General Game Playing in Common Lisp

Steve Losh
Reykjavík University
Menntavegi 1
Reykjavík, Iceland
steve@stevelosh.com

ABSTRACT

Common Lisp has a rich history in the field of artificial intelligence. One subfield of AI that is currently an area of active research is general game playing, which focuses on writing players that can learn to play *any* game intelligently given only its rules. We provide a short introduction to the field of general game playing and present `cl-ggp`, a framework for writing general game players in Common Lisp. We then implement a simple general game player in about forty lines of code to show the framework in action.

CCS CONCEPTS

•**Computing methodologies →Game tree search;** *Logic programming and answer set programming;* •**Software and its engineering →Application specific development environments;**

KEYWORDS

General game playing, Common Lisp

1 OVERVIEW OF GENERAL GAME PLAYING

Traditional game AI research has focused on creating agents for individual games, such as Deep Blue[2] for Chess and AlphaGo[7] for Go. The field of general game playing has a higher-level goal: creating agents capable of playing *any* game intelligently given only its rules. Instead of programmers using domain knowledge to create a new AI for each game from scratch, a general game player must be able to receive a set of rules and be ready to play strategically almost immediately[1].

While playing any game is a laudable goal, in practice there are some restrictions imposed on the games for practicality. Most existing research in the field of general game playing deals with finite, simultaneous-move, complete-information games, and the International General Game Playing Competition focuses on these types of games[3]. Relaxing these restrictions (e.g. playing games with incomplete information like Texas hold'em) is an area of active research.

[1]Typically players are given anywhere from thirty seconds to several minutes of preparation time after receiving the rules for any preprocessing they might want to perform.

10th ELS April 3–4, Brussels, Belgium

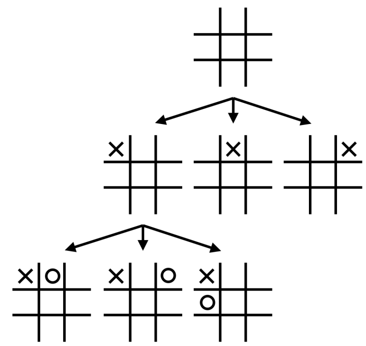

Figure 1: A partial game tree for tic-tac-toe.

Games of this type can be thought of as a tree of states[2], the leaves of which are terminal states where the game has ended. Figure 1 shows an example of a partial game tree for the game of tic-tac-toe.

2 OVERVIEW OF GAME DESCRIPTION LANGUAGE

Pitting general game players against each other in competitions requires a standardized way to describe game rules. A simple logic programming language called Game Description Language (GDL) was created for this purpose[3]. GDL is a variant of Datalog with several extensions and an s-expression-based syntax that Lisp users will appreciate. We present a basic overview here, but readers who want more details should consult the full specification[4].

A game state in GDL is described by a series of logical facts of the form `(true . . . )`. Listing 1 shows the state corresponding to the lower-left node in the example game tree.

The definition of a game in GDL consists of logical facts and rules. The roles and initial state of the game are given as facts as shown in Listing 2.

Other aspects of a game are described as logical rules of the form `(<= <head> <body. . . >)` as shown in Listing 3. Logic variables are named with a `?` prefix.

In lines 1-11 we define `row`, `line`, and `open` predicates which will be useful in defining the rest of the game rules. The definitions for `column` and `diagonal` have been omitted to save space.

[2]It may be possible to reach an identical game state from multiple paths, so the "game tree" would more properly be called the "game directed acyclic graph".
[3]GDL was intended to be a heavily restricted language to make it easier to reason about, but unfortunately turned out to be Turing complete[5]. In practice this is usually not a problem.

Listing 1: An example state for a game of tic-tac-toe.

```
(true (control x))
(true (cell 1 1 x))
(true (cell 1 2 o))
(true (cell 1 3 blank))
(true (cell 2 1 blank))
(true (cell 2 2 blank))
(true (cell 2 3 blank))
(true (cell 3 1 blank))
(true (cell 3 2 blank))
(true (cell 3 3 blank))
```

Listing 2: Role and initial state definition for tic-tac-toe.

```
(role x)
(role o)
(init (control x))
(init (cell 1 1 blank))
(init (cell 1 2 blank))
(init (cell 1 3 blank))
(init (cell 2 1 blank))
(init (cell 2 2 blank))
(init (cell 2 3 blank))
(init (cell 3 1 blank))
(init (cell 3 2 blank))
(init (cell 3 3 blank))
```

The `terminal` predicate (lines 13-17) determines whether a particular state is terminal. A game of tic-tac-toe terminates when one player has completed a line or there are no open spaces left on the board.

For all terminal states the `(goal <player> <value>)` predicate (lines 19-31) must describe a single goal value for every role. We give a player a score of 100 if they win by marking a line, 0 if their opponent wins, or 50 for a draw.

The `(legal <player> <move>)` predicate (lines 33-41) gives the legal moves for each player at any given state. All players must have at least one move in every non-terminal state. In our definition of tic-tac-toe there are two types of moves: when a player has control they must mark a blank cell, and when the other player has control they can only perform a "no-op" move.

When all players have made a move they are added to the state of the game as `(does <player> <move>)` facts. The successor state can then be computed from the `(next <fact>)` predicate (lines 43-60).

Tic-tac-toe has two kinds of information that must be tracked for each state: the contents of the board and which player has control. A new state is computed from the current state in four parts:

- Control changes hands every turn (lines 45-46).
- Cells that are already marked stay marked (lines 48-50).
- If a player uses their move to mark a cell, that cell is marked in the next state (lines 52-54).
- All other blank cells remain blank (lines 56-60).

There are several other restrictions on game definitions that we will not cover here. Interested readers should consult the full GDL specification[4] to learn all the details of playing games written in GDL.

Listing 3: Rules for tic-tac-toe.

```
1  (<= (row ?n ?mark)
2    (true (cell ?n 1 ?mark))
3    (true (cell ?n 2 ?mark))
4    (true (cell ?n 3 ?mark)))
5
6  (<= (line ?mark) (row ?n ?mark))
7  (<= (line ?mark) (column ?n ?mark))
8  (<= (line ?mark) (diagonal ?n ?mark))
9
10 (<= open
11   (true (cell ?r ?c blank)))
12
13 ;;;; Terminal
14
15 (<= terminal (line x))
16 (<= terminal (line o))
17 (<= terminal (not open))
18
19 ;;;; Goal Values
20
21 (<= (goal ?player 100)
22   (line ?player))
23
24 (<= (goal ?player 0)
25   (line ?other)
26   (distinct ?player ?other))
27
28 (<= (goal ?player 50)
29   (not (line x))
30   (not (line o))
31   (not open))
32
33 ;;;; Legal Moves
34
35 (<= (legal ?player (mark ?row ?col))
36   (true (cell ?row ?col blank))
37   (true (control ?player)))
38
39 (<= (legal ?player noop)
40   (true (control ?other))
41   (distinct ?player ?other))
42
43 ;;;; State Transitions
44
45 (<= (next (control x)) (true (control o)))
46 (<= (next (control o)) (true (control x)))
47
48 (<= (next (cell ?row ?col ?player))
49   (true (cell ?row ?col ?player))
50   (distinct ?player blank))
51
52 (<= (next (cell ?row ?col ?player))
53   (true (cell ?row ?col blank))
54   (does ?player (mark ?row ?col)))
55
56 (<= (next (cell ?row ?col blank))
57   (true (cell ?row ?col blank))
58   (does ?player (mark ?x ?y))
59   (or (distinct ?row ?x)
60       (distinct ?col ?y)))
```

3 WRITING GENERAL GAME PLAYERS IN COMMON LISP

To write a general game player capable of competing with other players we can split the implementation into three distinct parts:

1. The HTTP-based GGP network protocol, for connecting to and communicating with a central game server.
2. Parsing GDL game descriptions and reasoning about states to determine legal moves, terminality, goal values, etc.
3. An AI to search the game tree and find moves that will lead to a win for the player.

cl-ggp is a library written in Common Lisp to handle the tedious parts of this process. It is installable with Quicklisp[4]. The code is available as a Mercurial[5] or Git[6] repository[7], and is released under the MIT license. We present a short guide to its usage here, but for a much more thorough introduction readers should refer to the documentation[8].

The library contains two separate ASDF systems: cl-ggp and cl-ggp.reasoner.

3.1 The cl-ggp System

The main cl-ggp system handles the GGP network protocol and manages the basic flow of games. It takes a simple object-oriented approach similar to that of the ggp-base package[9] for the JVM.

To create a general game player users define a CLOS subclass of the ggp-player class and implement four methods to handle the main flow of the game:

(player-start-game <player> <rules> <role> <deadline>) is called by the framework when a new game begins. Each player will only ever be running a single game at a time. The method receives as arguments the GDL description of the game (a list of s-expressions), the role it has been assigned, and the time limit for any initial processing it may wish to do[10].

(player-update-game <player> <moves>) is called by the framework at the beginning of each turn. It receives as an argument the list of moves done by players in the previous turn (except for the first turn, in which moves will be nil). Players will typically use this method to compute the new state of the game.

(player-select-move <player> <deadline>) is called by the framework directly after player-update-game, and must return a move to perform before the given deadline.

(player-stop-game <player>) is called by the framework when a game has ended. Players can use it to trigger any cleanup they might require.

Once all the necessary methods have been defined, an instance can be created with (make-instance <class> :name <name> :port <port number>) and given to start-player to begin listening on the given port. stop-player can be used to stop a player and relinquish the port.

3.2 The cl-ggp.reasoner System

The cl-ggp.reasoner system implements a basic Prolog-based reasoning system to use as a starting point if desired. Under the hood it uses the Temperance[11] logic programming library to compute and reason about states. Temperance is an implementation of the Warren Abstract Machine[8][12] in pure Common Lisp.

The included reasoner is intended to be a simple starting point for experimentation. Users who want better performance or want more access to the reasoning process are encouraged to write their own reasoning systems.

The reasoner API consists of six functions:

(make-reasoner <rules>) Creates and returns a reasoner object for reasoning about the given GDL rules.

(initial-state <reasoner>) Returns the initial state of the game as described by the (init . . .) facts in the GDL.

(next-state <reasoner> <state> <moves>) Returns the successor state of the given state, assuming the given moves were taken.

(terminalp <reasoner> <state>) Returns t if the given state is terminal, nil otherwise.

(legal-moves-for <reasoner> <state> <role>) Returns a list of all legal moves for the given role in the given state.

(goal-value-for <reasoner> <state> <role>) Returns the goal value for the given role in the given state.

4 IMPLEMENTING A RANDOM PLAYER

The basic framework and included reasoner are enough to write a simple general game player that can play any GDL game legally (though not particularly intelligently). Such a player is shown in Listing 4.

We first define a subclass of ggp-player called random-player with slots for holding the information it will need to play a game.

In player-start-game we create a reasoner with the rules passed along by the framework. We store the reasoner and the assigned role in the player instance.

In player-update-game we compute the current state of the game. If moves is nil this is the first turn in the game, and so we simply request the initial state from the reasoner. Otherwise we compute the next state from the current one and the moves performed. We store the current state in the player for later use.

In player-select-move we compute the legal moves our role can perform in the current state and choose one at random. This ensures we always play legally, but is not usually a very effective strategy. A more intelligent player would use the given time to search the game tree and try to find which moves lead to high goal values for its role.

Finally in player-stop-game we clear out the slots of the player so the contents can be garbage collected.

[4] As of the time of this writing it is not in a Quicklisp dist, so you'll need to use Quicklisp's local project support.
[5] https://bitbucket.org/sjl/cl-ggp/
[6] https://github.com/sjl/cl-ggp/
[7] The most recent commit hashes at the time of this writing are abdfc9d (Mercurial) and 749651e (Git).
[8] https://sjl.bitbucket.io/cl-ggp/
[9] https://github.com/ggp-org/ggp-base/
[10] Symbols in rules are interned in the ggp-rules package to avoid polluting other namespaces.
[11] https://bitbucket.org/sjl/temperance/
[12] A virtual machine designed for compiling and running Prolog code.

Listing 4: A simple general game player capable of playing any GDL game legally.

```
(defclass random-player (ggp-player)
  ((role          :accessor p-role)
   (current-state :accessor p-current-state)
   (reasoner      :accessor p-reasoner)))

(defmethod player-start-game
    ((player random-player) rules role deadline)
  (setf (p-role player) role
        (p-reasoner player) (make-reasoner rules)))

(defmethod player-update-game
    ((player random-player) moves)
  (setf (p-current-state player)
        (if (null moves)
          (initial-state (p-reasoner player))
          (next-state (p-reasoner player)
                      (p-current-state player)
                      moves))))

(defmethod player-select-move
    ((player random-player) deadline)
  (let ((moves (legal-moves-for
                 (p-reasoner player)
                 (p-current-state player)
                 (p-role player))))
    (nth (random (length moves)) moves)))

(defmethod player-stop-game
    ((player random-player))
  (setf (p-current-state player) nil
        (p-reasoner player) nil
        (p-role player) nil))

(defvar *random-player*
  (make-instance 'random-player
                 :name "RandomPlayer"
                 :port 4000))

(start-player *random-player*)
```

Once the four required methods are defined we create an instance of the player and start it listening on the given port.

5 IMPROVING THE PLAYER

Classical game tree search strategies like minimax can be used to search the game tree for promising moves, and for small games they produce acceptable results. However, because a general game player must be able to play *any* game it cannot have a heuristic function hard-coded into it[13], which makes many traditional search techniques much more difficult.

One strategy that has recently gained popularity is simulation-based search, such as Monte-Carlo Tree Search[1]. MCTS works by running many random playouts from a state to determine its approximate value without expanding the entire game tree, operating under the assumption that a state where random playouts tend to produce good results is a good state to be in.

MCTS relies on running many random playouts to provide data, so the faster playouts can be run the better its results will be. In practice the bottleneck in running random simulations is reasoning about and computing states, so another improvement would be to optimize the reasoner or write an entirely new one, possibly not using Prolog-style reasoning at all (e.g. a propositional network[6]).

6 CONCLUSION

General game playing involves the creation of AI agents capable of playing any game intelligently given only its rules. After giving an overview of the field we presented `cl-ggp`, a framework for writing general game players in Common Lisp, and showed how to create a simple player with it. We hope this framework will reduce the friction involved in creating players and encourage more people to experiment with Common Lisp as a platform for research in general game playing.

ACKNOWLEDGMENTS

We would like to thank Stephan Schiffel for providing feedback on a draft of this paper.

REFERENCES

[1] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. Comput. Intellig. and AI in Games ()* 4, 1 (2012), 1–43.

[2] Murray Campbell, A Joseph Hoane Jr., and Feng-hsiung Hsu. 2002. Deep Blue. *Artificial Intelligence* 134, 1-2 (Jan. 2002), 57–83.

[3] Michael R Genesereth and Yngvi Björnsson. 2013. The International General Game Playing Competition. *AI Magazine* (2013).

[4] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. 2008. *General Game Playing: Game Description Language Specification.* Technical Report.

[5] Abdallah Saffidine. 2014. The Game Description Language Is Turing Complete. *IEEE Transactions on Computational Intelligence and AI in Games* 6, 4 (2014), 320–324.

[6] Eric Schkufza, Nathaniel Love, and Michael R Genesereth. 2008. Propositional Automata and Cell Automata - Representational Frameworks for Discrete Dynamic Systems. *Australasian Conference on Artificial Intelligence* (2008).

[7] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.

[8] David H D Warren. 1983. An Abstract Prolog Instruction Set. (Oct. 1983), 1–34.

[13] Any heuristic that works for one game could potentially backfire in a different game.