. 2009.

[2] M. R. Genesereth and R. E. Fikes. Knowledge interchange format, version 3.0 reference manual. 1992.

[3] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI Magazine*, 26(2):62–72, 2005.

[4] Istvan Szita Guillaume Chaslot, Sander Bakkes and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. 2008.

[5] Tristan Cazenave Jean M ehat. Ary, a general game playing program.

[6] Ingo Keller. *Palamedes: A General Game Playing IDE.* PhD thesis, Dresden University of Technology, 2009.

[7] Levante Kocsis and Csaba Szepesvari. Bandit based monte-carlo planning. 2006.

[8] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical report, 2008. most recent version should be available at http://games.stanford.edu/.

[9] Jean Méhat and Tristan Cazenave. Combining uct and nested monte carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):271–277, 2010.

[10] Barney Darryl Pell. *Strategy Generation and Evaluation for Meta-Game Playing.* PhD thesis, University of Cambridge, 1993.

[11] Jacques Pitrat. Realization of a general game-playing program. In *IFIP Congress (2)*, pages 1570–1574, 1968.

[12] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 1191–1196. AAAI Press, 2007.

[13] Michael Thielscher. A general game description language for incomplete information games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 994–999. AAAI Press, 2010.

[14] Rogério Reis Vítor Santos Costa, Luís Damas and Rúben Azevedo. *YAP Prolog user's manual*.

[15] Karol Waledzik and Jacek Mandziuk. Multigame playing by means of uct enhanced with automatically generated evaluation functions. In *Artificial General Intelligence*, volume 6830, pages 327–332. Springer Berlin / Heidelberg, 2011.

Appendix A

List of used acronyms

GGP General Game Playing

PEG Pursuit-Evasion game

PE Pursuit-Evasion

GM Game Manager

AAAI Association for the Advancement of Artificial Intelligence

MCTS Monte Carlo Tree search

UCT Upper Confidence Bounds applied to Trees

Appendix B

GDL descriptions

B.0.1 Attacker and guard (GDL)

```
1  (role attacker)
2  (role guard)
3
4  ;; init state
5  (init (position attacker 3 1)) ;; third row, first column
6  (init (position guard 3 6)) ;; third row, last column
7  (init (control attacker))
8  (init (step 1))
9
10 ;; next state
11 (<= (next (step ?newN))
12   (true (step ?n))
13   (inc ?n ?newN))
14
15 (<= (next (control attacker))
16   (true (control guard)))
17
18 (<= (next (control guard))
19   (true (control attacker)))
20
21 (<= (next (position ?char ?nx ?ny))
22   (does ?char (move ?dir))
23   (true (position ?char ?x ?y))
24   (nextPosition ?dir ?x ?y ?nx ?ny))
25
26 ;; legal
27 (<= (legal attacker (move ?dir))
28   (true (control attacker))
29   (true (position attacker ?x ?y))
30   (legalMove ?dir ?x ?y)
31   (distinct ?dir nowhere))
32
33 (<= (legal attacker (move nowhere))
34   (not (true (control attacker))))
35
36 (<= (legal guard (move ?dir))
37   (true (control guard))
38   (true (position guard ?x ?y))
39   (legalMove ?dir ?x ?y)
40   (distinct ?dir nowhere))
41
42 (<= (legal guard (move nowhere))
43   (not (true (control guard))))
```

```
44
45  ;; terminal
46  (<= terminal
47    passed)
48
49  (<= terminal
50    timeout)
51
52  (<= terminal
53    captured)
54
55  ;; goal
56  (<= (goal attacker 100)
57    passed)
58
59  (<= (goal attacker 0)
60    timeout)
61
62  (<= (goal attacker 0)
63    (captured))
64
65  (<= (goal guard 0)
66    passed)
67
68  (<= (goal guard 100)
69    captured)
70
71  (<= (goal guard 100)
72    timeout)
73
74  ;; functions
75  (<= captured
76    (true (position attacker ?x ?y))
77    (true (position guard ?x ?y)))
78
79  (<= timeout
80    (true (step 37)))
81
82  (<= passed
83    (true (position attacker ?x 6))
84    (indexM ?x)
85    (not captured)
86    (not timeout))
87
88  (<= (legalMove north ?x ?y)
89    (inc ?y ?ny)
90    (indexM ?x)
91    (indexN ?ny))
92
93  (<= (legalMove east ?x ?y)
94    (inc ?x ?nx)
95    (indexM ?nx)
96    (indexN ?y))
97
98  (<= (legalMove south ?x ?y)
99    (dec ?y ?ny)
100   (indexM ?x)
101   (indexN ?ny))
102
103 (<= (legalMove west ?x ?y)
104   (dec ?x ?nx)
105   (indexM ?nx)
106   (indexN ?y))
107
108 (<= (nextPosition north ?x ?y ?x ?ny)
```

```
109     (inc ?y ?ny)
110     (indexM ?x))
111
112  (<= (nextPosition east ?x ?y ?nx ?y)
113     (inc ?x ?nx)
114     (indexN ?y))
115
116  (<= (nextPosition south ?x ?y ?x ?ny)
117     (dec ?y ?ny)
118     (indexM ?x))
119
120  (<= (nextPosition west ?x ?y ?nx ?y)
121     (dec ?x ?nx)
122     (indexN ?y))
123
124  (<= (nextPosition nowhere ?x ?y ?x ?y)
125     (indexM ?x)
126     (indexN ?y))
127
128  ;; consts
129  ;; m rows, n cols
130  (indexM 1)
131  (indexM 2)
132  (indexM 3)
133  (indexM 4)
134
135  (indexN 1)
136  (indexN 2)
137  (indexN 3)
138  (indexN 4)
139  (indexN 5)
140  (indexN 6)
141
142  (inc 1 2)
143  (inc 2 3)
144  (inc 3 4)
145  (inc 4 5)
146  (inc 5 6)
147  (inc 6 7)
148  (inc 7 8)
149  (inc 8 9)
150  (inc 9 10)
151  (inc 10 11)
152  (inc 11 12)
153  (inc 12 13)
154  (inc 13 14)
155  (inc 14 15)
156  (inc 15 16)
157  (inc 16 17)
158  (inc 17 18)
159  (inc 18 19)
160  (inc 19 20)
161  (inc 20 21)
162  (inc 21 22)
163  (inc 22 23)
164  (inc 23 24)
165  (inc 24 25)
166  (inc 25 26)
167  (inc 26 27)
168  (inc 27 28)
169  (inc 28 29)
170  (inc 29 30)
171  (inc 30 31)
172  (inc 31 32)
173  (inc 32 33)
```

```
174 (inc 33 34)
175 (inc 34 35)
176 (inc 35 36)
177 (inc 36 37)
178
179 (dec 38 37)
180 (dec 37 36)
181 (dec 36 35)
182 (dec 35 34)
183 (dec 34 33)
184 (dec 33 32)
185 (dec 32 31)
186 (dec 31 30)
187 (dec 30 29)
188 (dec 29 28)
189 (dec 28 27)
190 (dec 27 26)
191 (dec 26 25)
192 (dec 25 24)
193 (dec 24 23)
194 (dec 23 22)
195 (dec 22 21)
196 (dec 21 20)
197 (dec 20 19)
198 (dec 19 18)
199 (dec 18 17)
200 (dec 17 16)
201 (dec 16 15)
202 (dec 15 14)
203 (dec 14 13)
204 (dec 13 12)
205 (dec 12 11)
206 (dec 11 10)
207 (dec 10 9)
208 (dec 9 8)
209 (dec 8 7)
210 (dec 7 6)
211 (dec 6 5)
212 (dec 5 4)
213 (dec 4 3)
214 (dec 3 2)
215 (dec 2 1)
```

B.0.2 Attacker and guard (GDL-II)

```
1  (role attacker)
2  (role guard)
3  (role random)
4
5  ;; init state
6  (init (position attacker 3 1)) ;; third row, first column
7  (init (position guard 3 6)) ;; third row, last column
8  (init (control attacker))
9  (init (step 1))
10
11 ;; next state
12 (<= (next (step ?newN))
13   (true (step ?n))
14   (inc ?n ?newN))
15
16 (<= (next (control attacker))
17   (true (control guard)))
18
19 (<= (next (control guard))
20   (true (control attacker)))
21
22 (<= (next (position ?char ?nx ?ny))
23   (does ?char (move ?dir))
24   (true (position ?char ?x ?y))
25   (nextPosition ?dir ?x ?y ?nx ?ny))
26
27 (<= (next (blocked ?x ?y))
28   (does random (chooseBlocked ?x ?y)))
29
30 ;; legal
31 (<= (legal attacker (move ?dir))
32   (true (control attacker))
33   (true (position attacker ?x ?y))
34   (legalMove ?dir ?x ?y)
35   (distinct ?dir nowhere))
36
37 (<= (legal attacker (move nowhere))
38   (not (true (control attacker))))
39
40 (<= (legal guard (move ?dir))
41   (true (control guard))
42   (true (position guard ?x ?y))
43   (legalMove ?dir ?x ?y)
44   (distinct ?dir nowhere))
45
46 (<= (legal guard (move nowhere))
47   (not (true (control guard))))
48
49 (<= (legal random (chooseBlocked ?x ?y))
50   (indexM ?x)
51   (indexN ?y))
52
53 ;; sees
54 (<= (sees attacker (guardPosition ?x2 ?y2))
55   (true (position attacker ?x1 ?x1))
56   (true (position guard ?x2 ?y2))
57   (or (inRow ?x1 ?y1 ?x2 ?y2)
58     (inCol ?x1 ?y1 ?x2 ?y2)))
59
60 (<= (sees guard (attackerPosition ?x1 ?y1))
61   (true (position attacker ?x1 ?x1))
62   (true (position guard ?x2 ?y2))
63   (or (inRow ?x1 ?y1 ?x2 ?y2)
```

```
64        (inCol ?x1 ?y1 ?x2 ?y2)))
65
66  ;; terminal
67  (<= terminal
68    passed)
69
70  (<= terminal
71    timeout)
72
73  (<= terminal
74    captured)
75
76  ;; goal
77  (goal random 100)
78
79  (<= (goal attacker 100)
80    passed)
81
82  (<= (goal attacker 0)
83    timeout)
84
85  (<= (goal attacker 0)
86    (captured))
87
88  (<= (goal guard 0)
89    passed)
90
91  (<= (goal guard 100)
92    captured)
93
94  (<= (goal guard 100)
95    timeout)
96
97  ;; functions
98  (<= captured
99    (true (position attacker ?x ?y))
100   (true (position guard ?x ?y)))
101
102 (<= timeout
103   (true (step 37)))
104
105 (<= passed
106   (true (position attacker ?x 6))
107   (indexM ?x)
108   (not captured)
109   (not timeout))
110
111 (<= (notBlocked ?x ?y)
112   (indexM ?x)
113   (indexN ?y)
114   (not (true (blocked ?x ?y))))
115
116 (<= (inRow ?x ?y1 ?x ?y2)
117   (indexM ?x)
118   (indexN ?y1)
119   (indexN ?y2))
120
121 (<= (inCol ?x1 ?y ?x2 ?y)
122   (indexM ?x1)
123   (indexM ?x2)
124   (indexN ?y))
125
126 (<= (legalMove north ?x ?y)
127   (inc ?y ?ny)
128   (indexM ?x)
```

```
129     (indexN ?ny)
130     (notBlocked ?x ?ny))
131
132  (<= (legalMove east ?x ?y)
133     (inc ?x ?nx)
134     (indexM ?nx)
135     (indexN ?y)
136     (notBlocked ?nx ?y))
137
138  (<= (legalMove south ?x ?y)
139     (dec ?y ?ny)
140     (indexM ?x)
141     (indexN ?ny)
142     (notBlocked ?x ?ny))
143
144  (<= (legalMove west ?x ?y)
145     (dec ?x ?nx)
146     (indexM ?nx)
147     (indexN ?y)
148     (notBlocked ?nx ?y))
149
150  (<= (nextPosition north ?x ?y ?x ?ny)
151     (inc ?y ?ny)
152     (indexM ?x))
153
154  (<= (nextPosition east ?x ?y ?nx ?y)
155     (inc ?x ?nx)
156     (indexN ?y))
157
158  (<= (nextPosition south ?x ?y ?x ?ny)
159     (dec ?y ?ny)
160     (indexM ?x))
161
162  (<= (nextPosition west ?x ?y ?nx ?y)
163     (dec ?x ?nx)
164     (indexN ?y))
165
166  (<= (nextPosition nowhere ?x ?y ?x ?y)
167     (indexM ?x)
168     (indexN ?y))
169
170  ;; consts
171  ;; m rows, n cols
172  (indexM 1)
173  (indexM 2)
174  (indexM 3)
175  (indexM 4)
176
177  (indexN 1)
178  (indexN 2)
179  (indexN 3)
180  (indexN 4)
181  (indexN 5)
182  (indexN 6)
183
184  (inc 1 2)
185  (inc 2 3)
186  (inc 3 4)
187  (inc 4 5)
188  (inc 5 6)
189  (inc 6 7)
190  (inc 7 8)
191  (inc 8 9)
192  (inc 9 10)
193  (inc 10 11)
```

```
194 (inc 11 12)
195 (inc 12 13)
196 (inc 13 14)
197 (inc 14 15)
198 (inc 15 16)
199 (inc 16 17)
200 (inc 17 18)
201 (inc 18 19)
202 (inc 19 20)
203 (inc 20 21)
204 (inc 21 22)
205 (inc 22 23)
206 (inc 23 24)
207 (inc 24 25)
208 (inc 25 26)
209 (inc 26 27)
210 (inc 27 28)
211 (inc 28 29)
212 (inc 29 30)
213 (inc 30 31)
214 (inc 31 32)
215 (inc 32 33)
216 (inc 33 34)
217 (inc 34 35)
218 (inc 35 36)
219 (inc 36 37)
220
221 (dec 38 37)
222 (dec 37 36)
223 (dec 36 35)
224 (dec 35 34)
225 (dec 34 33)
226 (dec 33 32)
227 (dec 32 31)
228 (dec 31 30)
229 (dec 30 29)
230 (dec 29 28)
231 (dec 28 27)
232 (dec 27 26)
233 (dec 26 25)
234 (dec 25 24)
235 (dec 24 23)
236 (dec 23 22)
237 (dec 22 21)
238 (dec 21 20)
239 (dec 20 19)
240 (dec 19 18)
241 (dec 18 17)
242 (dec 17 16)
243 (dec 16 15)
244 (dec 15 14)
245 (dec 14 13)
246 (dec 13 12)
247 (dec 12 11)
248 (dec 11 10)
249 (dec 10 9)
250 (dec 9 8)
251 (dec 8 7)
252 (dec 7 6)
253 (dec 6 5)
254 (dec 5 4)
255 (dec 4 3)
256 (dec 3 2)
257 (dec 2 1)
```

B.0.3 Basic PE game

```
1  ;; Pursuit evasion game with 2 pursuers and one evader
2  ;; Game ends after 100 states
3
4  (role evader1)
5  (role pursuer1)
6  (role pursuer2)
7
8  ;; init state
9  (init (position evader1 0))
10 (init (position pursuer1 10))
11 (init (position pursuer2 110))
12
13 (init (escapePoint 60))
14
15 (init (state 1))
16
17 ;; next state
18 (<= (next (position ?char ?V))
19   (does ?char (move ?char ?V)))
20
21 (<= (next (escapePoint ?V))
22   (true (escapePoint ?V)))
23
24 (<= (next (state ?NewN))
25   (true (state ?N))
26   (inc ?N ?NewN))
27
28 ;; legal
29 (<= (legal evader1 (move evader1 ?NewV))
30   (true (position evader1 ?V))
31   (adjacentVertex ?V ?NewV))
32
33 (<= (legal evader1 (move evader1 ?V))
34   (true (position evader1 ?V)))
35
36 (<= (legal ?purs (move ?purs ?NewV))
37   (pursuersTeam ?purs)
38   (true (position ?purs ?V))
39   (adjacentVertex ?V ?NewV)
40   (not (true (escapePoint ?NewV))))
41
42 (<= (legal ?purs (move ?purs ?V))
43   (pursuersTeam ?purs)
44   (true (position ?purs ?V))
45   (not (true (escapePoint ?V))))
46
47 ;; terminal
48 (<= terminal
49   (captured))
50
51 (<= terminal
52   (timeout))
53
54 (<= terminal
55   (escaped))
56
57 ;; goal
58 ;; evad1
59 (<= (goal evader1 100)
60   (escaped)
61   (not (timeout)))
62
63 (<= (goal evader1 0)
```

```
64      (captured)
65      (not (timeout)))
66
67  (<= (goal evader1 50)
68      (timeout))
69  ;; purs1
70  (<= (goal pursuer1 100)
71      (true (position pursuer1 ?V))
72      (true (position evader1 ?V))
73      (not (timeout)))
74
75  (<= (goal pursuer1 0)
76      (true (position pursuer1 ?V1))
77      (true (position evader1 ?V2))
78      (distinct ?V1 ?V2)
79      (not (timeout)))
80
81  (<= (goal pursuer1 25)
82      (timeout))
83  ;;purs2
84  (<= (goal pursuer2 100)
85      (true (position pursuer2 ?V))
86      (true (position evader1 ?V))
87      (not (timeout)))
88
89  (<= (goal pursuer2 0)
90      (true (position pursuer2 ?V1))
91      (true (position evader1 ?V2))
92      (distinct ?V1 ?V2)
93      (not (timeout)))
94
95  (<= (goal pursuer2 25)
96      (timeout))
97
98  ;; functions
99  (<= (adjacentVertex ?V1 ?V2)
100     (vertex ?V1)
101     (vertex ?V2)
102     (or (edge ?V1 ?V2 ?Cost1)
103       (edge ?V2 ?V1 ?Cost2)))
104
105 (<= (escaped)
106     (true (position evader1 ?V))
107     (true (escapePoint ?V)))
108
109 (<= (captured)
110     (true (position evader1 ?V))
111     (or (true (position pursuer1 ?V))
112       (true (position pursuer2 ?V))))
113
114 (<= (timeout)
115     (true (state 100)))
116
117 ;; teams
118 (evadersTeam evader1)
119 (pursuersTeam pursuer1)
120 (pursuersTeam pursuer2)
121
122 ;; graph representation
123 (vertex 1)
124 (vertex 2)
125 ...
126 (veretx N)
127
128 (edge 1 2 1)
```

```
129 (edge 3 4 1)
130 ...
131 (edge M N 1)
132
133 ;; other consts
134 (inc 0 1)
135 (inc 1 2)
136 ...
137 (inc 99 100)
```

Appendix C

Installation and user guide

C.1 Installation

On included CD we find folder called *eclipse projects* with three projects (*bc-domain-player*, *org.eclipse.palamedes.gdl.core*, *org.eclipse.palamedes.kif.core*), all these project need to be imported to the Eclipse's workbench. The guide how to import projects is in Eclipse documentation - Importing existing projects.

C.2 Usage

In package *cz.cvut.fel.ggp* we find 5 classes which starts player with the given strategy. Players without substring *2player* are intended for games which does not contain this substring too.

Class name	**Strategy**	Port
StartEvader1	Evader's main strategy	4001
StartEvader2Player	Evader's main strategy	4001
StartPursuer1	Pursuer's follow strategy	4002
StartPursuer2	Pursuer's follow strategy	4003
StartPursuer3	Pursuer's block strategy	4004

Table C.1: Player's main classes

For running matches we need run game manager. The game manager is included in CD and it can be find in eclispe_projectsbc-domain-playergamecontroller. It is java archive and it can be executed by command:

```
java -jar gamecontroller/gamecontroller-gui-r495.jar
```

If we want to start the game (e.g. *game_peg_mat11.kif*), we must make sure that the **match id** is identical with the name of file containing game description (it means: **game_peg_mat11**). The example how may look the game manager before the game is started is in Figure C.1.

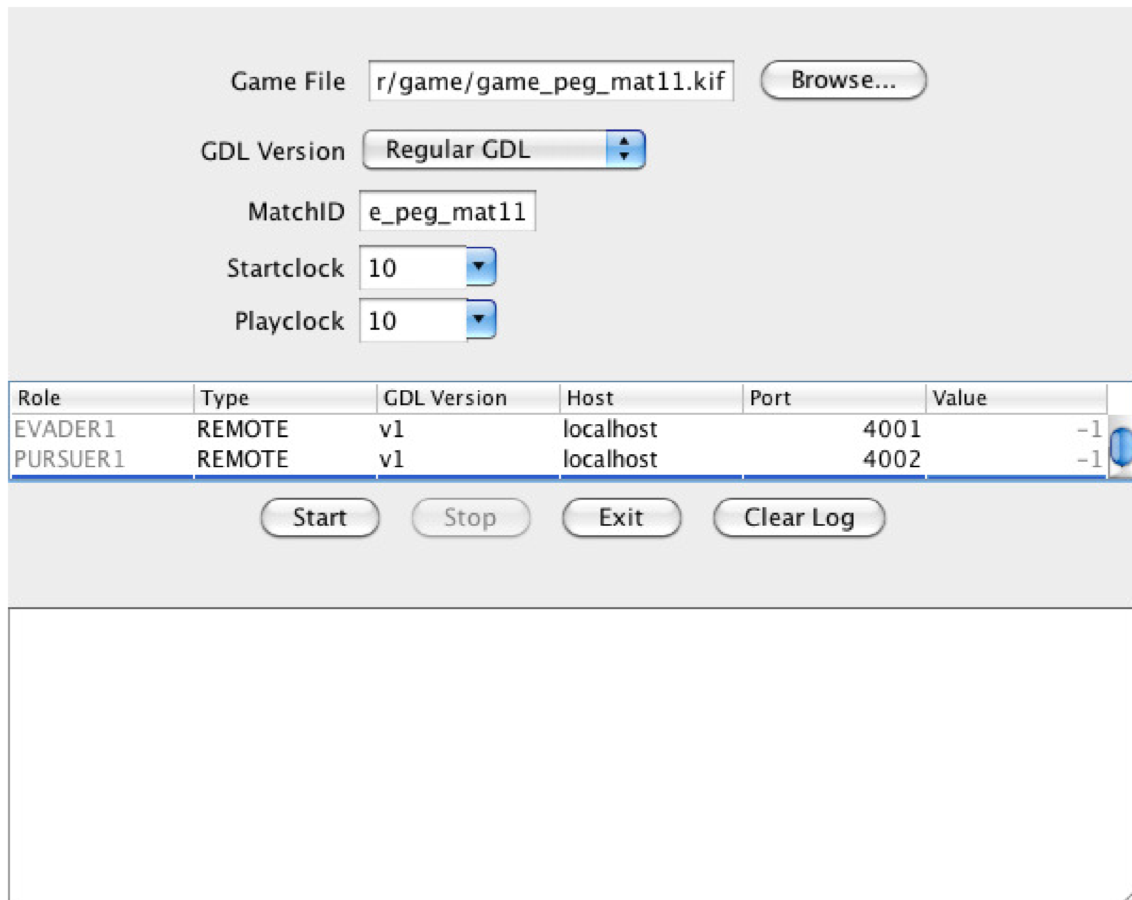


Figure C.1: Game Controller's window

Appendix D

Content of included CD

```
eclipse_projects/bc-domain-player/ -- Eclipse project with domain players
├── bin
├── game -- tested games
|   ├── examples
|   ├── test
|   └── tool
├── gamecontroller
├── gamegraph -- Environment representation for domain player
|   └── tool
└── src
    ├── cz
    |   └── cvut
    |       └── fel
    |           └── ggp -- Main package of the project. Contains classes
starting domain players
    |               ├── algorithms
    |               ├── environment -- Here is Game.java class, which holds
and provides all information about the game
    |               ├── player -- Here are players modified for playing PE
game (PEGPlayer, PEGPlayer2)
    |               └── strategy -- Contains strategies for players (evader's
main strategy, pursuer's follow and block strategies)
    └── org
        └── binwall
            └── graph -- Class for holding graph representation
game_logs/ -- Logs of played games
├── peg_common_goal
├── peg_game_2players
├── peg_separate_goal
└── peg_separate_goal_shorttime
text/ -- Text of this work
```