DOI:10.1145/3747852

While general game playing is intellectually engaging and fun, it also serves as a laboratory for practical applications, and provides a theoretical framework for defining rationality in a way that takes into account problem representation, incompleteness of information, and resource bounds.

BY MICHAEL GENESERETH

General Game Playing

GAMES OF STRATEGY, such as chess and checkers, couple intellectual activity with competition. We can exercise and improve our intellectual skills by playing such games. 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 (AI). The idea is that programs that think better should be able to win more games, so we can use game playing as an evaluation technique for intelligent systems.

Unfortunately, building programs to play specific games has limited value in this regard. To begin with, specialized game players have a very narrow focus. They can be good at one game but not another. Deep Blue[2] may have beaten the world chess champion, but it has no clue how to play checkers. A second, more subtle problem with specialized game-playing systems is 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 AI, but it requires moving more of the design work to the computer itself. This can be done by focusing attention on general game playing.

General game players are systems that accept descriptions of arbitrary games at runtime and use such descriptions to play those games effectively without human intervention. In other words, they do not know the rules until the games start.

Unlike specialized game players, such as Deep Blue[2] and Stockfish, general game players must be able to play different kinds of games. They must be able to play simple games (like tic-tac-toe) and complex games (like chess), games with differing numbers

» key insights

- **General game players are systems that accept descriptions of arbitrary games at runtime and use such descriptions to play those games effectively without human intervention.**
- **General-game-playing expertise depends on intelligence on the part of the game player and not just intelligence of the programmer of the game player.**
- **General game playing underscores the importance of knowledge representation, reasoning, and rational decision making in a world increasingly focused on machine learning and large language models.**

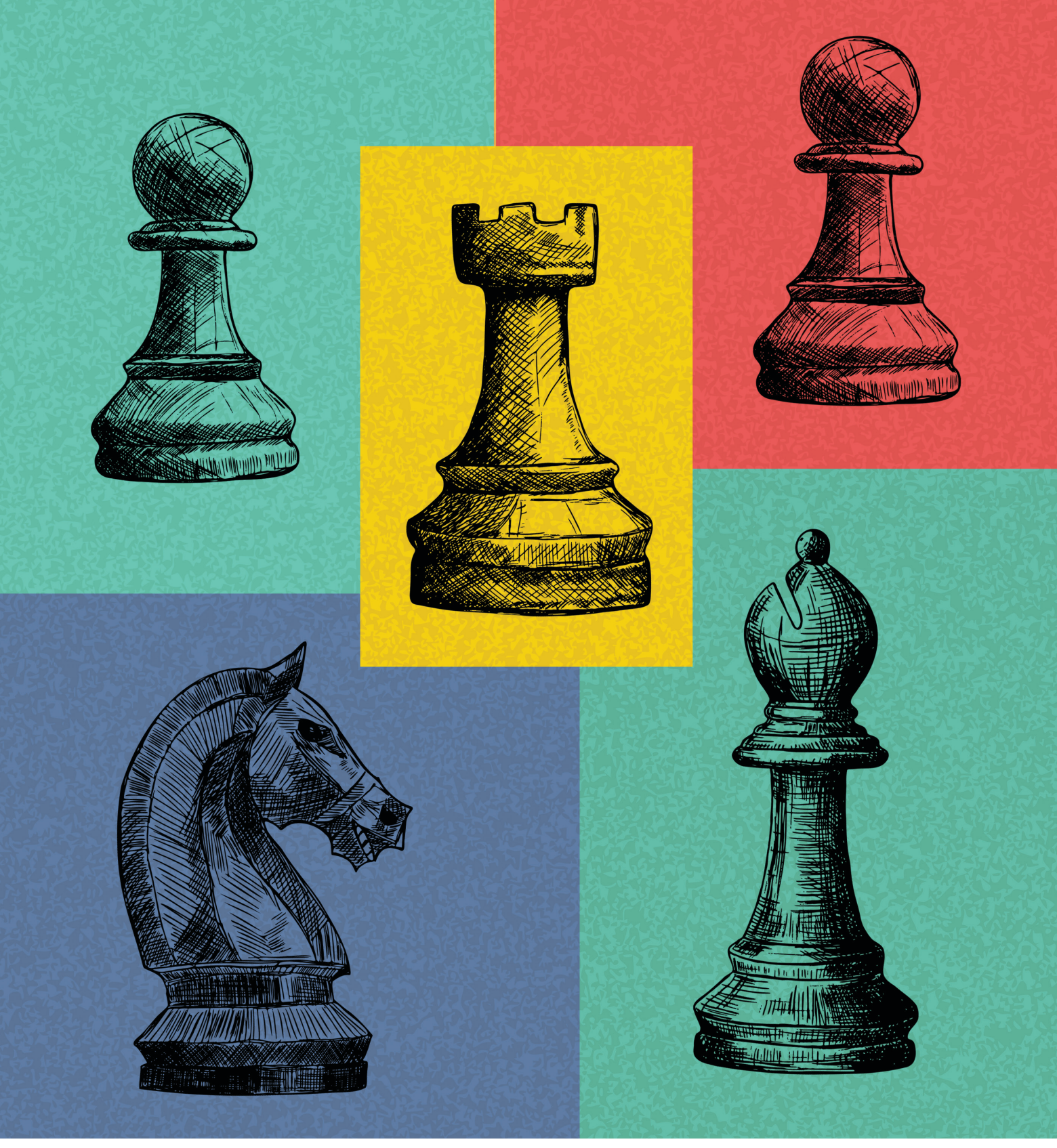


IMAGE BY ANDRIJ BORYS ASSOCIATES. USING ART BY OLEKSANDR YASHCHUK

of players, cooperative games and competitive games, games with or without communication among the players, and so forth. Importantly, they must be able to play games they have never seen before, including games no one has seen before.

Importantly, general game players cannot rely on algorithms designed in advance for specific games, as in the case of specialized game players. General-game-playing (GGP) expertise depends on intelligence on the part of the game player and not just intelligence of the programmer of the game player.

General game playing underscores the importance of knowledge representation, reasoning, and rational decision making in a world increasingly focused on machine learning (ML) and large language models (LLMs). It is worth noting that, despite important and impressive game-related developments in these latter areas (e.g. Bannerjee et al.,[1] Hausknecht et al.,[11] Levine,[16] and Silver.[25]), no ML

Table 1. International General Game Playing Competition winners, 2005 to 2016.

Year	Player	Developer(s)
2005	Cluneplayer	Clune (USA)
2006	Fluxplayer	Schiffel, Thielscher (Germany)
2007	Cadiaplayer	Bjornsson, Finnsson (Iceland)
2008	Cadiaplayer	Bjornsson, Finnsson (Iceland)
2009	Ary	Mehat (France)
2010	Ary	Mehat (France)
2011	TurboTurtle	Schreiber (USA)
2012	CadiaPlayer	Bjornsson, Finnsson (Iceland)
2013	TurboTurtle	Schreiber (USA)
2014	Sancho	Draper, Rose (UK)
2015	Galvanise	Emslie (UK)
2016	WoodStock	Piette (France)

Figure 1. 2012 Carbon vs. Silicon match: The human player taking counsel from another human.

program has ever won an official GGP competition due to time limits on game play, and it is unlikely that systems based on LLMs would perform any better, due to limits on time and space and due to "obfuscation" (wherein words in game descriptions are replaced by nonsense words). By contrast, undergraduates with minimal training in knowledge representation and reasoning methods are able to create GGP programs running on laptops that excel in such competitions and frequently beat humans.

Recent History

The idea of *general problem solving* dates from the earliest days of AI. In 1958, Newell and Simon proposed a *general problem solver*[19] that worked by applying general, task-independent problem-solving processes to descriptions of domain-specific tasks. The first explicit mention of GGP appeared in a subsequent paper by Jacques Pitrat[21] in 1968. A related notion was described by Barney Pell,[20] who further developed the concept in his 1993 doctoral thesis. In the early 2000s, Stanford University researchers enlarged the concept from chess-like games to arbitrary discrete dynamic systems, thus broadening the concept and making it applicable beyond the world of recreational games.

To promote work on GGP, in 2005 the Association for the Advancement of Artificial Intelligence (AAAI) established the International General Game Playing Competition (IGGPC),[7,8] an annual contest to determine the best automated general game players in the world. Table 1 shows the winners of the first dozen competitions.

In addition to pitting automated players against each other, the competition frequently included a demonstration match between the competition winner and a human player(s). While the human player won the first of these demonstrations, the computer won all subsequent competitions. For example, in 2012, CadiaPlayer, in addition to defeating the other automated competitors, defeated the human race, represented by Chris Welty, one of the developers of Watson, the IBM computer that beat the best players on the American game show Jeopardy! (As a consolation prize, the human was awarded two bottles of Scotch to ease his disappointment at letting down the human race.)

The International GGP Competition was suspended after 2016. However, by then, the competition had served its primary purpose—it led to workshops on GGP at multiple international conferences and the publication of numerous research papers on GGP and its applications. Today, regional competitions continue to be run, and there are rumors that the international competition will resume, albeit in a slightly different format.

Today, there are GGP sites in multiple countries around the world. The Gamemaster site[a] contains a specification of the game description language, games encoded in this language, and software for building players and running matches.

a http://gamemaster.stanford.edu

Game Playing

Despite the variety of games treated in GGP, all games share a common abstract structure. Each game takes place in an environment with finitely many states, with one 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.

Given this common structure, we can think of a game as a state graph, like the one shown in Figure 2. In this case, we have a game with one player, with eight states (named s_1, ... , s_8). The arcs in this graph capture the transition function for the game. For example, if the game is in state s_1 and a player performs action a, the game will move to state s_2. If the game is in state s_1 and a player performs action b, the game will move to state s_5. Each game has just one initial state, in this case s_1. However, there can be any number of terminal states, in this case s_4 and s_8. The numbers associated with each state indicate the utilities of those states for the player. Players earn those scores only in terminal states. However, they are provided for all states and in some games indicate incremental progress.

We can extend this model to accommodate games with multiple players with two modifications.

1. We annotate each state with rewards for all players in the game. (The rewards can be the same or different for different players.)

2. For each state, we specify which player is in control, that is, whose turn it is to play. Turns need not strictly al-

ternate; in some cases, a single player may get several turns in a row. (This model can be extended to games with simultaneous moves, but for the sake of simplicity, we avoid that complexity in what follows.)

Since all the games we are considering are finite, it is possible, in principle, to describe such games in the form of state graphs. Unfortunately, such explicit representations are not practical in all cases. 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 more than 10^{40} possible states.

In the vast majority of games, states and actions have a composite structure that allows us to define a large number of states and actions in terms of a smaller number of more fundamental entities. In chess, for example, states are not monolithic; they can be conceptualized in terms of pieces, squares, rows, columns, diagonals, and so forth. By exploiting this structure, it is possible to encode games in a form that is more compact than direct representation.

The first step in solving this problem is to conceptualize states as *datasets* (as sets of facts that are true in those states) and to conceptualize actions as operations applied to arguments, as suggested by the graph in Figure 3.

Given a "structured" state graph of this sort, the second step is to encode game rules in terms of these datasets and actions. In GGP, the most popular game description language is GDL (for Game Description Language).[17] Rules in GDL are written in Epilog,[9] a dynamic logic programming language based on Prolog. The following are the rules of tic-tac-toe as written in GDL:

```
role(x)
role(o)

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

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

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

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

goal(x,100) :- line(x) & ~line(o)
goal(x,50) :- ~line(x) & ~line(o)
goal(x,0) :- ~line(x) & line(o)
goal(o,100) :- ~line(x) & line(o)
goal(o,50) :- ~line(x) & ~line(o)
goal(o,0) :- line(x) & ~line(o)

terminal :- line(x)
terminal :- line(o)
terminal :- ~open

open :- true(cell(M,N,b))
```

There are two players: x and o. In the initial state, all cells are blank (represented here by the symbol b), and the x player has control. It is legal for the player in control to mark a cell if that cell is blank. (In GDL, symbols that begin with capital letters are

Figure 2. Game as a state graph.

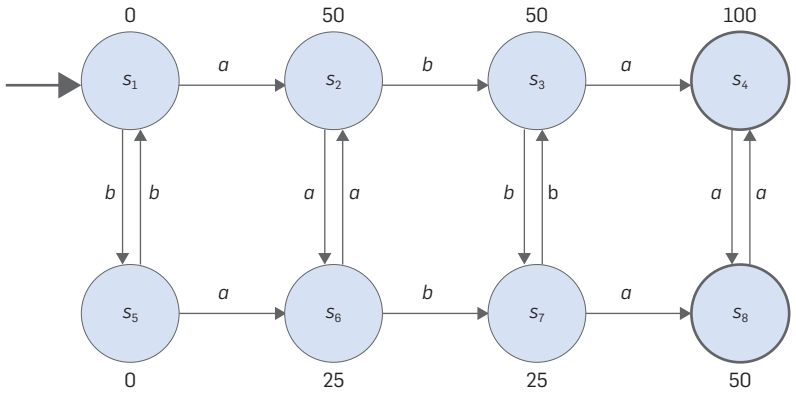


Figure 3. Game as a "structured" state graph.

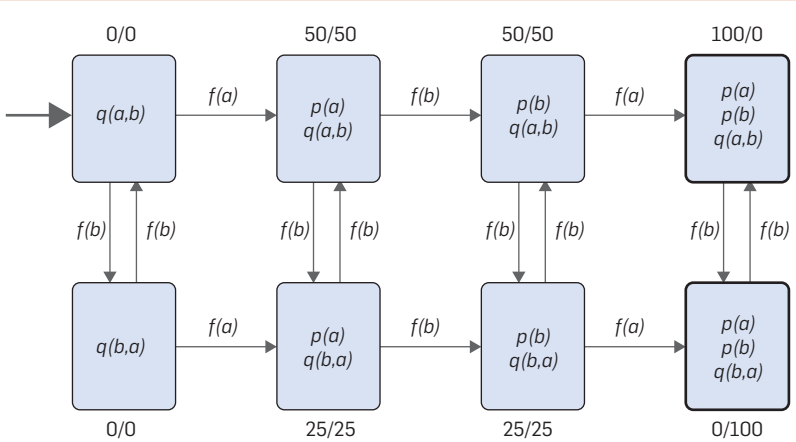
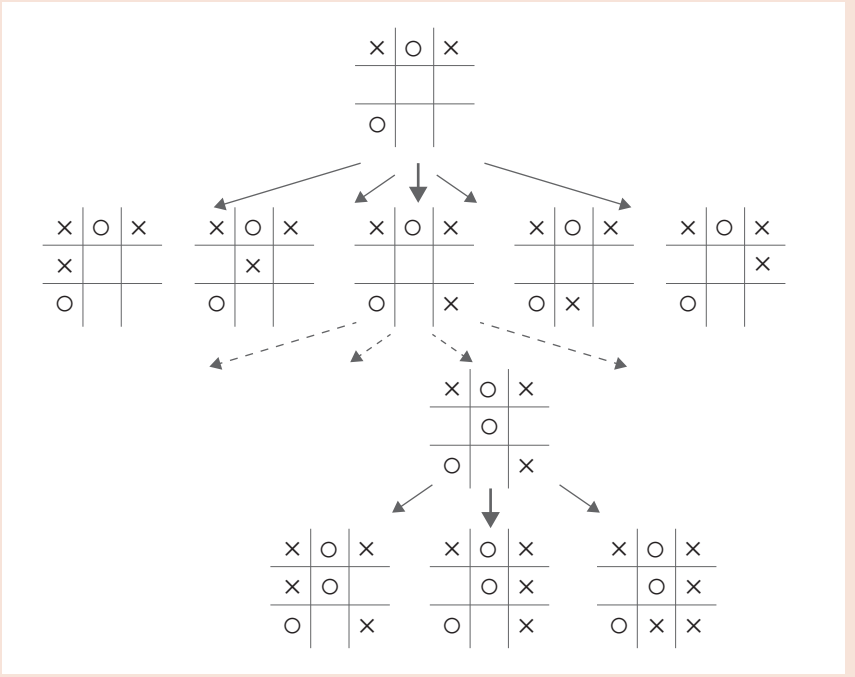


Figure 4. Partial game tree for tic-tac-toe.



variables, while symbols that begin with lowercase letters are constants. The `:-` operator is read as "if"—the expression to its left is true if the expressions that follow it are true.) If the player in control marks a cell, that cell contains the player's mark in the next state and the `b` is removed. Also, control alternates on each play. (The `::` operator relates the action mentioned to its left as a transition rule that applies whenever the conditions before the `==>` are true and produces a state in which the conditions to its right are true.) A line is a row of marks, a column of marks, or a diagonal. The `x` player / `o` player receives 100 points in any state with a line of `x` marks / `o` marks, it receives 50 points in any state with no lines, and it gets 0 points in any state with a line of `o` marks / `x` marks. Finally, a game terminates whenever there is a line of player marks or if the game is no longer open, that is, there are no blank cells.

The main thing to note about this example is that one page of rules fully describes a game of thousands of states. That is a significant savings over the state graph for tic-tac-toe (which contains more than 5,000 states). The compression in more complex games can be even more dramatic. For example, it is possible to describe the rules of chess in just four pages of rules like the ones above.

Interestingly, to prevent programmers from building in specialized capabilities for specific words in game descriptions, it is common in GGP to *obfuscate* descriptions. All words are consistently replaced by nonsense words. The only exceptions are variables and game-independent constants, such as `role`, `legal`, `terminal`.

Game Playing

Having a formal description of a game is one thing; being able to use 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 particular 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.

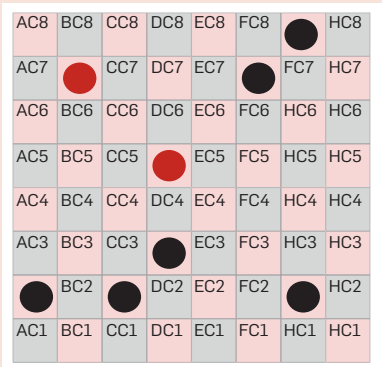
From a description such as this, a general game player can reconstruct a game tree. See the partial game tree shown in Figure 4. The player starts with the initial state, computes the legal moves in that state, and for each move deduces the next state. This is done repeatedly to expand the tree until a terminal state is reached on each branch. Given a game tree, a player can use minimax[22] or an equivalent technique to determine its best possible move, and it can use sophisticated search techniques, such as alpha-beta pruning, to save work.[14]

Unfortunately, it is not always possible to search to the end of the game tree. In tic-tac-toe, there are thousands of states. This is large but manageable. In chess, there are more than 10^{40} states.[24] 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 players must select actions without knowing with certainty 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 effective general game playing.

The approach used in early GGP programs was to develop game-independent heuristics,[3,13,15,23] for example, proximity to a goal state, 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 and offers more possible actions than to be boxed into a corner. Symmetrically, proponents of mobility argue that it is good to minimize the mobility of one's opponents. All of these heuristics have been shown to be effective in some games. Unfortunately, they are only heuristics. They frequently

Figure 5. Position in final match of IGGPC 2006.



fail, sometimes with comical consequences.

The final match of IGGPC 2006 is an example. The game was cylinder checkers, that is, checkers played on a cylinder. In this game, as in ordinary checkers, a player is permitted to move one of their ordinary pieces (pieces that are not kings) one square forward on each turn. In the position shown in Figure 5, red is moving from top to bottom, and black is moving from bottom to top. If a piece is blocked by an opponent's player, he can "jump" that player if there is an empty square on the other side. Moreover, the player must make such a jump if one is available. The objective of the game is to take all or as many of the opponent's pieces as possible while preserving one's own pieces. Here is a snapshot of the game. It is red's turn to play. What should it do? And what do you think it did?

Here's a hint. The player in this case was Cluneplayer,[3] and it had decided, for some reason or other, that limiting the opponent's mobility was a good heuristic. If it were to move the rearmost piece, black would have multiple moves. However, if it were to move the piece in front, black would be forced to capture its piece. In other words, it would have at most one move. Clearly, moving the forward piece minimizes the opponent's mobility, so that is what Cluneplayer did. Actually, the whole match played out this way, with red giving black captures at every opportunity. It was sad to watch but also a little comical. The moral is that, while non-guaranteed heuristics are sometimes useful, they are not always useful.

Monte Carlo Search

The second generation of GGP programs used a different approach to evaluating non-terminal states, notably Monte Carlo Search and Monte Carlo Tree Search.

In Monte Carlo Search (MCS), 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 Tree Search (MCTS)[6] (Figure 6) improves on this by replacing the random choice with more intelligent selections. The result of introducing such techniques was dramatic. Suddenly, automated general game players began to perform at a high level. Using MCTS, Cadioplayer won the competition three times.

The use of statistical techniques changed the character of general game players from curiosities to programs capable of serious play. Virtually every general game playing program today includes some variant of MCS or MCTS. Unfortunately, statistical search alone has weaknesses, notably on games with complex descriptions requiring a lot of computation time and on games where there are few goal states; and, consequently, it must be combined with other methods to be successful.

Metagaming

Whereas the GGP techniques described above concentrated on game tree search performed at *runtime*, sub-

Figure 6. Monte Carlo Tree Search.

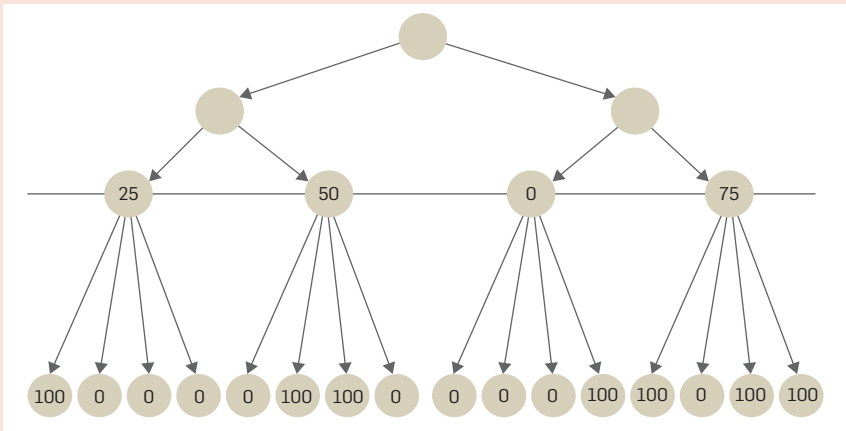
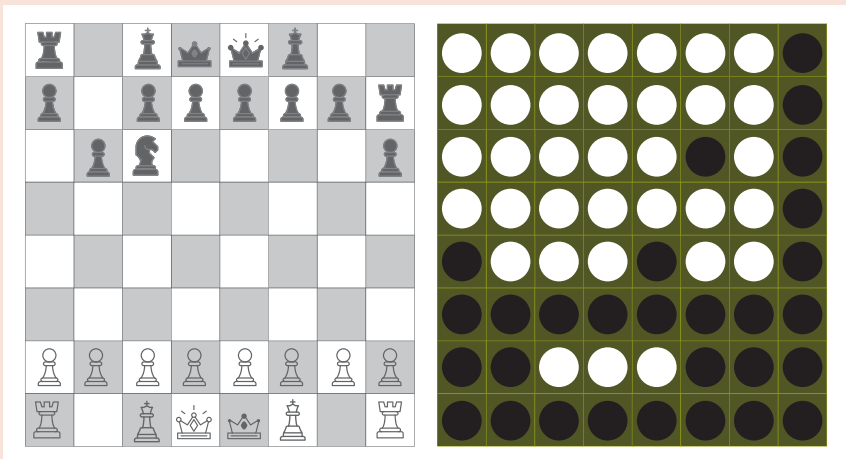


Figure 7. Hodgepodge.



sequent work was focused on *offline* processing of game descriptions. Examples include things like factoring games into independent subgames, reformulating game descriptions, and automatic programming. Often, it is the case that this sort of processing more than pays for itself. In such cases, players can expend a little time up front and gain a lot in processing time later.

This is really what programmers do when building specialized game players, and now we are building game players that can do these things for themselves. And this is what GGP was intended to stress from its very beginnings.

One example of offline analysis is game decomposition, also called *factoring*.[4,10] Consider the game of Hodgepodge pictured in Figure 7. Hodgepodge is actually two games glued together. Here we show chess and othello, but it could be any two games. One move in a joint game of 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, that is it can be divided into two independent games. Realizing this can have dramatic benefit. To see this, consider the size of the game tree for hodgepodge. Suppose one game tree has branching factor `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 huge 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.

Table 2 illustrates the computational benefits of factoring on three highly factorable games: *multiple buttons and lights*, *multiple switches*, and *multiple tic-tac-toe*. The first column in Table 2 identifies the game, the second column indicates the depth to which the tree is searched, the third column lists the number of milliseconds to search the game tree to that depth without factoring, and the last column shows the number of milliseconds after factoring.

The interesting thing about general game playing is that sometimes the cost of analysis is proportional to the size of the description rather than the size of the game tree.

Factoring is just one example of game reformulation. 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 description without expanding the entire game tree. The interesting thing about general game playing is that 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 mentioned above. In such cases, players can expend a little time and gain a lot in search savings.

Of course, factoring games and finding transformations such as these requires a lot of algorithmic expertise. What we really need is Donald Knuth in a box. Or maybe Corman and Leiserson. And, of course, since we are talking about games, we should have expertise at game tree search. So, we had better cram Jonathan Schaeffer into our box as well.

An important point here is that GGP is not just about game tree search; to an even greater extent, it is about game descriptions and their use, and ultimately about automatic programming.

Conclusion

General game playing is an interesting application in its own right. It

Table 2. Computational benefits of factoring on three highly factorable games.

Game	Depth	Unfactored Cost	Factored Cost
Multiple Buttons and Lights	4	16,700	6
Multiple Switches	5	22M	210
Multiple Tic-Tac-Toe	3	88,000	150

is intellectually engaging and more than a little fun. But it is much more than that. It serves as a laboratory for practical applications, in business and law, science and engineering. In fact, some of the games used in competitions are drawn from such areas. More fundamentally, it provides a theoretical framework for defining rationality in a way that takes into account problem representation, incompleteness of information, and resource bounds. The upshot is that it raises questions about the nature of intelligence and serves as a laboratory in which to evaluate competing approaches to intelligence.

General game playing is a setting within which AI is the essential technology. It concentrates attention on the notion of runnable specifications. Building systems of this sort dates from the early years of AI.

It was in 1958 that John McCarthy invented the concept of the "advice taker."[18] The idea was simple. He wanted a machine that he could program by description. He would describe the intended environment and the desired goal, and the machine would use that information to determine its behavior. There would be no programming in the traditional sense. McCarthy presented his concept in a paper that has become a classic in the field of AI:

> *The main advantage we expect the advice taker to have is that its behavior will be improvable merely by making statements to it, telling it about its environment and what is wanted from it. To make these statements will require little, if any, knowledge of the program or the previous knowledge of the advice taker.*

An ambitious goal! But that was a time of high hopes and grand ambition. The idea caught the imaginations of numerous subsequent researchers—notably Bob Kowalski, the high priest of logic programming, and Ed Feigenbaum, the inventor of knowledge engineering. In a paper written in 1974, Feigenbaum gave his most forceful statement of McCarthy's ideal:[5]

> *The potential use of computers by people to accomplish tasks can be "one-dimensionalized" into a spectrum representing the nature of the instruction that must be given the computer to do its job. Call it the what-to-how spectrum. At one extreme of the spectrum, the user supplies his intelligence to instruct the machine with precision exactly how to do his job step-by-step ... At the other end of the spectrum is the user with his real problem ... He aspires to communicate what he wants done ... without having to lay out in detail all necessary subgoals for adequate performance.*

Some have argued that the way to achieve intelligent behavior is through specialization. That may work as long as the assumptions one makes in building such systems are true. For general intelligence, however, general intellectual capabilities are needed, and such systems should be capable of performing well in a wide variety of tasks. In the words of Robert Heinlein:[12]

> *A human being should be able to change a diaper, plan an invasion, butcher a hog, conn a ship, design a building, write a sonnet, balance accounts, build a wall, set a bone, comfort the dying, take orders, give orders, cooperate, act alone, solve equations, analyze a new problem, pitch manure, program a computer, cook a tasty meal, fight efficiently, die gallantly. Specialization is for insects.*

References

1. Bannerjee, B., Kuhlmann, G., and Stone, P. Value function transfer for general game playing. In *Proceedings of the ICML Workshop on Structural Knowledge Transfer for Machine Learning* (2006).
2. Campbell, M., Hoare, A.J., and Hsu, F.-H. Deep Blue. *Artificial Intelligence 134*, 1-2 (2002), 57–83.
3. Clune, J. Heuristic evaluation functions for general game playing. In *Proceedings of the 22nd AAAI Conf. on Artificial Intelligence*. AAAI (2007), 1134–1139.
4. Cox, E., Schkufza, E., Madsen, R., and Genesereth, M. Factoring general games using propositional automata. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA-09)*. IJCAI (2009).
5. Feigenbaum, E. *ACM Turing Award Lectures*. Association for Computing Machinery (2007).
6. Finnsson, H. and Bjornsson, Y. Simulation-based approach to general game playing. In *Proceedings of the 23rd AAAI Conf. on Artificial Intelligence*. AAAI (2008), 259–264.
7. Genesereth, M. and Bjornsson, Y. The International General Game Playing Competition. *AI Magazine 34*, 2 (2012).
8. Genesereth, M., Love, N., and Pell, B. General game playing: Overview of the AAAI competition. *AI Magazine 26* AAAI (2005), 62–72.
9. Genesereth, M.R. and Chaudhri, V.K. *Introduction to Logic Programming*. Springer (2020).
10. Gunther, M., Schiffel, S., and Thielscher, M. Factoring general games. In *Proceedings of the IJCAI-09 Workshop on General Game Playing*. IJCAI (2009).
11. Hausknecht, M., Khandelwal, P., Miikkulainen, R., and Stone, P. HyperNEAT-GGP: A HyperNEAT-based Atari general game player. In *Proceedings of the 14th Annual Conf. on Genetic and Evolutionary Computation* (2012).
12. Heinlein, R.A. *Time Enough for Love*. Ace Books (1988).
13. Kirci, M., Sturtevant, N.R, and Schaeffer, J. A GGP feature learning algorithm. *Kunstliche Intelligenz 25*, 1 (2011), 35–41.
14. Knuth, D.E. and Moore, R.W. An analysis of alpha-beta pruning. *Artificial Intelligence 6*, 4 (1975), 293–326.
15. Kuhlmann, G. and Stone, P. Automatic heuristic construction in a complete general game player. In *Proceedings of the 21st AAAI Conf. on Artificial Intelligence*. AAAI (2006), 1457–1462.
16. Levine, J. et al. General video game playing. In *Proceedings of the Dagstuhl Seminar on Artificial and Computational Intelligence in Games* (2013), 1–7.
17. Love, N., Hinrichs, T., and Genesereth, M. Game Description Language Specification. *Stanford University* (2006).
18. McCarthy, J. Programs with common sense. In *Proceedings of theSymp. on Mechanization of Thought Processes*. National Physical Laboratory (1958).
19. Newell, A., Shaw, J.C., and Simon, H.A. Report on a general problem-solving program. In *Proceedings of the Intern. Conf. on Information Processing*. (1959), 256–264.
20. Pell, B. *Strategy Generation and Evaluation for Meta-Game Playing*. Trinity College (1993).
21. Pitrat, J. Realization of a general game-playing program. *IFIP Congress 2* (1968), 1570–1574.
22. Russell, S.J. and Norvig, P. *Artificial Intelligence: A Modern Approach*. Pearson (2021).
23. Schiffel, S. and Thielscher, M. Fluxplayer: A successful general game player. In *Proceedings of the 22nd AAAI Conf. on Artificial Intelligence*. AAAI (2007), 1191–1196.
24. Shannon, C.E. Realization of a general game-playing program. *Philos. Mag. 41*, 314 (1950), 256–275.
25. Silver, D. et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science 362*, 6419 (2018), 1140–1144.

Michael Genesereth (genesereth@stanford.edu) is a professor in the Computer Science Department at Stanford University and a professor by courtesy in the Stanford Law School, Stanford, CA, USA.