General Game Playing - Killer App for Logic Programming

Michael Genesereth

Stanford University, Stanford, CA, USA

Abstract

A key challenge in teaching Logic Programming (LP) is providing students with good examples. In our experience, we have found that General Game Playing (GGP) is a popular application area that is especially well-suited to LP. In this article, we review the basic components of GGP - game description, game playing, and metagaming - and, for each, we show how LP plays a role and how that component helps in teaching LP.

1. Introduction

Playing strategy games like chess and checkers couples intellectual activity with competition. By playing games, we can exercise and improve our intellectual skills. The competition adds excitement and allows us to compare our skills to those of others. The same motivation accounts for interest in Computer Game Playing as a testbed for Artificial Intelligence. Programs that think better should be able to win more games, and so we can use competitions as an evaluation technique for intelligent systems.

Unfortunately, building programs to play specific games has limited value in AI. (1) To begin with, specialized game players are very narrow. They can be good at one game but not another. Deep Blue may have beaten the world Chess champion, but it has no clue how to play checkers. (2) A second and more fundamental problem with specialized game playing systems is that they do only part of the work. Most of the interesting analysis and design is done in advance by their programmers. The systems themselves might as well be tele-operated.

All is not lost. The idea of game playing can be used to good effect to inspire and evaluate good work in Artificial Intelligence, but it requires moving more of the design work to the computer itself. This can be done by focussing attention on General Game Playing [6][7][8].

General game players are systems able to accept descriptions of arbitrary games at runtime and able to use such descriptions to play those games effectively without human intervention. In other words, they do not know the rules until the games start. Once the game begins, they receive a game description; and, based solely on this description, they must figure out how to play the game legally and effectively.

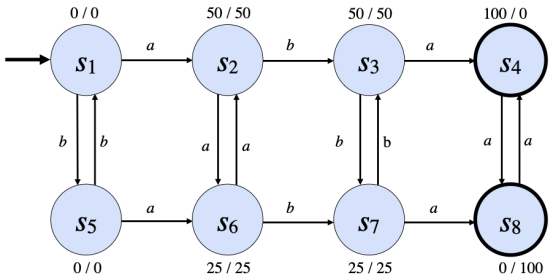
Unlike specialized game players, such as Deep Blue, general game players cannot rely on algorithms designed in advance for specific games. General game playing expertise must depend on intelligence on the part of the game player and not just intelligence of the programmer of the game player. In order to perform well, general game players must incorporate results from various disciplines, such as knowledge representation, reasoning, and rational decision making; and these capabilities have to work together in a synergistic fashion.

In what follows, we review the key components of GGP - game description, game playing, and metagaming - and, for each, we show how LP plays a role and how that component helps in teaching LP.

2. Game Description

While GGP encompasses a wide variety of games, all games in GGP share a common abstract structure. In particular, every game can be modeled as a state graph like the one below.

PEG 2024: 2nd Workshop on Prolog Education, October, 2024, Dallas, USA.



Each game takes place in an environment with finitely many states, with a distinguished initial state and one or more terminal states. In addition, each game has a fixed, finite number of players; each player has finitely many possible actions in any game state, and each state has an associated goal value for each player. The dynamic model is discrete update: on each step, there is one player in control, that player performs an action, and the environment updates only in response to the action taken by the player in control.

Unfortunately, it is impractical to use such explicit representations to communicate game rules to game players. Even though the numbers of states and actions are finite, these sets can be extremely large; and the corresponding graphs can be larger still. For example, in chess, there are thousands of possible moves and more than 10^{30} states.

Fortunately, we can do better. Using Logic Programming, we can exploit the regularities in the game to produce more compact compact encodings of game rules as logic programs.

Consider the case of Tic Tac Toe (Noughts and Crosses). We use datasets to describe states of the game. Here we use a ternary relation named cell (that captures data about which cells contain which marks and which cells are empty) together with a unary relation control (that indicates which player is in control, i.e. next to play).

```
cell(1,1,b)
cell(1,2,b)
cell(1,3,b)
cell(2,1,b)
cell(2,2,b)
cell(2,3,b)
cell(3,1,b)
cell(3,2,b)
cell(3,3,b)
control(x)
```

We use Prolog-style [13] view definitions to define various properties of states. The row relation holds of a row number and a player if and only if all three cells in the row have that player's mark. The column and diagonal relations are defined analogously. A player has a line of marks if and only if the player has a row or a column or a diagonal. A game is terminal if and only if one of the players has a line of marks or if there are no remaining empty cells. The game rules also specify constraints and goals, i.e. the actions that are legal in each state and a characterization of the states satisfy the goal of each player.

```
row(M,Z) :- cell(M,1,Z) & cell(M,2,Z) & cell(M,3,Z)
column(N,Z) :- cell(1,N,Z) & cell(2,N,Z) & cell(3,N,Z)
diagonal(Z) :- cell(1,1,Z) & cell(2,2,Z) & cell(3,3,Z)
diagonal(Z) :- cell(1,3,Z) & cell(2,2,Z) & cell(3,1,Z)
```

```
line(Z) :- row(M,Z)
line(Z) :- column(M,Z)
line(Z) :- diagonal(Z)

terminal :- line(x)
terminal :- line(o)
terminal :- ~open
open :- cell(M,N,b)

legal(mark(M,N)) :- cell(M,N,b)

goal(x) :- line(x)
goal(o) :- line(o)
```

Given a state and an action, the next state is determined uniquely by the corresponding Epilog-style [9] operation rules. If a player in control marks a cell, the result is a state in which that cell is no longer blank and instead contains the player's mark. Furthermore, control changes hands.

```
mark(M,N) :: control(W) ==> ~cell(M,N,b) & cell(M,N,W)
mark(M,N) :: control(x) ==> ~control(x) & control(o)
mark(M,N) :: control(o) ==> ~control(o) & control(x)
```

These few rules are all that we need to fully describe a game of thousands of states. That's a significant saving over the state graph. The improvement in more complex games can be even more dramatic. For example, it is possible to describe the rules of Chess in just three or four pages of rules like these. As this example illustrates, Logic Programming is well-suited to game description.

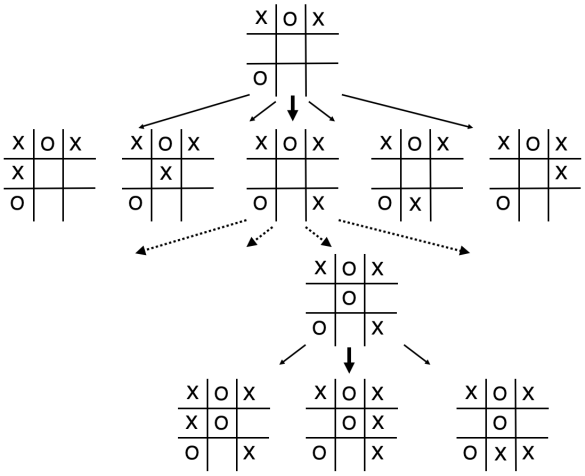
Conversely, game description is well-suited to teaching Logic Programming. The datasets are simple; the relations and operations are well-defined; and the specifications are precise. Students learn about modeling worlds in terms of objects and relations and operations. And they get to explore tradeoffs between different representations. Should we explicitly say that a cell contains a blank, as in the rules above, or should we represent "blankness" as the absence of a mark? Should we write view definitions to define relations in terms of others or should we represent our relations explicitly in our datasets and use operations to keep them up to date?

3. Game Playing

Having a formal description of a game is one thing; using that description to play the game effectively is something else entirely. The player must be able to compute the initial state of the game. It must be able to compute which moves are legal in every state. It must be able to determine the state resulting from a particular combination of moves. It must be able to compute the value of each state for each player. And it must be able to determine whether any given state is terminal.

Fortunately, languages like Prolog and Epilog have not only declarative semantics that allows us to define game rules but also practical interpreters that can be used to those process those descriptions; and we can build general game players using these tools.

A general game player typically starts with the initial state, computes the legal moves in that state, and for each move deduces the next state. In principle, this can be done repeatedly to expand the tree until a terminal state is reached on each branch. Given a game-tree of this sort, a player can use minimax or an equivalent technique to determine its best possible move; and it can save work by using more sophisticated search techniques, such as alpha-beta pruning.



Of course, for games of interesting size, it is not always possible to search to the end of the game tree. In Tic-Tac-Toe, there are a thousands of states. This is large but manageable. In Chess there are more than 10^{30} states. A state space of this size, being finite, is fully searchable in principle but not in practice. Moreover, the time limit on moves in most games means that players must select actions without knowing whether they are the best or even good moves to make.

The alternative is to do incomplete search, on each move expanding the game tree as much as possible within the available time and then making a choice based on the estimated values of non-terminal states. In traditional game playing, where the rules are known in advance, the programmer can invent game-specific evaluation functions to help in this regard. For example, in chess, we know that states with higher piece count and greater board control are better than ones with less material or lower control. Unfortunately, it is not possible for a GGP programmer to invent such game-specific rules in advance, as the game's rules are not known until the game begins. The program must evaluate states for itself. Doing this effectively is the key to general game playing.

The approach used in first-generation GGP programs was to implement heuristics that apply to all games, e.g. intermediate state values, player mobility, and opponent restriction. Consider mobility. Proponents argue that, all other things being equal, it is better to move to a state that affords the player greater mobility, that is more possible actions. Better than being boxed into a corner. Symmetrically, proponents of mobility argue that it is good to minimize the mobility of one's opponents. Although these heuristics have been shown to be effective in some games, they are only heuristics; they frequently fail, sometimes with disastrous consequences.

Monte Carlo Search is a popular alternative [4][5]. The player expands the tree a few levels. Then, rather than using a local heuristic to evaluate a state, it makes some probes from that state to the end of the game by selecting random moves for all players. It sums up the total rewards for all such probes and divides by the number of probes to obtain an estimated utility for that state. It can then use these expected utilities in comparing states and selecting actions. Monte Carlo and its variants have proven highly successful in general game playing; they changed the character of general game players from curiosities to programs capable of serious play. Virtually every general game playing program today uses some variant of Monte Carlo search.

While GGP game descriptions are encoded as game-specific logic programs, it is possible to implement general game players as game-independent logic programs that process game descriptions as data. This has significant pedagogical value in teaching Logic Programming. Prolog and Epilog are especially good for implementing methods like minimax; and Epilog (with its destructive update) is particularly good for computing probes in Monte Carlo.

Unfortunately, in GGP, we also need to worry about time, making sure that our players make moves within acceptable bounds. While it is possible to manage time in Prolog and Epilog, the resulting programs are not especially perspicuous. As a result, in the GGP community, competitive game players are typically implemented in imperative languages like C or Java or Javascript, though in most cases

they incorporate logic programming interpreters as subroutines that take game descriptions as inputs and compute game states, legal moves, updates, and so forth.

4. Metagaming

Unfortunately, Minimax and Monte Carlo search have problems of their own, especially on games with complex descriptions requiring significant computation time. The solution is for the player (1) to find ways to analyze states and expand the game tree efficiently and (2) to find ways to decrease the size of the game tree. This is where metagaming comes in.

Metagaming is match-independent game processing, with the goal of optimizing performance in playing specific matches of the game. Metagaming is usually done offline, i.e. before the game, between moves, or in parallel with game play.

There are two broad categories types of metagaming, e.g. logical optimization (program transformations that decrease the cost of exploring a game tree) and game tree restructuring (program transformations that decrease the size of the game tree).

Logical Optimization. Logical optimization involves transformation of logic programs to logically equivalent programs that run more efficiently for one's interpreter.

Subgoal ordering is a simple example. Consider the two rules shown below. They are logically equivalent, but the second is computationally superior. In the worst case, without indexing, the first rule runs in time that is $O(n^4)$, where n is the number of object constants in the language whereas the second rule runs in time that is $O(n^3)$. The good news is that there are efficient algorithms for doing this analysis and ordering subgoals.

```
s(X,Y) :- p(X) & r(X,Y) & q(X)
s(X,Y) :- p(X) & q(X) & r(X,Y)
```

Of course, this is just one technique. Other logical optimizations include subgoal elimination, rule elimination, caching / tabling [12], view materialization, reification, and conceptual reformulation [2]. View materialization is especially interesting, as it involves not just the selection of relations to materialize but also the "differentiation" [10] of relation definitions to produce update rules. Try doing that with a program written in Java!

Game Tree Restructuring. Game decomposition, also called factoring [3], is an example of game-tree restructuring. Consider the game of Hodgepodge. Hodgepodge is actually two games glued together. Here we show Chess and Othello, but it could be any two games. One move in Hodgepodge corresponds to one move in each of the two constituent games. Winning requires winning at least one of the two games while not losing the other.

What makes Hodgepodge interesting is that it is "factorable", i.e. it can be divided into independent subgames. Realizing this can have dramatic benefit. To see this, consider the size of the game tree for Hodgepodge. Suppose that the game tree for one subgame has branching a and the other has branching factor b. Then the branching factor of the joint game is a times b, and the size of the fringe of the game tree at level n is $(a * b)^n$. However, the two games are independent. Moving in one subgame does not affect the state of the other subgame. So, the player really should be searching two smaller game trees, one with branching factor a and the other with branching factor b. In this way, at depth n, there would be only $a^n + b^n$ states. This is a significant decrease in the size of the search space. Luckily, in many cases it is possible to discover such factors in time proportional to the size of the game description.

Factoring is just one example of game tree restructuring. There are many others. For example, it is sometimes possible to find symmetries in games that cut down on search space. In some games, there are bottlenecks that allow for a different type of factoring. Consider, for example, a game made up of one or more subgames in which it is necessary to win one game before moving on to a second game. In such a case, there is no need to search to a terminal state in the overall game; it is sufficient to limit search to termination in the current subgame. These examples are extreme cases, but there are many simpler everyday examples of finding structure of this sort that can help in curtailing search.

The trick in metagaming is to analyze and/or reformulate a game without expanding the entire game tree. The interesting thing about metagaming is this - sometimes the cost of analysis is proportional to the size of the description rather than the size of the game tree, as in the example above. In such cases, players can expend a little time in factoring and gain a lot in search savings.

Metagaming is an essential part of effective general game playing, and it illustrates the benefit of representing game rules as logic programs. Metagaming treats logic programs as data; and there are known algorithms for metagaming techniques like logical optimization and game tree restructuring. Although there are smart compilers for programs in imperative programming languages, those compilers are generally not capable of advanced optimizations like game factoring.

5. Conclusion

We have found that games are a particularly good application area for teaching logic programming. The datasets are simple; the views and operations are well-defined; and the constraints are precise and complete. Students learn about modeling; they learn how to use Logic Programming in search-based problem solving; and they learn metareasoning techniques where programs are treated as data. Moreover, students find games interesting. Our course on GGP is one of the most popular on campus, and students remember the experience years later.

References

[1] Y. Bjornsson and H. Finnsson. CADIAPLAYER: A simulation-based general game player. IEEE Trans. on CI&AI in Games, 1(1):4-15, 2009.

[2] R. Chirkova and M. R. Genesereth. Linearly Bounded Reformulations of Conjunctive Databases. In Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings. 987-1001. https://doi.org/10.1007/3-540-44957-4_66.

[3] Cox, E.; Schkufza, E.; Madsen, R.; and Genesereth, M. 2009. Factoring General Games Using Propositional Automata. Paper presented at the IJCAI-09 Workshop on General Game Playing (GIGA-09), July 11, Pasadena, CA.

[4] Finnsson, H. 2012. Simulation-Based General Game Playing. Ph.D. dissertation, School of Computer Science, Reykjavik University.

[5] Finnsson, H., and Bjornsson, Y. 2008. Simulation-Based Approach to General Game Playing. In Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08), 259-264. Menlo Park, CA: AAAI Press.

[6] M. R. Genesereth, N. Love, and B. Pell: General Game Playing: Overview of the AAAI Competition. In AAAI Magazine 26(2): 62-72, 2005.

[7] M. R. Genesereth, Y. Bjornsson: The International General Game Playing Competition. In AAAI Magazine 34(2): 107-111, 2013.

[8] M. R. Genesereth, M. Thielscher: Michael Genesereth and Michael Thielscher. General Game Playing. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool, 2014.

[9] M. Genesereth, V. Chaudhri: Logic Programming. Synthesis Lectures on Artificial Intelligence and Machine Learning, February 2020. https://doi.org/10.2200/S00966ED1V01Y201911AIM044

[10] L. V. Orman: Differential Relational Calculus for Integrity Maintenance, IEEE Transactions on Knowledge and Data Engineering, vol. 10, no. 2, March/April 1998.

[11] Schiffel, S., and Thielscher, M.: Fluxplayer: A successful general game player. In Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, 1191-1196, 2007.

[12] D. S. Warren: Programming in Tabled Prolog. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.4635

[13] Wikipedia: Prolog. https://en.wikipedia.org/wiki/Prolog