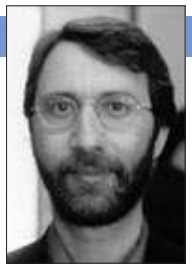
Playing with AI



By Haym Hirsh
Rutgers University
Hirsh@cs.rutgers.edu

The use of puzzles and games in AI research dates to its earliest days. In the early 1950s, Claude Shannon and Alan Turing wrote papers proposing the creation of computer programs that could successfully challenge humans in two-person games. By the late 1950s, complete programs for a number of games had been developed, most prominently in chess (by Alex Bernstein at IBM and by Allen Newell, Cliff Shaw, and Herbert Simon at the Carnegie Institute of Technology), and in checkers (by Arthur Samuel at IBM). Similarly, puzzles found their way into AI research only a short time later. The early 1960s saw the use by Newell, Shaw, and Simon of various puzzles in their development of GPS, and John McCarthy proposed the *mutilated checkerboard* puzzle as a difficult problem for automated proof methods. Thus by the mid1960s puzzles and games were viewed as valuable testbeds for AI research, both as distinctly human cognitive tasks that were hoped to be difficult yet sufficiently circumscribed to allow significant research progress, and easy-to-understand testbeds for exploring and explaining foundational questions in AI.

For many years, work in puzzle solving and (especially) game-playing has been somewhat isolated from work in other areas of AI. However, this has begun to change recently. Many ideas that have been developed in other areas of AI are now finding important applications for those working on game-playing and puzzle-solving systems. Similarly, these problems, often unfairly labeled pejoratively as "toy domains," have proven to be fertile terrain for research in AI and have contributed ideas back into other areas of AI. Through the essays of five leaders in this field, this issue's "Trends and Controversies" explores the ongoing work and lessons already learned by those applying AI to game-playing and puzzle-solving. Although each author's efforts concerns a very different task, the essays make it clear that there are common themes that underlie the various successes that they report. For example, probabilistic reasoning can be used to order or prune parts of a search space, as is done by Buro's Logistello (for Othello) and Littman's Proverb (for crossword puzzles). Intelligent and careful use of approximation can also be very important—for example, Korf's evaluation function for Rubik's Cube is based on the search space for simplified versions of the problem, and Proverb approximates the cyclical constraints of a given crossword puzzle with a tree. Finally, even more basic is the unstated but ubiquitous and intentional disregard for how humans perform the tasks in the design of their AI counterparts—in no case do the programs go about their business in a way motivated by how humans attack these problems.

Most interesting to me, however, is the extent to which the new ideas represent advances and improvements to ideas already present in the early days of the field. This is especially true for Samuel's checkers player, which learned its evaluation function from sample games (against human opponents, from book games, and from self-play). We similarly now see Logistello learn its evaluation function from self-play, and its opening book both from self-play as well as from its subsequent games with others. Much of Proverb's knowledge and success comes from large libraries of clues and solutions to past crossword puzzles. Particularly intriguing is that the speed of modern computers has now allowed some data-generation *during* a game, making it possible to, in effect, learn while playing. Sheppard's Maven (for Scrabble) uses self-play in such a fashion to estimate the value of a bound during play. Finally, also present in Samuel's work, is the fact that in addition to fast computers and clever ideas, success usually requires clever implementations. Samuel's program demonstrated this in multiple ways, such as by going so far as writing board positions to its drum memory to minimize the seek time on each subsequent drum read. Today we see similar careful awareness of the power and limitations of current computer hardware. Korf's evaluation function for the Rubik's cube requires computing enormous tables that fit into today's computer storage systems. Similarly, Schaeffer precomputes and stores all checkers endgames of eight pieces or fewer so that once Chinook reaches such a point it plays the game without error.

The essays in this "Trends and Controversies" make it clear that success in games and puzzles requires more than minimax or A* search and a fast computer, and that puzzles and games can still play an important role in AI research. Personally, given how many of the ideas can be traced back to the earliest days of our field, I also hope they will remind us to occasionally return to the writings of the early masters—they still offer us insights into accomplishing our goals today. Finally, the efforts described in these essays have the potential to help many of us reconnect with the spirit of fun offered by puzzles and games that brought many of us to this discipline in the first place.

—Haym Hirsh

Sliding-tile puzzles and Rubik's Cube in AI research

Richard E. Korf, University of California, Los Angeles

The best-known sliding-tile puzzle is the Fifteen Puzzle, shown in Figure 1a. Other sizes include the 3×3 Eight Puzzle and the 5×5 Twenty-Four Puzzle. The standard $3 \times 3 \times 3$ Rubik's Cube (see Figure 1b) also comes in $2 \times 2 \times 2$, $4 \times 4 \times 4$, and $5 \times 5 \times 5$ versions. Both these puzzles had similar histories, about 100 years apart. Finding optimal or shortest solutions to these puzzles has spawned important results in space-efficient search algorithms and admissible heuristic functions.

Sliding-tile puzzles

Sam Loyd invented the Fifteen Puzzle in the 1870s.[1] When it appeared in the scientific literature in 1879,[2] the journal's editor added the following comment:

> The "15" puzzle for the last few weeks has been prominently before the American public, and may safely be said to have engaged the attention of nine out of ten persons of both sexes and of all ages and conditions of the community. But this would not have weighed with the editors to induce them to insert articles upon such a subject in the *American Journal of Mathematics*, but for the fact that the principle of the game has its root in what all mathematicians of the present day are aware constitutes the most subtle and characteristic conception of modern algebra, viz: the law of dichotomy applicable to the separation of the terms of every complete system of permutations into two natural and indefeasible groups,... Accordingly, the editors have thought that they would be doing no disservice to their science, but rather promoting its interest by exhibiting this *a priori* polar law under a concrete form, through the medium of a game which has taken so strong a hold upon the thought of the country that it may almost be said to have risen to the importance of a national institution.

One reason for the worldwide Fifteen Puzzle craze was that Loyd offered a $1,000 cash prize to transform a particular initial state to a particular goal state. William Johnson and

William Story proved that it wasn't possible, that the entire state space was divided into even and odd permutations, and that there is no way to transform one into the other by legal moves.

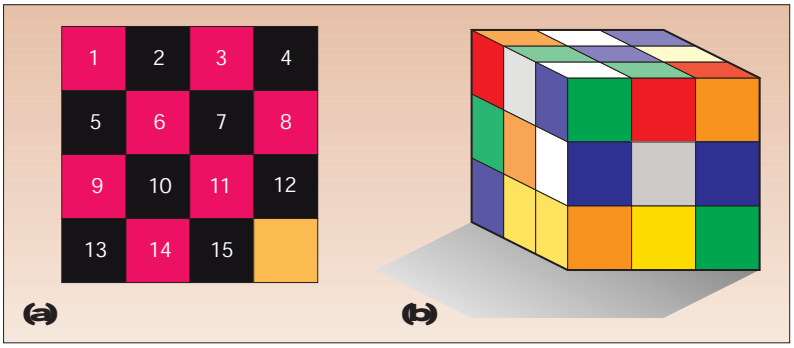


Figure 1. Fifteen puzzle (a) and Rubik's Cube (b).

Eight Puzzle. Because the state space for the Eight Puzzle contains only 181,440 (9!/2) states, it can be exhaustively enumerated. Schofield published a table listing the number of states at each distance from a given goal state.[3] The average length of a shortest path between two states is about 22 moves, and the maximum distance between any pair of states is 31 moves.

The next important advance related to this problem was the development of the A* algorithm.[4] A* introduced a *heuristic evaluation function* that estimates the distance between a pair of states. For the sliding-tile puzzles, a natural heuristic function is the Manhattan distance. It is computed by counting the number of grid units that each tile is from its goal position and summing these values for all tiles. The Manhattan distance is also a lower bound on the optimal solution length. A* is guaranteed to return an optimal path to the goal, if the heuristic function is nonoverestimating or *admissible*.

Fifteen Puzzle. Unfortunately, A* cannot compute optimal solutions to the Fifteen Puzzle, because it stores in memory every state it generates. Because the Fifteen Puzzle contains $16!/2 \approx 10^{13}$ states, the available computer memory is exhausted in minutes. This problem motivated the search for more memory-efficient algorithms and gave rise to Iterative-Deepening-A* (IDA*).[5] Based on depth-first search, IDA* generates roughly the same number of nodes as A*, but its memory requirement is only linear in the maximum search depth, eliminating the space constraint entirely. IDA* was used to find optimal solutions to 100 randomly generated Fifteen Puzzle instances, using the Manhattan-distance heuristic. Some problems generated billions of nodes, and the average optimal solution length is about 53 moves.

Twenty-Four Puzzle. While in principle IDA* with Manhattan distance could solve the 5×5 Twenty-Four Puzzle, with 10^{25} (25!/2) states, the time required is not currently practical. A more accurate admissible heuristic function is needed. The first such function was the *linear-conflict* enhancement to the Manhattan distance.[6] The key idea is that if two tiles are in their correct row, but reversed relative to their goal positions, then at least one tile must temporarily move out of the row, to allow the other to pass, and then move back. These two vertical moves do not appear in the Manhattan distance of the two tiles, but can be added to it without overestimating actual distance. The same idea applies to reversed tiles in their correct columns.

This idea can be generalized. The Manhattan distance is the sum of the distance each individual tile has to move, assuming that there are no interactions between the tiles. Viewed in this way, the next step is to consider each pair of tiles and compute how far each tile has to move, including any interactions between the pair. For most tile pairs, this *pairwise distance* equals the sum of their Manhattan distances. For some pairs, such as those in a linear conflict, for example, their pairwise distance exceeds the sum of their Manhattan distances, and this larger value can serve in an admissible heuristic. In addition to linear conflicts, two other situations give rise to larger pairwise distances, one involving tiles near the corners and the other involving tiles near the final blank position. Using linear conflicts and these additional pairwise enhancements with IDA*, we found optimal solutions to 10 random instances of the Twenty-Four Puzzle.[7] Some of these problems generated trillions of nodes, and their average solution length is over 100 moves.

An additional idea used in those experiments was pruning duplicate nodes representing the same state. Because a depth-first search, such as IDA*, doesn't store most of the states it generates, given two different paths to the same state, depth-first search generates both paths and then redundantly explores the subtrees below that state as well. Several techniques emerged for dealing with this problem, including the use of a finite-state machine to prune duplicate paths in graphs with cycles.[8] Each innovation increased the size and complexity of problems that could be solved.

Rubik's Cube

Rubik's Cube was invented in 1974 by Hungary's Erno Rubik, and like the Fifteen Puzzle 100 years earlier, became a worldwide sensation in the early 1980s. More than 100 million Rubik's Cubes have been sold. A powerful problem-solving theory, embodied in the General Problem Solver program,[9] is that to solve a problem with multiple subgoals, first find an ordering of the subgoals so that once a subgoal is solved, it doesn't have to be violated to solve the remaining subgoals. What makes Rubik's Cube so difficult that this is impossible. For example, once one plane is solved, it must be messed up, at least temporarily, to make further progress.

Learning a strategy. The solution, as millions of people discovered, is to learn a set of macro-operators, which are sequences of primitive operators, or individual twists of the cube. A useful macro-operator leaves the solved portion of the cube intact by the end of the macro application and makes further progress on the unsolved part. A computer program was written to automatically learn a set of macro-operators sufficient to solve any legally scrambled $3 \times 3 \times 3$ cube.[10] The average length of solutions generated by this strategy was about 86 moves.

Finding optimal solutions. The next challenge was to find optimal or shortest solutions to Rubik's Cube. This is easy for the $2 \times 2 \times 2$ cube, because its state space contains only 3,674,160 states. The $3 \times 3 \times 3$ cube, however, contains about 4.3252×10^{19} states. (The slogan on the box, that there are "billions of combinations," is a rather considerable understatement.) This is the number of states reachable from any

The role of games in understanding computational intelligence

Jonathan Schaeffer, University of Alberta

The AI research community has made one of the most profound contributions of the 20th century to mankind's knowledge. This research has led to the realization that intelligence is not uniquely human. Using computers, it is possible to achieve human-like behavior in nonhumans. In other words, it is possible to create the illusion of human intelligence in a computer.

This idea has been vividly illustrated throughout the history of computer games research. Unlike most of the early work in AI, game researchers were interested in developing high-performance, real-time solutions to challenging problems. This led to an ends-justify-the-means attitude: the result—a strong chess program, for example—was all that mattered, not the means by which it was achieved. In contrast, much of the mainstream AI work used simplified domains, while eschewing real-time performance objectives. This research typically used human intelligence as a model: all one had to do was emulate the human example to achieve intelligent behavior. The battle (and philosophical) lines were drawn.

The difference in philosophy can be easily illustrated. The human brain and the computer are different machines, each with its own sets of strengths and weaknesses. Humans are good at, for example, learning, reasoning by analogy, and image processing. Computers are good at numeric calculations, repetitious computations, and memorizing large sets of data. These machine architectures are largely complementary: the human's processing strengths are the computer's weaknesses and the computer's strengths are human weaknesses. Given a problem to be solved and a specified architecture (human brain or silicon computer), a good solution should cater to the strengths of the machine being used, not the weaknesses. When viewed in this light, it is not surprising that the unhuman-like approaches have won out.

Building high-performance game-playing programs has been one of AI's major triumphs. This is due, in part, to the success achieved in games such as backgammon, chess, checkers, Othello, and Scrabble, where computers are playing as well as or better than the best human players. However, its success is also due to the examples it sets for the research community. These include tackling challenging problems (rather than trivial subsets, as is still often seen in AI research) and the emphasis on the results of the system without regard for the methods used to achieve those results (the ends justify the means).

This essay illustrates a number of techniques used by game-playing programs to achieve the illusion of human-like intelligence.

Brute-force search

Humans are poor searchers. They cannot search quickly, and usually not optimally. In contrast, computers are very good at searching. Considering millions of possibilities per second, looking for a solution in a maze (or tree) of possibilities is easy to do in a computer. However, humans are very good at discovering, generalizing, and using knowledge; computers are primitive in comparison. Even after 50 years of research, no one understands how to represent and manipulate knowledge effectively. Hence, many computer-based solutions for games programs trade off knowledge for searching. They use large, deep searches to compensate for inadequate knowledge (so-called brute-force search, sometimes used in a derogatory context). Search itself is dynamic knowledge.

This idea culminated in the Deep Blue victory in an exhibition match with world chess champion Garry Kasparov in 1997. Deep Blue used a 32-processor IBM SP-2 computer, with each processor connected to 16 specially designed chess chips.[1] Each chip was capable of quickly searching large chess trees. The result was a chess machine whose search considered all possible moves at least 12 ply (one ply is one move by one side) into the future, while selectively extending the search considerably deeper for interesting moves.

Deep Blue searched 200 million chess positions *per second*.[1] Kasparov considered two. Given those numbers, to some observers it was not a surprise that the computer won the match. The real surprise was how long humans have been able to withstand the technological onslaught. The next generation of Deep Blue chess chips will be 10 times faster than those used in the Kasparov match. Improved computer technology improves the perceived "intelligence" of compute-bound AI applications.

Brute-force search is now an accepted tool in the AI researchers' toolbox, and has been used to achieve many notable successes (with games and in other domains).

Large memory

Computer memory is cheap. The price per byte of disk storage is plummeting. This trend will continue for the foreseeable future.

Human memories are notoriously fallible, have a fixed capacity, and degenerate with time. Storing large amounts of data is impractical. Humans compensate for this by distilling large amounts of knowledge into a manageable number of rules (or heuristics) that are effective at reconstructing the data. Computer memory, on the other hand, can be made infallible, expanded to fit the needs, and preserved forever. Hence, a brute-force approach to storage can be used: save everything.

The Chinook checkers program (8×8 draughts) uses this idea.[2] The game theoretic result (win, loss, or draw) for all positions with eight or fewer pieces on the board were computed: roughly 444 billion positions (4.44×10^{11}). This knowledge lets the program play perfectly when it reaches a certain position in the database (the program will always win a won position and never lose a drawn position). It also significantly affects the search, in that it introduces perfect knowledge. It is not uncommon to see a position with 20 pieces on the board being searched deeply enough to back up a database score to the root of the search. Typically, Chinook can announce the final result of the game (assuming that the human opponent does not make a mistake) within 15 moves of the start.

The database was compressed into six gigabytes for real-time decompression. While the early versions of the program were I/O bound, continually accessing the database on disk to look up position values during a search, the current version preloads the entire database into random-access memory. The resulting speed benefits only serve to widen the gap between Chinook's capabilities and what the best humans can achieve.

Unexplainable knowledge

How does one acquire knowledge for use in a game-playing program? The traditional approach is to consult human domain experts and attempt to distill rules for strong play from them. This is a difficult task, especially

given state. Like the Fifteen Puzzle, the complete state space of Rubik's Cube consists of separate components with no legal moves connecting them—in fact, 12 such components for the standard cube.

Optimal solutions to Rubik's Cube required using IDA* with a heuristic based on *pattern databases*.[11,12] Herbert Kociemba of Germany developed similar ideas independently. The standard Rubik's Cube consists of 27 subcubes, or *cubies.* The 20 movable cubies include eight cubies on the cube's corners and 12 cubies on the edges. Corner cubies stay on the corners; edge cubies stay on the edges. The total number of different permutations and orientations of the corner cubies is only 88,179,840. Thus, with a breadth-first search that ignores the edge cubies, we can compute the exact number of moves required to solve each state of the corner cubies and store these values in memory. Because this number ranges from zero to 11 moves, each entry requires only four bits, for a total of 42 Mbytes of storage. Because any solution must solve both the corner and edge cubies, the number of moves to solve just the corner cubies is a lower bound on the total number of moves required to solve the cube. Thus, this value is an admissible heuristic.

given the difficulty humans have in expressing their subconscious decision-making processes. For a computer solution, one ideally wants to eliminate the weak link—the human expert. The computer should be able to discover and refine all the knowledge it needs. While this goal remains elusive in general, there have been some notable successes in the games domain.

The Othello program Logistello plays Othello better than all humans.[3] The program evaluates positions using 11 patterns that, with reflections and rotations, come to 46. Assigning a score to each possible value for each pattern results in roughly 1.2 million evaluation scores that need to be determined. Using self-play (Logistello playing games against itself) and linear regression, the program can incrementally learn the value of these parameters. With relatively little effort (roughly a month of computation), the program achieves world-class ability. In effect, this is a brute-force approach to integrating knowledge. Program designers can include as much (or as little) knowledge as they like, and let the parameter-learning process decide what is used and how important it is. The TD-Gammon backgammon program pioneered similar techniques where self-play and temporal difference learning of a neural net resulted in play that is comparable to that of the human world champion.[4]

There is little in the way of useful information that humans can extract from the large number of seemingly random numbers with which Logistello evaluates positions. Just as computer program designers have difficulty understanding human knowledge, so too do humans have difficulty understanding computer "knowledge."

Simulations

The early research into games was restricted to two-player, perfect-information games. With the decline of interest in chess research in the 1990s, efforts switched to other games, including those with multiple players and having imperfect information. Imperfect information provides an interesting challenge. In these types of games, humans observe their opponent's actions and make inferences as to the missing information. Human intuition and experience can be amazingly accurate. Computers have difficulty approximating experience and intuition, but are capable of precise probability calculations. The computer's solution is simulation: instantiate the missing information many times, each time calculating the likely outcome. In this way, a statistical profile can be obtained that indicates the computer's best course of action.

Bridge,[5] poker,[6] and Scrabble[7] programs—all games of imperfect information—use similar techniques to achieve their success: they simulate hundreds or thousands of scenarios. For example, the bridge program GIB decides what card to play by simulating the play of the hand.[5] The program internally deals out cards that are consistent with the bidding to the opponents. It then plays out the hand to the end to see which card play leads to the best result (most tricks won). It repeats this process roughly 100 times, each time with different cards for the opponent. After enough simulations, it becomes clear which card play, on average, leads to the best result. The program does not understand well-known bridge concepts such as finesse or squeeze; everything is done using uninformed search.

Comprehension without understanding

Documents contain human-understandable information. Because this information was designed to be easily understood by a human (such as text, sounds, and images), it is often difficult for a computer to figure it out. This communications gap between man and machine is an imposing obstacle to technology. No one wants to re-express human knowledge in computer-understandable terms unless absolutely necessary. We need to make computers understand documents. The computer solution is to create the illusion of understanding, without actually doing any comprehension.

The Proverb program solves crossword puzzles.[8] Human crossword solvers use the semantics of the clues to deduce the answers needed for the puzzle. The clues are intended for a human audience and include a combination of factual information, common knowledge, missing words, and word plays. Writing a computer program to understand the semantics is a challenging problem. Proverb's solution is to not understand the clues. It includes a variety of solvers (or agents) that examine the words in the clues to identify likely answers. For example, Proverb uses a database of clues and answers from previous puzzles to find an answer to a clue 34% of the time. Specialized agents can go out on the Internet and query, for example, dictionary, history, geography, and movie databases. Each agent returns a set of answers. Proverb combs through the plausible answers, trying to fit them into the puzzle grid and satisfy all the constraints. Proverb scores 95% letters correct on the *New York Times* crossword puzzles, without understanding the meaning of any of the clues.

Comprehension without understanding is a powerful technique. Similar ideas are being applied to, for example, classifying Web pages. Given an arbitrary Web page, is it possible to classify its type (personal page, company, course, and so forth)? Statistical techniques can do an effective job. For example, counting the frequency of words appearing on the page can be a powerful indicator of the type of the page. No human would ever explicitly compute word frequencies to do this.

COMPUTER GAMES research is a microcosm for AI research. By liberating the computer from blindly following the human example, amazing feats of intelligence are possible with relatively little computer programming effort. Our notion of intelligence will never be the same.

References

1. F-H. Hsu, "IBM's Deep Blue Chess Grandmaster Chips," *IEEE Micro*, Mar./Apr., 1999, pp. 70–81.
2. J. Schaeffer, *One Jump Ahead*, Springer-Verlag, New York, 1997.
3. M. Buro, "Logistello—A Strong Learning Othello Program," 1997; www.neci.nj.nec.com/homepages/mic/ps/log-overview.ps.gz.
4. G. Tesauro, "Temporal Difference Learning and TD-Gammon," *Comm. ACM*, Vol. 38, No. 3, 1995, pp. 58–68.
5. M. Ginsberg, "GIB: Steps Toward an Expert-Level Bridge-Playing Program," *Proc. Int'l Joint Conf. AI,* AAAI Press, Menlo Park, Calif., 1999, pp. 584–589.
6. D. Billings et al. "Using Probabilistic Knowledge and Simulation to Play Poker," *Proc. AAAI Nat'l Conf.,* AAAI Press, Menlo Park, Calif., 1999, pp. 697–703.
7. B. Sheppard, private communication, Oct. 1998.
8. G. Keim et al., "Proverb: The Probabilistic Cruciverbalist," *Proc. AAAI Nat'l Conf.,* AAAI Press, Menlo Park, Calif., 1999, pp. 710–717.

Armand Prieditis first proposed using the number of moves needed to solve the corner cubies as a heuristic for the whole cube.[13] Additional pattern databases, based on subsets of the edge cubies, can also be built, with the final heuristic being the maximum of the individual heuristic values. Initially, 10 randomly generated initial states were solved optimally using this technique.[11] One of these states generates over a trillion nodes, and the median optimal solution length is 18 moves.

Conclusions and further work

I have chronicled the major milestones in finding optimal solutions to sliding-tile puzzles and Rubik's Cube. The numbers of nodes generated in these experiments increased from 181,440 for the Eight Puzzle in 1967, through billions for the Fifteen Puzzle in 1985, to trillions for the Twenty-Four Puzzle and Rubik's Cube in 1996. Given this trend, it is tempting to ascribe all this progress simply to faster computers. While Moore's Law has played an important role, these results would not have been possible without A*, linear-space algorithms such as IDA*, and more accurate admissible heuristic functions, such as pattern databases. Even today's computers

couldn't find optimal solutions to the Fifteen Puzzle with just brute-force search. Similarly, finding optimal solutions to the 6×6 Thirty-Five Puzzle or the $4 \times 4 \times 4$ Rubik's Cube will in all likelihood require the development of new algorithmic techniques.

Another problem is finding the diameter of these problem-space graphs. In other words, how many moves apart can two states be, via a shortest solution path. This number is 31 moves for the Eight Puzzle and 80 moves for the Fifteen Puzzle,[14] but it's not known for any larger versions. The corresponding value for the $2 \times 2 \times 2$ Rubik's Cube is 11 moves, but is unknown for any larger cubes. The current conjecture for the diameter of the $3 \times 3 \times 3$ Rubik's Cube is 20 moves.

While most research on games and puzzles has strived to achieve expert human-level performance, finding optimal solutions to sliding-tile puzzles or Rubik's Cube is far beyond human capabilities. Even expert cubists that can solve a cube in less than 30 seconds employ over 50 moves, compared to an optimal solution of 20 or less.

References

1. S. Loyd, *Mathematical Puzzles of Sam Loyd: Selected and Edited by Martin Gardner*, Dover, New York, 1959.
2. W.W. Johnson and W.E. Storey, "Notes on the 15 Puzzle," *Am. J. Mathematics,* Vol. 2, 1879, pp. 397–404.
3. P.D.A. Schofield, "Complete Solution of the Eight Puzzle," *Machine Intelligence 3*, American Elsevier, New York, 1967, pp. 125–133.
4. P.E. Hart, N.J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. Systems Science and Cybernetics*, Vol. SSC-4, No. 2, July 1968, pp. 100–107.
5. R.E. Korf, "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, Vol. 27, No. 1, 1985, pp. 97–109.
6. O. Hansson, A. Mayer, and M. Yung, "Criticizing Solutions to Relaxed Models Yields Powerful Admissible Heuristics," *Information Sciences*, Vol. 63, No. 3, 1992, pp. 207–227.
7. R.E. Korf and L.A. Taylor, "Finding Optimal Solutions to the Twenty-Four Puzzle," *Proc. 13th Nat'l Conf. AI,* AAAI Press, Menlo Park, Calif., 1996, pp. 1202–1207.
8. L. Taylor and R.E. Korf, "Pruning Duplicate Nodes in Depth-First Search," *Proc. 10th Nat'l Conf. AI*, AAAI Press, Menlo Park, Calif., 1993, pp. 756–761.
9. A. Newell and H.A. Simon, "GPS, a Program that Simulates Human Thought," *Computers and Thought,* E. Feigenbaum and J. Feldman, eds., McGraw-Hill, New York, 1963, pp. 279–293.
10. R.E. Korf, "Macro-Operators: A Weak Method for Learning," *Artificial Intelligence,* Vol. 26, No. 1, 1985, pp. 35–77.
11. R.E. Korf, "Finding Optimal Solutions to Rubik's Cube Using Pattern Databases," *Proc. 14th Nat'l Conf. AI,* AAAI Press, Menlo Park, Calif., 1997, pp. 700–705.
12. J.C. Culberson and J. Schaeffer, "Pattern Databases," *Computational Intelligence,* Vol. 14, No. 4, 1998, pp. 318–334.
13. A.E. Prieditis, "Machine Discovery of Effective Admissible Heuristics," *Machine Learning*, Vol. 12, 1993, pp. 117–141.
14. A. Brungger et al., "The Parallel Search Bench ZRAM and its Applications," *Annals of Operations Research*, Vol. 90, 1999, pp. 45–63.

How machines have learned to play Othello

Michael Buro, NEC Research

Today's top programs for perfect-information games use a variety of techniques to increase the move quality subject to the limited time available in tournaments. These programs

- achieve a high raw search speed by means of assembler routines or by using very fast parallel or special-purpose hardware that allows deep game-tree searches and thereby enables playing a strong game even if only poor evaluation functions are used;
- use smart evaluation functions that are often automatically tuned;
- perform selective searches to follow interesting variations more deeply or to cut off probably irrelevant lines of play early, without missing many decisive variations; and
- utilize large opening books and perfect endgame databases for improving performance in the opening and endgame phases.

The combination of all these techniques is ideal for top-level play.

Unfortunately, there are incompatibilities as well as trade-offs between these techniques. For instance, affordable hardware realizations require a simple structure both for the evaluation function and the selective search mechanism. These restrictions might cause a lower playing strength than expected compared to that of a workstation implementation of a smarter search algorithm coupled with a better evaluation function. On the other hand, weaker but faster evaluation functions allow deeper searches that might lead to a better overall performance than the use of smart but slow functions in conjunction with shallower searches.

Despite these design problems, game designers can often improve existing implementations by working on each of the mentioned topics separately, aiming for the right balance. This is very important, because neglecting one issue can reduce the overall performance considerably.

In this essay, I describe the cornerstones of Logistello, which dominated the computer Othello scene from 1993 until 1997. In late 1997, it retired from active tournament play after beating the then human world champion Takeshi Murakami 6-0 (see Figures 2 and 3).[1] Although Othello—a popular Japanese board game—served as a test domain for research in evaluation function construction, selective search, and opening book learning, the novel ideas I will discuss are game-independent and worth considering in other games and domains.

Positional evaluation—GLEM

Many AI systems use evaluation functions for guiding search tasks. In the context of strategy games, they usually map game positions into the real numbers for estimating the winning chance for the player to move. Decades of research have shown how hard a problem-evaluation function construction is, even when focusing on particular games.

To simplify the construction task, the notion of evaluation features emerged. This notion assumes that there exist reasonable approximations of the perfect evaluation function in the form of combinations of a few distinct numerical properties of the position—called *features*. Given this, evaluation functions can be constructed in two phases by selecting features and combining them.

Selecting features is one of the most important and difficult subtasks in constructing a game-playing program. It requires both domain-specific knowledge

and programming skills because of the well-known trade-off between speed and knowledge in game-tree search. A couple of years ago, the authors of the best game-playing programs still picked not only features but also their weights in the course of a tedious optimization process. This is somewhat surprising, because in 1959, Arthur Samuel had already proposed ways for automatically tuning weights.[2] While selecting features is difficult for a machine, fitting even a large number of weights given a set of training positions is not. Research focused on the latter topic produced TD-Gammon, a world-class backgammon program written by Gerry Tesauro.[2]

Figure 2. Takeshi Murakami playing Logistello.

Figure 3. Othello board.

In the effort of improving Othello's evaluation function, I went a step further toward the ultimate goal of automatic evaluation-function construction. Based on a generalized linear-evaluation model—called GLEM—I have developed efficient procedures for generating training positions, exploring the feature space, and fitting feature weights.[3] Rather than combining a few features by using complicated nonlinear functions, I propose to construct evaluation functions by linearly combining a large number of features that are Boolean combinations of atomic relations. This approach lets us model nonlinear effects directly, without the detour over analytic functions and opens up practical ways for generating features automatically. While refining the evaluation model, Logistello's evaluation function underwent drastic changes from a classic form—featuring only a handful of manually weighted features—to its final version, which utilizes approximately 100,000 binary features in conjunction with over 1.2 million automatically tuned parameters. Observing that short Boolean combinations of simple binary features (such as, "is a white disc on h8?") can approximate important Othello concepts combined with the "mechanical" analysis of millions of training positions has produced an expert program capable of beating any human player. Interestingly, the game knowledge encoded by the set of over a million configuration weights goes far beyond the mobility features we intended the system to approximate in the first place.

This result encourages the application of GLEM to other games or even to search or decision problems in other domains. Attractive candidates are chess and Go because both games are very popular and well analyzed. And yet, for chess, hardware roughly equivalent to 2,000 ordinary PCs is currently needed to compete with the human world champion. For Go, the status is even worse because brute-force search is infeasible due to the large branching factor. Because a good evaluation function is not known either, amateurs can still beat the best Go programs handily. In our opinion, the key to better chess and Go programs lies in improved evaluation functions. A starting point could be the analysis of known features with regard to their approximation by simple Boolean functions as proposed by GLEM.

Selective search—Multi-ProbCut

Human players can find good moves without searching the game tree in its full width. Using their experience, they can prune unpromising variations in advance. The resulting game trees are narrow and might be rather deep. By contrast, the original minimax algorithm searches the entire game tree up to a certain depth and even its efficient improvement—the alpha-beta algorithm—may only prune backwards because it must compute the correct minimax value.

The selective-search procedure ProbCut permits pruning of subtrees that are unlikely to affect the minimax value and uses the time saved to analyze more probably relevant variations. This approach capitalizes on the fact that values returned by minimax searches of different depths are highly correlated provided that a reasonably good evaluation function and—if necessary—a quiescence search is used. In this case, we would expect that a shallow search result $v(s)$ is a good predictor for the deep minimax value $v(d)$. Based on this estimation, we could determine whether the deep minimax value lies outside the current alpha-beta window with a prescribed likelihood. If so, the position need not be searched more deeply because the deep search result will unlikely change the root's minimax value. Otherwise, the deep search is performed yielding the true value. Here, a shallow search has been invested, but relative to the deep search the effort involved is negligible, due to the exponential tree sizes.

A natural way to express the relationship between search results of different depth is a linear model of the form $v(d) = a^*v(s) + b + e$ where a and b are real constants and e is a normally distributed error variable having mean 0 and variance σ^2. Once all model parameters are estimated by linear regression applied to a large number of training pairs $(v(d), v(s))(i)$, ProbCut can test the cut conditions $v(d) \le \alpha$ and $v(d) \ge \beta$ efficiently during game-tree search: after computing the shallow search result $v(s)$, the search terminates in the current position if $a^*v(s) + b$, which is an unbiased estimator for $v(d)$, lies outside of $[\alpha - t^*\sigma, \beta + t^*\sigma]$. Here, t is an adjustable confidence parameter that can be optimized by means of tournaments.

In the first ProbCut implementation used in Logistello, $s = 4$ and $d = 8$ were chosen and $t = 1.5$ was empirically found to be the best cut threshold. For this parameter con-

stellation, the winning percentage of the ProbCut-enhanced version of Logistello playing against the brute-force version was 74% in a 70-game tournament.

Although ProbCut already marks a big—and potentially game-independent—improvement over brute-force alpha-beta search, it can still be refined in several ways. Multi-ProbCut (MPC)

- allows for pruning at different search heights,
- uses game-stage dependent cut thresholds, and
- conducts shallow check searches using iterative deepening.[4]

The latter improvement detects extreme positions much earlier. Incorporated in Logistello, MPC featuring up to ($s = 5$, $d = 17$) cuts and two cut thresholds (for the opening and middle game) beats regular ($s = 4$, $d = 8$) ProbCut by about 72%. At equal search times, MPC looks five to seven plies further ahead in selected lines compared with brute-force alpha-beta search and achieves a winning percentage of about 80%.

In summary, for Othello and the chosen evaluation function, MPC significantly outperforms ProbCut as well as brute-force alpha-beta search. MPC's amazing performance demonstrates that the alpha-beta algorithm wastes most of its time by analyzing irrelevant variations. MPC, on the other hand, detects potential bad moves very early and postpones their further investigation. In this way, it concentrates on probably relevant lines of play without overlooking crucial tactical variations near the root. It remains to be shown whether MPC can be successfully applied to other games. Because it coexists with most of the alpha-beta enhancements currently used in chess programs, MPC might improve these programs, too.

Opening book learning

In spite of evaluation and search improvements, programs still show weaknesses in the opening phase, stemming from a lack of strategic planning. To mitigate this problem, game developers use opening books that store move sequences or positions together with moves. Their automatic generation was of little interest up to now, because move sequences can be taken from the literature, suited to one's own requirements (such as the striving for tactical complications) and manually updated if necessary. Today, many game-playing programs are attached to servers, playing against human players and other programs 24 hours a day. Thus, it has become necessary for the programs to update their opening books automatically without human intervention.

If a player wants to be successful not only in a single game against an unknown opponent but in a sequence of games, he or she might face simple but effective playing strategies by the opponent that cannot be met by the well-known game-tree search techniques alone. Perhaps the most obvious and simple one is the following: "If you have won a game, try it the same way next time." A program with no learning mechanism and no random component follows this strategy, but is also a victim of it, because it does not deviate and therefore can lose games twice in the same way. To avoid this, the program must find reasonable move alternatives. It can do so passively, as the following strategy shows: "Copy the opponent's winning moves next time when colors are reversed." This elegant method lets the opponent show you your own faults so you can play the opponent's winning moves next time by yourself. In this way, even an otherwise stronger opponent can be compromised, because—roughly speaking—eventually he is playing against himself. Thus, copying moves makes it necessary to come up with good move alternatives actively. To do so, a player must understand his winning chances after deviations from known lines.

These basic requirements of a skilled match strategy lead directly to an algorithm for guiding opening book play based on minimax search.[5] The procedure builds a game tree from played variations—starting with the initial game position—and labels the leaves depending on the particular game outcomes. Moreover, in each interior node, the algorithm evaluates the heuristically best move not played so far and adds the corresponding edge and node together with its evaluation to the tree. Given such a tree, the program can easily guide the opening book play by propagating leaf evaluations to the root using the minimax algorithm.

Several of today's best Othello programs have effectively used variations of this opening-book algorithm: surprises in tournament games caused by blindly following nonevaluated opening lines are no longer to be feared, many programs playing on the Othello server (telnet:external.nj.nec.com: 5000) are improving their books autonomously, and extensive automatic book preparation by self-play is now possible that has revealed refutations of many common opening lines used by human players.

Outlook

All of today's top Othello programs use variations of the evaluation, search, and opening-book algorithms I've discussed. Whether they can successfully apply to other games is currently under investigation. Anyway, after the great success of learning backgammon and Othello programs, it now seems clear that future progress in more complex problems (in which brute-force search is infeasible)—such as Go—also depends on advances in machine learning.

Because games can serve as "most simple but already hard" prototypes of real-world decision problems, games research is an important branch of AI. Construction of problem-solving algorithms in complex domains greatly benefits from a practical framework for automatic feature construction, training-set generation, weight assignment, selective search, and post mortem analysis. Although the work on Othello I've discussed has opened doors in these directions, there is much room for improvement—which becomes apparent when picking the next harder game on the list.

References

1. M. Buro, "The Othello Match of the Year: Takeshi Murakami vs. Logistello," *ICCA J.,* Vol. 20, No. 3. 1997, pp. 189–193.
2. A.L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM J Research and Development*, Vol. 3, No. 3, 1959, pp. 211–229.
3. M. Buro, "From Simple Features to Sophisticated Evaluation Functions," *First Int'l Conf. Computers and Games (CG'98), Lecture Notes in Computer Science*, Springer-Verlag, New York, Vol. 1558, 1998.
4. M. Buro, *Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello*, NECI Tech. Report #96, Princeton, N.J., 1997.
5. M. Buro, Toward Opening Book Learning, *ICCA J.,* Vol. 22, No. 2, 1999, pp. 98–102.
6. G. Tesauro, "Temporal Difference Learning and TD-Gammon," *Comm. ACM*, Vol. 38, No. 3, 1995.

Mastering Scrabble

Brian Sheppard, Hasbro

If I had to identify one factor that enabled recent advances in game AI, it would be that programmers have a large arsenal of methods to adapt to their needs. An example is in order, so I'll describe Maven, my Scrabble AI.

The most important skill in Scrabble is the ability to find high-scoring plays. So Maven includes an exhaustive move generator, which produces each legal move, along with a score and a list of the tiles remaining on the rack. Thus, Maven achieves this critical skill using full-width search.

A note about word lists is in order. Nowadays, I can get a computerized word list from the National Scrabble Association (NSA). But this is a recent advance. The development of Maven included several man-months of data entry. I will describe the process, because every game AI project involves similar drudge work.

I bought a copy of the *Official Scrabble Players Dictionary* (OSPD) from my local bookstore and started typing the words. I couldn't bear to type every single word, so I invented a "little language" of the form "v assert -or -ors," which I postprocessed into "ASSERT ASSERTED ASSERTING ASSERTS ASSERTOR ASSERTORS." This trick cut the typing in half.

Then came a verification stage, where I statistically profiled the word list to determine its error rate. My initial data entry omitted 2% of the words and misspelled 1%. There ensued a proofreading chore to correct these errors. I then validated key lists of words such as the two-letter words, JQXZ words, and so forth, against printed lists from the NSA. This step ensured that any remaining errors were unlikely to matter, because they would occur among low frequency words. Eight errors remained (out of 95,000 words), which I found several years later when I compared my list against the list of another person who had undertaken the same task.

Then I decided to add the "long words" to my list, because the OSPD only contains main entries up to eight letters long. This process involved scanning *Webster's 10th Collegiate Dictionary*, proofreading, profiling, cross-checking, and so on. It was a huge task, but very typical of game AI development.

Evaluation functions

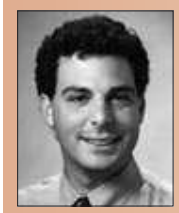
To evaluate a position properly you have to model the factors that are important to the domain. Maven's development is interesting because there wasn't a well-developed positional model of Scrabble at the time. At least, there was none that I could find. Of course, experts used certain precepts in choosing plays, but I didn't know any expert players, and I didn't have access to any of their writings. I had to model the domain "from first principles." I might have been lucky in this regard, because almost every precept held by experts prior to the advent of Maven has been proven false. Since modeling is a messy task that nearly every game AI developer has to do sometime, I will walk you through the steps I followed.

I reasoned that a move changes three things: the score, the tiles held by the player, and the position. So, in gross terms, I have the equation Evaluation = Score + Rack + Position. This is a good start because my move generator already computes the score and the tiles left on the rack (the *rack leave*). I was confident that I could build an evaluator for rack leaves, because I had a trick up my sleeve. But what should I do about this annoying *Position* term? Did I have to develop a complicated (and slow) pattern-matching algorithm for evaluating the myriad possible changes in position that could occur as a result of a move?

Upon reflection, I decided that the Position term was usually very close to 0, so I could ignore it (with one exception). The reason is that the board is a resource that affects both players, so any openings for high scores tend to cancel out. The opponent's advantage is that he moves first, so a hot spot is more likely to benefit him. Maybe you should penalize hot spots by a small amount, but maybe not. You have to consider that the opponent is a weaker player than Maven, so hot spots disproportionately benefit Maven. The only

Michael Buro is a scientist at the NEC Research Institute. He wrote Logistello, the world champion-class Othello program. He earned a diploma in computer science from the Technical University of Aachen and a PhD in machine learning in games from the University of Paderborn, Germany. He is a member of the AAAI and the ICCA. Contact him at NEC Research Inst., 4 Independence Way, Princeton, NJ 08540; mic@research.nj.nec.com; www.neci.nj.nec.com/homepages/mic/mic.html.

Richard E. Korf is a professor of computer science at the University of California, Los Angeles. He received his BS from MIT, and his MS and PhD from Carnegie-Mellon University, all in computer science. His research is in the areas of problem solving, planning, and heuristic search in artificial intelligence. He received an NSF Presidential Young Investigator Award and is a fellow of the AAAI. Contact him at the Computer Science Dept., UCLA, Los Angeles, CA 90095; korf@cs.ucla.edu; www.cs.ucla.edu/~korf.



Michael Littman is an assistant professor of computer science at Duke University. His main interests are in machine learning, examining algorithms for decision-making under uncertainty, and statistical natural-language processing. He received his PhD from Brown University, and his master's and bachelor's degrees from Yale University. His crossword work was chosen for the best paper award at AAAI 99. Contact him at Box 90129, Duke Univ., Durham, NC 27708-0129; mlittman@cs.duke.edu; www.cs.duke.edu/mlittman/.

Brian Sheppard is director of technology at Hasbro Interactive. His research interests include heuristic search and multiplayer network games. He received a BA in mathematics from Harvard College. He is the author of the Scrabble program Maven, one of the first programs to achieve championship caliber in any game. Contact him at Hasbro Interactive, 50 Dunham Rd., Beverly, MA 01915; bsheppard@hasbro.com.

Jonathan Schaeffer is a professor in the Department of Computing Science at the University of Alberta. His research interests include heuristic search and parallel-computing environments. He received a BSc from the University of Toronto and an M.Math and a PhD from the University of Waterloo. He is a member of the IEEE, ACM, AAAI, and ICCA. Contact him at the Dept. of Computing Science, Univ. of Alberta, Edmonton, Alberta, Canada T6G 2H1; jonathan@cs.ualberta.ca; www.cs.ualberta.ca/~jonathan.

Coming Next Issue

Special Millenium Issue: The Experts Expound on AI's Greatest Trends and Controversies

exception is that direct access to triple-word squares is a factor that should be evaluated, because such a spot is high scoring, easy to use, and unlikely to be left around for the next turn. Still, calculation showed that direct access to triple-word square is only worth a few points (usually under three).

As for Rack evaluation, I whipped out my trick: I would use self-play to generate games and "feed back" the impact of holding specific tiles into the evaluation function. This method worked well in Scrabble. In fact, the evaluation function improved from zero initial knowledge to beyond the level of the human champions of the day, while using only a single day of training.

Self-play combined with feedback is a fundamental method employed in most competitive programs. It works in other nondeterministic games, and in deterministic games, too, if combined with tricks that ensure exploration.

But self-play can only take you so far. Self-play brings a program into greater internal consistency, but if a fundamental computational process is missing, you won't discover it through self-play. Actual comparison against human experts is required to diagnose such deficiencies.

The most direct form of comparison is competition. Competition measures skill using the same standards that humans use. You can also participate in post-mortem discussions that provide guidance about where to invest additional effort.

Maven's competitive games showed that Maven was championship caliber. They showed that I could stop worrying about things that I always believed were unimportant, but experts told me were huge. For example, was it important that Maven didn't vary its play as a function of the score? Was it important to consider the skill of the opponent? Was it important to *block* or *open* the board? Well, maybe it was important, but it was insignificant compared to Maven's skill in scoring points and keeping good tiles.

Other forms of comparison are indirect. For example, I compared Maven's moves against moves made by experts and against annotations written by experts. I published annotations "written" by Maven, to elicit feedback from experts. All of these things helped somehow, if only to provide reassurance to the author.

The endgame

I also learned about the importance of the endgame, which is the phase of the game where there are no tiles in the bag, and so the game becomes deterministic. Maven made serious endgame errors by failing to block a good spot for the opponent or failing to leave itself a way to play off all of its tiles.

Achieving good endgame play required that I scrap Maven's whole approach, because it is impossible to build a static evaluator that evaluates an endgame position using only one ply of lookahead. Clearly, the searching techniques of perfect-information games needed to be brought to bear, but the leading candidate (full-width alpha-beta) had serious shortcomings for this application.

For one thing, alpha-beta requires almost best-first move ordering for good search efficiency, whereas my move generator produces moves in order of rows of the board. The prospect of ordering moves after generating them was unattractive, because there are an average of 200 moves at the start of an endgame, and there could be many, many more if the side-to-move held two blanks. Also, move generation was comparatively slow (about 1 second on the hardware of the day), which limited us to about 120 nodes per search. Obviously, you can't search a tree whose branching factor is 200 at the root if you have only 120 nodes to work with. As if that weren't enough, there are vitally important endgames where the one side is "stuck with the Q" and cannot play out. In such cases, the best strategy may be to play out "one tile at a time." Such endgames can last 14 ply, with several hundred legal moves per ply, and the highest-scoring moves are almost always bad!

What was needed was a search algorithm that was naturally full-width, variable-depth, and appropriately selective (that is, able to distribute 120 nodes of search so as to explore a potentially huge space). Fortunately, I have read nearly every paper about search algorithms ever written, so Berliner's B* algorithm was familiar to me. The technique of applying B* is very interesting and novel, but alas, this is not the right forum for describing it, as it is highly domain-specific. To continue our topic, every game programmer needs to be familiar with the literature, because there are many general-purpose methods available. The programmer must also accurately judge the applicability of each method to his domain. Finally, general methods usually require domain-specific adaptation to reach their full potential.

Statistical lookahead

Finally, is developing one novel technique too much to ask of a game programmer? Actually, most successful game programmers have contributed a novel method. It seems that one cannot conquer a new game simply by applying previously known techniques. So I tentatively put forward that Maven was the first to use the technique of statistical lookahead for playing games.

Statistical lookahead is now recognized as a general method, having been applied (and independently discovered) by pioneers in games such as backgammon, bridge, and poker. The technique might be new to readers, so I will take a moment to describe it.

The idea is to evaluate moves by "playing them out" at high speed. The move with the highest average outcome is selected. During the process, you can gain speed by pruning moves that have proven to be inferior. This technique has many domain-specific details, such as the question of which alternatives are considered, how the game is played out, how to prune moves, how to model opponent's behavior, and so forth. Virtually any domain with randomness (such as backgammon) or hidden information (bridge) or both (Scrabble and poker) can benefit from statistical lookahead. I think this technique will produce a treasure-trove of research results (and practical results) because of its adaptability.

So, successful game AI results from combining many general methods. Maven would not be what it is without full-width search, evaluation functions, self-play, feedback, competition, indirect comparisons, knowledge engineering, perfect-information search techniques, and statistical lookahead. Plus a lot of luck.

Computers and language games

Michael L. Littman, Duke University

It's amazing that we can communicate at all. There are millions of words and phrases that can be uttered and understood, each with its own particular shade of meaning. New words constantly enter the lexicon, and old words continually evolve in their meanings, making the situation appear downright grim. Even so, language use rarely seems like hard work to us. In fact, to many people, language is one of the best toys around.

Language games draw their challenge and excitement from the richness and ambiguity of natural language. Acrostics, cryptograms, Jeopardy, and even riddles and puns are all forms of language games. Games like Scrabble, word search, or Boggle, although word-related, typically don't involve true language, because word meanings are not important. And it's having to make judgments about word meanings that separates language games from their purely logical counterparts.

In purely logical games, the rules defining winning positions and legal moves are quite clear. Not so for crossword puzzles, one of the most popular language games and the focus of this essay. Consider the crossword clue "The Hindenburg, e.g." (5 letters). Probably most of us would agree that BLIMP is a valid answer and ROGER isn't. But, what about LARGE or RIGID, or even MOVIE? While acceptable answers, they are far from ideal.

From an AI perspective, language games are an interesting challenge. The language component makes them different from board games and more closely related to applications such as text summarization or machine translation. But, like logical games, success in solving crosswords is crisply defined: how well does the system do in producing the right answer?

The crossword problem

A group of us at Duke University became interested in language games and decided to build a crossword-puzzle-solving program in the fall of 1998. We called the resulting system Proverb for "probabilistic cruciverbalist" because it uses probability theory to solve crossword puzzles.[1]

The lack of formal rules was the very first difficulty we had to face. Michael Garey and David Johnson, in their book on NP-completeness,[2] provide formal definitions for hundreds of computational problems, including crossword puzzles. In their version of the crossword problem, the puzzler receives a grid and a dictionary of legal words. A solution is an assignment of dictionary words to each slot in the puzzle, so that the across and down words fit together.

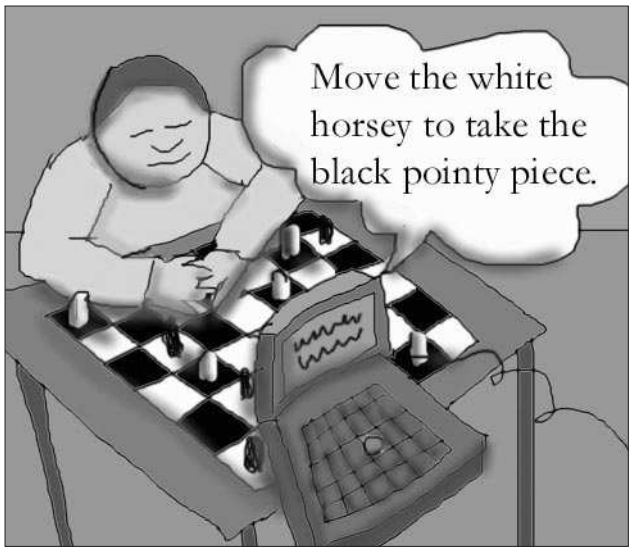
While this clearly captures some of what it means to solve a puzzle, it isn't a useful characterization of real crosswords. For one thing, there is no official dictionary of legal answers. For Proverb, we compiled a list of 2.1 million words and short phrases from a combination of online dictionaries, news wire articles, compendia of famous people's names, and other sources. Even so, this list only covers around 95% of answers on an average puzzle. For example, the 16 October 1999 *TV Guide* crossword contains 66 answers, three of which were not in Proverb's extended word list: SHOOTME, BRIDEOFCHUCKY, and PRINCESSBRIDE. A priori, it's hard to rule out *any* letter sequence as a possible crossword answer: XLNC has been clued as "Ambassador's title," (4 letters).

But, there's another, more significant reason that we need a different formalization of crossword puzzles: in the correct solution, the answers must relate to the clues. We can adapt the Garey and Johnson problem statement to reflect this by having a separate dictionary for each slot of the puzzle. This takes us a step closer to a practical formalization, but it misses two important facts:

- As in the Hindenburg example, some answers are just better than others. We ought to allow marginal answers, but there should be a preference for the better answers.
- Even if there were a perfect way of deciding which answers are okay and which aren't, it is unlikely we could write a program that could capture the distinction perfectly. The best we could expect a computer program to do would be to assign candidate answers confidence scores so that acceptable answers are generally ranked above unacceptable ones.

This line of thinking informed the formalization we chose for Proverb. The crossword puzzle problem divides into two pieces: candidate generation and grid filling. In candidate generation, Proverb analyzes each clue and generates a long list of candidate answers. It then assigns each candidate a prior probability indicating how likely it is that the candidate is an answer to the clue. If RED, HUE, DYE, and TAN are all equally good answers for "Color," (3 letters), then each would be assigned a prior probability of 0.25. However, in actual crossword puzzles, RED is about twice as common as any of the other answers to this clue, so it ought to be assigned a higher prior probability.

In grid filling, the probability-weighted candidate lists and the grid serve for generating a solution to the puzzle. Because candidate lists are generally long, there can be multiple solutions. We define the best solution to be one that maximizes the expected number of correctly answered clues. To define this formally, we assume that the probability of a given solution is proportional to the product of the prior probabilities of the answers that make up the solution.



An early chess program.

Candidate generation

Perfect candidate generation requires human-level knowledge.

Consider the clue "Deadly sin," (5 letters). Answering this requires background knowledge to know that four of the seven deadly sins have five letters: GREED, WRATH, SLOTH, and PRIDE. It requires lexical knowledge to know that ANGER is an appropriate alternate to WRATH. It requires syntactic knowledge to realize that LUSTS and SEVEN are related terms, but are not appropriate given the wording of the clue.

Some clues require an understanding of cause-and-effect or even phonetic relationships between words: "Result of bird pharyngitis," (12 letters): CROWINGPAINS. Still others only make sense if you are familiar with crossword conventions or current events: "Emulate Mia," (5 letters): ADOPT.

For these reasons, human-level proficiency in candidate generation seems an elusive goal. In designing Proverb, we turned to standard computer science and AI techniques for help with candidate generation. Proverb uses a set of databases, each with one or more ways of turning a clue into a query. Of the many databases it draws upon, the most important is the *cluedb*, a collection of clues and answers from previously published crossword puzzles. Depending on the difficulty of the puzzle being solved, from 30% to 60% of the clues are already present in the cluedb.

Because of the wide variety of ways in which clues can be worded, the cluedb contributes in several different ways to candidate generation. The simplest and most accurate is exact match; if the clue being attacked appears in the cluedb, the answer or answers in the cluedb are returned as a candidate with high prior probability. Proverb also queries the cluedb using word-overlap measures as pioneered in the information-retrieval community: the more words in common a clue has with one in the cluedb, the more likely it is to share its answer.

The cluedb is also used as a source of word-word associations. In the Hindenburg example, Proverb returns BLIMP because "Hindenburg" appears with "airship" in one clue, and "airship" appears with "blimp" in another clue. By building association chains of this kind, Proverb can generalize a bit beyond the specific set of clues it has to work with. Of course, this kind of free association leads to some odd connections. The answer ROGER also comes back because "Hindenburg" appears in a clue with "Ebert" (Hindenburg succeeded Friedrich Ebert as Reich president in 1925), which appears in a clue with "Roger" (because of movie critic Roger Ebert). (You can play with a version of the candidate generation algorithm online at www.oneacross.com.)

Grid filling

In some ways, Proverb's candidate generation is amazing: for "Broadcast," (10 letters), it instantly returns DISTRIBUTE, PASSAROUND, and PUBLICIZED. I doubt even the best human crossword solvers could do this. However, it also generates TELEVISION, TECHNICIAN, and JOURNALIST, among other 10-letter words. While there is a clear connection between these words and the clue, few people would consider them to be valid answers.

However, what Proverb lacks in answer precision, it makes up for in the grid-filling stage. Unlike humans, Proverb can simultaneously entertain thousands upon thousands of choices for how to fill in the grid. To generate a solution to the puzzle, Proverb first computes the posterior probability for each candidate. This is the probability that the candidate appears in a solution; it's a combination of how well the candidate answers the clue (its prior probability) and how well it fits with other answers in the grid.

Computing the posteriors exactly is a #P-complete problem; it's like counting the number of answers to a Boolean satisfiability problem—probably worse than NP-complete. For Proverb, we designed a tree-based approximation algorithm; we later learned that the same approach is used in decoding transmissions from deep-space probes. Proverb then uses A* search to find a solution that maximizes the sum of the posteriors; this is the solution that maximizes the expected number of words correct.

Proverb's performance on real puzzles is impressive. In about 15 minutes of wall-clock time per puzzle, Proverb generates solutions that average 95% words correct on a testset of 370 puzzles from *The New York Times*, *USA Today*, *The LA Times*, *TV Guide*, and other sources. Proverb was also run on a set of seven puzzles used in the 1999 American Crossword Puzzle Tournament and scored around the middle of the pack of the 250 human competitors.

Future challenges

In spite of its strong performance on real crosswords, Proverb is still a far cry from black-belt level. Expert human solvers can finish a moderately challenging 15×15 puzzle in four or five minutes with no errors. While increasing Proverb's speed is not difficult, brining its accuracy up to championship levels will require a bit more work.

One way for Proverb to improve performance would be to read between the lines more. For example, encountering the clue "Behold to Brutus," (4 letters) with answer ECCE should tell Proverb that "ecce" means "behold" in Latin. Armed with this information, Proverb would be prepared to answer "Latin behold," (4 letters). Through a more syntactic analysis of the cluedb and other text sources, a structured knowledge base could be constructed for use in solving novel clues with high accuracy. This knowledge base would have natural applications beyond crossword puzzles.

Proverb already supports a simplified form of this reasoning. In particular, it looks through the cluedb for clues with the same answer and constructs transformation rules for turning one clue into another, preserving meaning. Using this, Proverb finds some simple grammar-type rules such as: "X of Egypt" is equivalent to "Egypt's X" for all X. It also finds some crossword-specific transformation rules such as: "Nice X" is equivalent to "X in French" for all X.

But, of course, our work on Proverb just scratches the surface of the sort of techniques that can help computers tackle language games. And, different techniques will be most useful in different games.

Even so, expert competence in any language game requires mastery across all levels of linguistic knowledge, from pronunciation and spelling, to syntax, meaning, and world knowledge. Thus, while a grandmaster chess program can be a narrowly focused savant, a competition-class crossword program would necessarily possess a broad set of skills. This makes the program itself much more of a challenge to build, but probably also a whole lot more interesting to chat with after a match. ❐

References

1. G.A. Keim et al., "Proverb: The Probabilistic Cruciverbalist," *Proc. 16th Nat'l Conf. AI*, AAAI Press, Menlo Park, Calif., 1999, pp. 710–717.
2. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, Miller Freeman, San Francisco, 1979.