WIREs COGNITIVE SCIENCE

Article type: Overview

Article title: Game Playing

Christopher D. Rosin
Parity Computing, Inc.
San Diego, California, USA
c.rosin@paritycomputing.com

Abstract

Game playing has been a core domain of artificial intelligence research since the beginnings of the field. Game playing provides clearly defined arenas within which computational approaches can be readily compared to human expertise through head-to-head competition and other benchmarks. Game playing research has identified several simple core algorithms that provide successful foundations, with development focused on the challenges of defeating human experts in specific games. Key developments include minimax search in chess, machine learning from self-play in backgammon, and Monte Carlo tree search in Go. These approaches have generalized successfully to additional games. While computers have surpassed human expertise in a wide variety of games, open challenges remain and research focuses on identifying and developing new successful algorithmic foundations.

This is the pre-peer reviewed version of the following article:

Rosin, C. D. (2014), Game playing. WIREs Cogn Sci, 5: 193–205. doi: 10.1002/wcs.1278

which has been published in final form at:

http://onlinelibrary.wiley.com/doi/10.1002/wcs.1278/pdf

Game playing has been a core domain of artificial intelligence research since the beginnings of the field[1,2]. While some domains of artificial intelligence research are open-ended and require substantial real-world knowledge (e.g. natural language understanding), game playing provides clearly defined arenas within which computational approaches can be readily compared to human expertise through head-to-head competition and other benchmarks. The competitive aspect of game playing also helps motivate progress[3]. Game playing research has made steady progress against human expertise.

Game-playing research tends to be motivated by the challenges presented by competing with human experts in specific games. The open research questions are focused on identifying and developing techniques that can advance the state of the art in playing strength in games where human experts still dominate. Games in which available techniques already dominate human experts tend to get less attention, and there is less attention on researching new computational approaches for their own sake if it seems unlikely they could eventually advance the state of the art in playing strength.

One direct approach to creating game-playing software is to work with expert human players, attempting to directly encode the heuristics, patterns, and reasoning processes they use. *Expert systems*[4] are representative of this approach. Early attempts at particular games often reflect this approach[5,6]. Improving the strength of such an approach often yields increasing complexity, which can increase the difficulty of further improvement. An alternative approach is to attempt to find simple fundamental algorithms that provide some computational leverage against the game, with an ability to improve performance given faster computing hardware, and providing a solid well-founded basis for further enhancement in software. When a successful simple foundation is discovered, it can enable rapid progress in a game, and it tends to displace earlier complicated approaches. Such displacement has been observed in backgammon and Go (described below), and a key contribution of game playing research is a powerful demonstration of the potential and impact that these fundamental algorithms can have. Three such foundational algorithms are described below, in the context of games that helped motivate their development. Applications to additional games are also summarized, concluding with open areas of research. Basic game-playing notions are defined along the way.

Minimax Search in Chess

Chess has been a focus of artificial intelligence research since the beginnings of the field. Basic techniques for minimax search in chess were described early on by Norbert Winer in 1948 and Claude Shannon in 1950.[1,2] Extensive developments in chess programs based on minimax search yielded steady increases in playing strength (Figure 1), leading to the defeat of world champion Garry Kasparov by the special-purpose supercomputer Deep Blue in 1997[7], and the defeat of world champion Vladimir Kramnik by the program Deep Fritz running on standard commodity computing hardware in 2006.[8]

Chess is a deterministic two-player game with alternating turns, and with both players having complete information about the state of the game at all time. In this setting, the game can be described by a *game tree* (or *search tree*) as in Figure 2. A node in the game tree corresponds to a *position* (or *state*): a single specific configuration of pieces on the

board, with one of the two players to move. Edges extending outward from the node correspond to legal moves, and lead to child nodes corresponding to the resulting positions with the opposing player to move. The number of legal moves is called the *branching factor* of the game tree.

In principle, the game tree can be extended all the way until the end of the game (ending in victory for one player). If the tree is started from a position that is already near the end of the game, this is feasible in practice as well (Figure 2). In this case, the winner of the chosen starting position can be determined exactly by the *minimax search* procedure (Algorithm 1).[9] For a node in which all of the player's moves lead to final end-of-game positions, the node is a victory for the player if and only if at least one of the legal move edges leads to a victory. This procedure can then be extended recursively up the tree to label each node as a victory for one player or the other. The name "minimax" comes from assuming that one player is maximizing the value (e.g. a win for the first player is valued at +1) and the other is minimizing (e.g. a win for the second player is valued at 0); these reflect the operations performed over the children of a node to determine the node's value, alternating between "min" and "max" at alternating levels of the tree.

Algorithm 1: Minimax Search

MINIMAX(position P):

If P is final // *final because it is end-of-game, or cut-off has been reached*
Then Return (VALUE(P)) // *higher values indicate better results for first player*
Else
Let S be {the set of positions resulting from legal moves from P}
If first player is the player to move in P
Then Return (**Max** over P' in S of MINIMAX(P'))
Else Return (**Min** over P' in S of MINIMAX(P'))

Considering the entire game tree for a game, starting from the game's initial position and extending all the way to the end of the game, the number of distinct legal positions is referred to as the *state space size*.[10] It is a measure of the complexity of searching the entire game tree. In chess, the state space size is estimated to be at most 10^{53} (Table 3). While modern chess programs can search millions of positions per second, searching the full game tree is far beyond current capabilities (and may never be feasible).[11]

So, for positions far from the end of the game, it is not computationally feasible in a complex game like chess to apply minimax search all the way to the end. In this case, minimax search needs to be cut off before reaching an end-of-game position. A simple approach is to perform a fixed-depth minimax search which applies a cut-off after a fixed constant number of moves from the initial position. At this cut-off point, the game is incomplete yet the position needs to be evaluated according to the degree to which it is

favorable for one player or the other; this determines the VALUE function in Algorithm 1. The function used to do this is called a heuristic static *evaluation function*[9,12], which maps details of the configuration of pieces on the board to a number that scores the value of the position for one of the players. It is called "heuristic" because, in a complex game like chess, it is rarely possible to do this in any provably exact way. And it is called “static” because it is primarily computed using features of the position at hand rather than using further searching.

For chess, a starting point for an evaluation function is the commonly understood piece values (Table 1). Using an evaluation function based on these, a minimax value of +5 would indicate that, within the limited depth of minimax search, white is able to obtain an advantage at least as large as the value of a rook. Of course, this evaluation function ignores other critical positional factors, and in practice chess programs use a wide variety of carefully tuned features in their evaluation functions.[7] Even so, these evaluation functions are approximate and may barely hint at the real objective, which is a player's ability to force an eventual checkmate (perhaps many moves away) with optimal play.

The key reason for the effectiveness of minimax search in chess is that deeper minimax boosts the quality of evaluation and the resulting quality of play, so that even a relatively crude evaluation function can lead to reasonably accurate assessments and good-quality play when coupled with sufficiently deep minimax search. While the conditions for success or failure of this effect have been studied in simplified models[12,13,14,15], in a complex game like chess there is no assurance that it will always be true, and the practical success of this approach needs to be tested empirically. And it has been successful. As available computing hardware has increased in speed, larger searches have led to increasing strength of minimax search in chess (Table 1), even using the same software.[16,17]

Software enhancements have been critical as well though. Beyond improved heuristic evaluation functions, important improvements to the basic minimax search method in chess include:

- *Quiescence search*[2,13,18]*:* do not cut off search in the midst of an exchange of pieces or other rapid change in evaluation. This is an important modification to fixed-depth cut-offs.
- *Alpha-beta pruning*[19]: stop searching early down lines of play that will never be visited because a better alternative is known to be available. This is an exact method that doesn’t change the results of minimax search, and under best-case scenarios can enable minimax search to go twice as deep in the same amount of time.
- Heuristic enhancements such as the *killer heuristic*[20]*, null move heuristic*[18], and *iterative deepening*[21] improve the effectiveness of alpha-beta search, in part by reordering search to prioritize favorable moves.

The combination of software improvements and increasingly powerful hardware led to steady improvements in computer chess over decades of development, culminating in Deep Blue's victory. Since then, software has continued to improve, enabling continued improvements using conventional computing hardware (Figure 1). Current ratings estimates put software chess-playing ability well beyond that of human champions.

Minimax Search in Other Games

Minimax search has been the most widely successful method in computer game playing.

- Shogi, a chess-like game from Japan, has added complexity from the return of captured pieces to the board. But ongoing improvement in computer Shogi[22] led to the defeat of professional players by minimax-based Shogi programs in a 5-game match in 2013.[23]
- Deep minimax searching in Othello combined with carefully tuned evaluation functions has resulted in programs such as Logistello[24] that have consistently defeated human experts in head-to-head play.
- Checkers is somewhat less complex than chess, and minimax search based Chinook defeated the world champion[25] prior to Deep Blue's victory in chess. Ongoing advances in search techniques and computing speed enabled a full minimax-based solution to checkers, all the way to the end of the game, obviating any further need for heuristic evaluation.[26]
- Despite having a vast state space (Table 3), the game of Gomoku (5 in a row) has also been solved by minimax search using clever pruning techniques.[10]

Machine Learning from Self-Play in Backgammon

The game of backgammon takes place on an effectively one-dimensional board in which players move their pieces in opposite directions, racing to reach their "home" area (Figure 3). A player's turn consists of rolling two dice and then moving pieces by the amounts indicated by the dice. An isolated piece that is alone at a location on the board (a “blot”) can be hit by the opposing player and returned to the opponent's home; this aspect necessitates strategic decisions about offense versus defense versus simply winning the race. While the dice are a significant factor in determining the winner of a single game, tournaments usually involve longer matches in which the more skilled player becomes more likely to win. A doubling cube can be used to increase the value of a single game within the match, and expert play of the doubling cube requires a player to have a good estimate of the probability of winning the game.

The dice rolls add chance nodes to the game tree (Figure 4). Minimax search is still possible in principle, but requires averaging at each chance node over all 21 distinct combinations of two dice. This greatly expands the amount of searching required, and minimax search in backgammon is commonly limited to 2-3 ply.[27] With the role of minimax search so limited, the main requirement is for an accurate evaluation function, and the machine learning approach described here yields accurate evaluations even with no minimax search.

In the 1970's, painstakingly handcrafted evaluation functions achieved an intermediate level of play in the BKG program.[5,28] But this approach has not progressed much further; there are vast numbers of plausibly relevant features and ways to use combinations of them, and even expert backgammon players face uncertainty in their own evaluations.[27]

The Neurogammon program[29] by Gerald Tesauro used a neural network (Figure 5), which weights input features and combinations of features to generate an output. The weights of the neural network are adjusted through a gradient descent training process, to match a given target on each sample input provided. With Neurogammon, the inputs were positions from human expert games, and the targets reflected moves chosen by experts. Neurogammon learned to play moves like those of the experts. This was a moderately successful approach that reached a strong intermediate level[27]; Neurogammon won the 1989 International Computer Olympiad backgammon championship.[29]

But Neurogammon was limited to approximating the moves of fallible experts using available expert games, and had additional limitations such as an inability to learn the expected value of positions.[30] TD-Gammon[27,31,32] is a successor to Neurogammon, which used similar neural networks but a different approach to learning. TD-Gammon's neural network would play games against itself, relying on the dice to sample a wide variety of positions during self-play. The output of the neural network predicts the expected value of the position (the probability of winning), and moves with the highest predicted value are chosen. During self-play, Temporal Difference (TD) learning was used to train the neural network. TD learning is based on the observation that, when predicting expected outcome, later predictions are more accurate because they have more information about how the game evolves and are closer to the final value at the end of the game – so later predictions can be used as targets for learning earlier predictions. Algorithm 2 shows the TD learning algorithm in a configuration used by TD-Gammon[27]. The final "Update" of W in Algorithm 2 improves the alignment of the neural network's predicted Y_f with the final VALUE of the game (a win or loss); this provides feedback from the ground truth of actual win or loss. The "Update" loop before that trains each position's neural network evaluation to better align it with the neural network evaluation of the next position in the game. Temporal Difference learning has been shown to converge to optimality in simple scenarios[33], although TD-Gammon is a complex application that requires heuristic approximations.

Algorithm 2: TD(0) Temporal Difference Learning Update with one Self-Play Game. Below, C is a small constant learning late.

TDSelfPlay(neural network weights W):

Self-play a complete game G using neural network with weights W to select moves
Let Z = VALUE(final end-of-game position of G)
Let $X_1 \ldots X_f$ be the sequence of feature vectors representing each position in G
Let $Y_1 \ldots Y_f$ be the sequence of neural network outputs evaluating each position in G
For t = 1 to f-1:
Update W by adding C * ($Y_{t+1} - Y_t$) * gradient of Y_t with respect to weights
Update W by adding C * ($Z - Y_f$) * gradient of Y_f with respect to weights

Despite lacking access to the human expert training data used by Neurogammon, TD-Gammon's self-play approach achieved superior results.[27] Starting from scratch using only its own self-play training data, over the course of hundreds of thousands of games TD-Gammon acquires advanced backgammon knowledge.[27,31,32] In the few head-to-head matchups that have occurred, TD-Gammon and successors using its techniques have been even with championship-level backgammon players, and expert players acknowledge the superior capabilities of such programs.[27,34] Analysis using TD-Gammon and its successors has become standard in the backgammon-playing community, and has informed the way experts now play.[34] Such analysis allows the quality of all moves in a match to be quantified, and using this measure the quality of TD-Gammon's play was superior to that of its human expert opponents even relatively early in TD-Gammon's development (Table 2).

Machine Learning from Self-Play in Other Games

- TD-Gammon had a strong influence on the development of reinforcement learning and applications outside of games[27], and work on backgammon and reinforcement learning influenced the development of Monte Carlo tree search (see section below). Temporal Difference self-play learning has also been applied with modest success to other games including Go[35] and chess[36]. But outside of backgammon, self-play and Temporal Difference learning haven't had impact comparable to that in backgammon. In studying reasons for the particular success of self-play in backgammon[27,37], the strong random element seems to be an important factor: the dice will force broad exploration of the set of possible positions, and even at a beginning level of self-play with many mistakes there will be occasional observations of more desirable positions that would be sought in higher-level play.

- TD-Gammon was influenced directly by early work on self-play learning in checkers.[27,38] Self-play learning achieved the best early results in checkers, before being eventually superseded by minimax methods that achieved world champion levels of play.[25]

Monte Carlo Tree Search in Game of Go

Go is an ancient Asian game in which two players (black and white) alternate placing stones on the intersections of a 19x19 grid (see Figure 6). A block of adjacent stones can be captured if the opponent occupies all its liberties (adjacent empty intersections). The goal is to end the game scoring more territory (stones plus surrounded area) than the opponent. Competitive Go tournaments are held regularly, with a substantial community of highly skilled professional players.

Because most unoccupied points are legal moves, the branching factor is much higher than chess -- over 300 in the opening phase, and most of these moves are difficult to definitively rule out. The overall state space size is far larger than that in chess (Table 3). Minimax search becomes infeasible beyond several ply, which is very shallow compared to the full depth of the tree in this lengthy game (often well over 100 moves long).

Another daunting challenge has been the difficulty of developing good static evaluation functions. The value of a move can depend on its absolute position and nearby stones, but it is also sensitive to details of the position across significant areas of the board. Specific patterns in Go such as “ladders” occur commonly and can extend a stone's impact all the way across the board. Labor-intensive efforts in computer Go have developed special-purpose focused tactical searches, features based on large dictionaries of important patterns, and heuristics for quantifying influence in areas of the board.[6,30] Such methods become complex, with many interacting parts, and progress in “classical” Go programs with this labor-intensive approach has been slow.

Unlike the situation in backgammon, where suitable features can be found for machine learning of strong evaluation functions, it is unclear what representation could support strong evaluation functions in Go. Attempts at applying Temporal Difference learning to Go have made some progress, but are limited by the representational power of the simple input features to the neural network.[35] Such issues have also limited results from efforts to apply machine learning to acquire knowledge from expert-level game records.[39,40] The value of arrangements of stones depends on details that are difficult to establish without search, yet search is difficult without guidance from evaluation. Machine learning of strong static evaluation functions in Go remains an unsolved problem.

The strength of modern Go programs is instead based on Monte Carlo tree search (see Figure 7).[41,42] Rather than using a static evaluation function to evaluate a position P, instead many games are randomly played out starting from P and proceeding all the way to the end of the game (where evaluation just requires identifying the winner of the game). The evaluation is then the average win rate (Algorithm 3). The move with the highest average win rate is selected for play. While the strategy used in the random playouts affects the quality of the results, a baseline is to make completely random legal moves (though avoiding some simple suicidal moves). The simplicity of Monte Carlo tree search stands in sharp contrast to the complex ad hoc programs that preceded it, and has radically changed the approach to computer Go.

Algorithm 3: Basic Monte Carlo Evaluation

```
MONTE-CARLO-VALUE(position P):
    Repeat N times:
        Let P' be a copy of P
        Until P' is an end-of-game position:
            Update P' with a random legal move
        Add VALUE(P') to the running total T
    Return (T/N)    // Average value
```

While more playouts to collect better statistics improves results up to a point, once the average win rate converges there is little to be gained by simply collecting more statistics.[43] One could attempt to combine Algorithm 1 for minimax search with Algorithm 3 to estimate VALUE. Some nodes, however, will show clearly poor results even from a small number of Monte Carlo playouts, so it makes sense to allocate playout resources where they will be most informative. The UCT algorithm, a generalization of Algorithm 3, smoothly balances deeper search in promising nodes with further exploration of uncertain nodes.[44,45,46] It is based on the UCB algorithm for effective allocation of trials in a multi-armed bandit setting; the allocation balances "exploitation" of moves that have returned good results so far with "exploration" of moves that have been little tested.[47] UCT has been proven to converge to optimal play in the limit[44], but with limited resources in a complex game like Go its use is heuristic and must be tested experimentally. And UCT has been empirically successful in Go and other games. It has become the standard baseline Monte Carlo tree search algorithm.

Beyond UCT, researchers have increased the strength of Monte Carlo tree search for Go in a variety of ways, particularly in two categories:

- *Improved playout policies*: Patterns learned from expert play using a simple representation can provide substantial improvement over random playouts.[40] But roughly equivalent performance can be obtained by biasing moves using a set of 3x3 patterns carefully hand-selected to improve resulting Monte Carlo playing strength.[46] Further improvement is challenging; biasing playouts too strongly may lose too much of the random sampling Monte Carlo requires, and more generally it appears that simply choosing stronger moves during playouts doesn't necessarily yield stronger play from Monte Carlo tree search.[48] It may be feasible to develop new machine learning approaches that improve policies in ways appropriate for Monte Carlo tree search.[49]

- *Good initial estimates of move values at each position in the Monte Carlo search tree*. UCT collects information slowly, with little ability to choose good moves at a node until many playouts have been averaged. Even the highly approximate simple static evaluation functions available in Go can improve Monte Carlo strength when used to initialize estimates at a node.[48] And various methods share information from playouts across multiple moves and multiple nodes in the search tree, increasing accuracy of evaluations more quickly.[48,50]

Prior to the ascendance of Monte Carlo techniques around 2007, "classical" programs based on a complex mix of heuristics were approximately rated at intermediate amateur levels (around 5-15 kyu)[30,50], but with worsening performance as human opponents learn to exploit their weaknesses.[30] Since then, progress with Monte Carlo techniques has been rapid (Figure 8). The Monte Carlo program MoGo scored the first computer Go victory against a human professional in 2008 at a maximum handicap of 9 stones.[50] In 2012-2013, there were multiple victories against top-ranked professionals at a handicap of 4 stones.[51] But substantial progress beyond this is required to defeat a professional Go player in an even game (0 handicap stones).

Monte Carlo in Other Games

- One of the original proposals for Monte Carlo search was to improve the strength of TD-Gammon, using its learned neural network evaluation function to bias move choice during playouts.[52] This is closely related to the backgammon playing community's longstanding practice of rolling out many games by hand from a given starting position to estimate the value of that position. Such rollouts using strong neural network programs like TD-Gammon have since become a standard analysis tool in the backgammon community.[34] Furthermore, some of the research on Monte Carlo tree search for Go originates in the reinforcement learning community that developed TD learning; it is possible to view Monte Carlo tree search as a kind of reinforcement learning that takes place in real time while analyzing a position to choose a move.[48,53]

- Following success in Go, Monte Carlo has been established as the leading method in other games including Hex[54] and Lines of Action[55], and has been used to close in on human expert levels of play in Havannah[56]. Monte Carlo techniques have surpassed human ability in challenging solitaire games, including Thoughtful Solitaire[57] and Morpion Solitaire[58], in the latter case overcoming a human record that had stood for over 30 years. Monte Carlo tree search is also the leading method in the "General Game Playing" challenge, in which new games are formulated at the start of a competition and programs have little time to analyze their rules before competing.[59]

- Some imperfect information games such as card games have stochastically chosen information hidden in opposing player hands. Monte Carlo techniques play an important role in sampling from a distribution over this hidden information, for example in bridge[60] and Scrabble[61]. While developed independently of Monte Carlo tree search, these techniques share the notion of simulating randomly chosen alternative states for the game.

Research Frontiers

Games in which humans are still superior to software tend to attract the most research attention. Some of these may require the development of new approaches to overcome current obstacles, as happened with the application of Temporal Difference learning to backgammon.

- While Monte Carlo techniques have progressed more rapidly in Go than most researchers anticipated[50,53], Go-playing programs are still far from consistently defeating champion-level professional players. It is quite possible that new fundamental advances will be required before this goal is achieved.

- Poker has attracted substantial research interest.[62] Complexity comes from hidden opponent cards, and the need for randomized strategies that bluff appropriately during betting. A favored technique is state abstraction, in which game states that differ by only very minor details are grouped into the same bucket, to yield a simplified game that can be nearly exactly solved by techniques such as counterfactual regret minimization.[63] The program Polaris used these techniques in the game of Limit Hold'Em to defeat a panel of human experts in a significant tournament with 6000 hands of play.[64] But the most popular game of tournament poker, No Limit Hold'em, has a much larger state space due to unrestricted betting (Table 3), and new approaches may be needed to address it. And additional challenges arise in settings with more than two players, where it becomes particularly important to maximize gains by identifying and exploiting deficiencies in opposing strategies.

- In games such as poker and bridge, important information about the opponent's state is hidden from the player. But this information is primarily determined stochastically from a known probability distribution at the start of the game, making it feasible to reason about it and sample from it as in the Monte Carlo techniques used in bridge.

  But in a different class of partially observable games, the opponent has a great degree of control over the hidden information, leading to a massive number of possible states of the opposing position with information about these only gradually revealed. One game in this class in Kriegspiel[9,65], a form of chess in which the location of opposing pieces is only revealed when they interact with the player's. Another such game is Stratego[66], in which the locations of all pieces are selected by the opponent and only revealed to the player when there is a potential capture. In these cases, it is entirely unclear what distribution the opponent used to establish the opposing position. Assumptions may be unwarranted, but it is unclear how to apply search techniques with such massive uncertainty unless many assumptions are made about the opponent's strategy. Human players do not find these issues too daunting; Stratego is sometimes considered a children's game. But effective computer play of such games may require a new approach.

- Arimaa is a game specifically designed to be difficult for computers to match human ability. The game is relatively new, and as of 2013 while programs have made progress on the game,[67] human expertise has been getting stronger as well and remains ahead.

- The General Game Playing competitions[59] have seen substantial progress, and new techniques may be developed here. One of the questions this line of work raises is whether a single approach may be feasible across most games. The three techniques described above (minimax search, machine learning from self-play, and Monte Carlo tree search) have

developed in different directions and it is an open question whether it would be fruitful to attempt to unite their benefits.

- Some domains in AI have benefitted from a game-playing approach, even though the game strategy component may be only a small part of the challenge. Watson's victory over human champions in Jeopardy[68] represented a significant step for natural language processing and question answering. The RoboCup robotic soccer competitions are helping to drive robotics research and have seen steady progress.[69] Videogames provide a broad set of challenges for perception and learning in a controlled environment[70], with some such games providing unique game-playing challenges like massive numbers of available moves each turn[71].

Conclusion

Artificial intelligence research has yielded steady progress in game playing, reaching the level of world champion playing strength in a wide variety of games. This progress has been built on top of simple core algorithms. These algorithms include minimax search, machine learning from self-play, and Monte Carlo tree search as described here. New algorithms are likely to be required in those games where human experts still hold the upper hand. Game playing provides fertile ground for developing new computational approaches and benchmarking them against human expertise.

References

1. Wiener N. Cybernetics or Control and Communication in the Animal and the Machine. John Wiley & Sons, Inc., 1948.
2. Shannon C. Programming a computer for playing chess. Philosophical Magazine 1950, 41:256-275.
3. Schaeffer J. One Jump Ahead: Challenging Human Supremacy in Checkers. Springer, 1997.
4. Michie D. Introductory Readings in Expert Systems. Gordon and Breach, 1982.
5. Berliner H. BKG: A program that plays backgammon. Carnegie Mellon University Computer Science Department Paper 2375, 1977.
6. Müller M. Computer Go. Artificial Intelligence 2002, 134:145-179.
7. Campbell M, Hoane Jr. AJ, Feng-Hsiung H. Deep Blue. Artificial Intelligence 2002, 134: 57-83.
8. Newborn M. Beyond Deep Blue. Springer, 2011.
9. Russell S, Norvig P. Artificial Intelligence: A Modern Approach (3rd Edition). Prentice Hall, 2009.
10. Allis V. Searching for Solutions in Games and Artificial Intelligence. Ph.D. Thesis, University of Lumburg, Maastricht, The Netherlands, 1994.
11. Sreedhar S. Checkers, Solved. IEEE Spectrum, July 1, 2007, http://spectrum.ieee.org/computing/software/checkers-solved

12. Pearl J. Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley, 1984.
13. Scheucher A, Kaindl H. The reason for the benefits of minimax search. Proceedings of the 11th International Joint Conference on Artificial Intelligence, 1989, p. 322-327.
14. Nau DS. Decision quality as a function of search depth on game trees. JACM 1983, 4: 687-708.
15. Pearl J. On the nature of pathology in game searching. Artificial Intelligence 1983, 20: 427-453.
16. Thompson K. Computer chess strength. Advances in Computer Chess 1982, 3: 55-56.
17. Heinz EA. New self-play results in computer chess. Computers and Games, Lecture Notes in Computer Science 2001, 2063:262-276.
18. Beal DF. A generalised quiescence search algorithm. Artificial Intelligence 1990, 43: 85-98.
19. Knuth DE, Moore RW. An analysis of alpha-beta pruning. Artificial Intelligence 1975, 6:293-326.
20. Akl SG, Newborn MM. The principal continuation and the killer heuristic. ACM Annual Conference 1977, 466-473.
21. Korf RE. Depth-first iterative-deepening. Artificial Intelligence 1985, 27:97-109.
22. Iida H, Sakuta M, Rollason J. Computer Shogi. Artificial Intelligence 2002, 134:121-144.
23. Pro shogi players defeated by computer programs. The Japan Times News. April 22, 2013.
24. Buro M. The evolution of strong Othello programs. Entertainment Computing, Springer, 2003, pp. 81-88.
25. Schaeffer J, Lake R, Lu P, Bryant M. CHINOOK the world man-machine checkers champion. AI Magazine, 1996, 17:21-29.
26. Schaeffer J et al. Checkers is solved. Science 2007, 317:1518-1522.
27. Tesauro G. Programming backgammon using self-teaching neural nets. Artificial Intelligence 2002, 134:181-199.
28. Berliner HJ. Backgammon computer program beats world champion. Artificial Intelligence 1980, 14:205-220.
29. Tesauro G, Neurogammon wins computer olympiad. Neural Computation 1989, 1:321-323.
30. Schaeffer J, van den Herik HJ. Chips Challenging Champions: Games, Computers, and Artificial Intelligence, North Holland, 2002.
31. Tesauro G. TD-Gammon, a self-teaching backgammon program, achieves master-level play. Neural Computation 1994, 6:215-219.
32. Tesauro G. Temporal difference learning and TD-Gammon. Communications of the ACM 1995, 38:58-68.
33. Sutton RS. Learning to predict by the methods of temporal differences. Machine Learning 1988, 3:9-44.
34. Woolsey K. Computers and rollouts. GammOnLine, January 2000. http://www.bkgm.com/articles/GOL/Jan00/roll.htm
35. Schraudolph NN, Dayan P, Sejnowski TJ. Temporal difference learning of position evaluation in the game of Go. Advances in Neural Information Processing 6, Morgan Kaufmann, 1994.
36. Baxter J, Tridgell A, Weaver L. Reinforcement learning and chess. Machines that Learn to Play Games. Nova Science Publishers, 2001, pp. 91-116.
37. Pollack JB, Blair AD. Why did TD-Gammon work? Advances in Neural Information Processing 9, 1996, pp. 10-16.

38. Samuel AL. Some studies in machine learning using the game of checkers. IBM Journal of Research and Development 1959, 3:210-229.
39. Stern D, Herbrich R, Graepel T. Bayesian pattern ranking for move prediction in the game of Go. Proc. 23rd International Conference on Machine Learning 2006, pp. 873-880.
40. Coulom R. Computing Elo ratings of move patterns in the game of Go. Computer Games Workshop 2007, Amsterdam.
41. Bouzy B, Cazenave T. Computer Go: An AI oriented survey. Artificial Intelligence 2001, 132:39-103.
42. Bouzy B, Helmsetter B. Monte-Carlo Go developments. Advances in Computer Games, Springer, 2004, 159-174.
43. Yoshimoto H et al. Monte Carlo Go has a way to go. AAAI 2006, pp. 1070-1075.
44. Kocsis L, Szepesvári C. Bandit based Monte-Carlo planning. ECML 2006, Springer, pp. 282-293.
45. Gelly S, Wang Y. Exploration exploitation in Go: UCT for Monte-Carlo Go. On-line Trading of Exploration and Exploitation, 2006, Whistler, BC, Canada.
46. Gelly S, Wang Y, Munos R, Teytaud O. Modification of UCT with patterns in Monte-Carlo Go. INRIA No. RR-6062, 2006. http://hal.inria.fr/inria-00117266/fr/
47. Auer P, Cesa-Bianchi N, Fischer, P. Finite-time analysis of the multiarmed bandit problem. Machine Learning 2002, 47:235-256.
48. Gelly S, Silver D. Combining online and offline knowledge in UCT. ICML 2007, Corvallis, Oregon, USA.
49. Silver D, Tesauro G. Monte-Carlo simulation balancing. ICML 2009, Montreal, Canada.
50. Gelly S, Kocsis L, Schoenauer M, Sebag M, Silver D, Szepesvári C, Teytaud O. The grand challenge of computer Go: Monte Carlo tree search and extensions. Communications of the ACM 2012, 55:106-113.
51. Human-Computer Go Challenges. http://www.computer-go.info/h-c/ accessed June 30, 2013.
52. Tesauro G, Galperin GR. On-line policy improvement using Monte-Carlo search. NIPS 1996, 1068-1074.
53. Littman M. A new way to search game trees: technical perspective. Communications of the ACM 2012, 55:105.
54. Arneson B, Hayward R, Henderson P. MoHex wins Hex tournament. ICGA Journal 2010, 33:181-186.
55. Winands MHM, Bjornsson Y, Saito JT. Monte Carlo Tree Search in Lines of Action. IEEE TCIAIG 2010, 2:239-250.
56. Havannah Challenge 2012 games. MindSports. http://www.mindsports.nl/index.php/arena/havannah/644
57. Yan X, Diaconis P, Rusmevichientong P, Van Roy B. Solitaire: Man Versus Machine. NIPS 2005.
58. Rosin CD. Nested Rollout Policy Adaptation for Monte Carlo Tree Search. Proceedings of IJCAI 2011, pp. 649-654.
59. Bjornsson Y, Finnsson H. CadiaPlayer: A simulation-based General Game Player. IEEE TCIAIG 2009, 1:1-12.
60. Ginsberg ML. GIB: Imperfect information in a computationally challenging game. J. Artificial Intelligence Research 2001, 14:303-358.

61. Sheppard B. World-championship-caliber Scrabble. Artificial Intelligence 2002, 134:241-275.
62. Rubin J, Watson I. Computer poker: A review. Artificial Intelligence 2011, 175:958-987.
63. Zinkevich M, Johanson M, Bowling M, Piccione C. Regret minimization in games with incomplete information. NIPS 2007.
64. Bowling M et al. A demonstration of the Polaris poker system. Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems 2009, pp. 1391-1392.
65. Ciancarini P, Favini GP. Monte Carlo tree search in Kriegspiel. Artificial Intelligence 2010, 174:670-684.
66. Schadd M, Satz I. The 2nd Stratego Computer World Championship. ICGA Journal 2009, 31:251-252.
67. Fotland D. Building a world-champion Arimaa program. Computers and Games, Lecture Notes in Computer Science 2006, 3846:175-186.
68. Ferucci D et al. Building Watson: An overview of the DeepQA project. AI Magazine 2010, 31:59-79.
69. Chen X, Stone P, Sucar LE, van der Zant T (eds). RoboCup 2012: Robot Soccer World Cup XVI, Lecture Notes in Computer Science. 2013.
70. Bellemare MG, Naddaf Y, Veness J, Bowling M. The Arcade Learning Environment. J. Artificial Intelligence Res. 2012, 47:253-279.
71. Branavan SRK, Silver D, Barzilay R. Non-linear Monte-Carlo search in Civilization II. Proc. IJCAI 2011, pp. 2404-2410.
72. Arts AFC. Competitive play in Stratego. M.S. Thesis, Maastricht University, 2010.
73. Cox C-J. Analysis and implementation of the game Arimaa. M.S Thesis, Maastricht University, 2006.
74. Johanson M. Measuring the size of large no-limit poker games. 2013, arXiv:1302.7008
75. Russell S, Norvig P. Artificial Intelligence: A Modern Approach (1st Edition). Prentice Hall, 1995.
76. KGSBotRatings. Accessed June 30, 2013. http://senseis.xmp.net/?KGSBotRatings
77. KGS Robots Page. Accessed June 30, 2013. http://senseis.xmp.net/?KGSBots
78. Go News and Sensations, An Exclusive Interview with the Legendary 'smartrobot' 9d (Jin Jing 2p). 2011. http://gosensations.com

Figures & captions

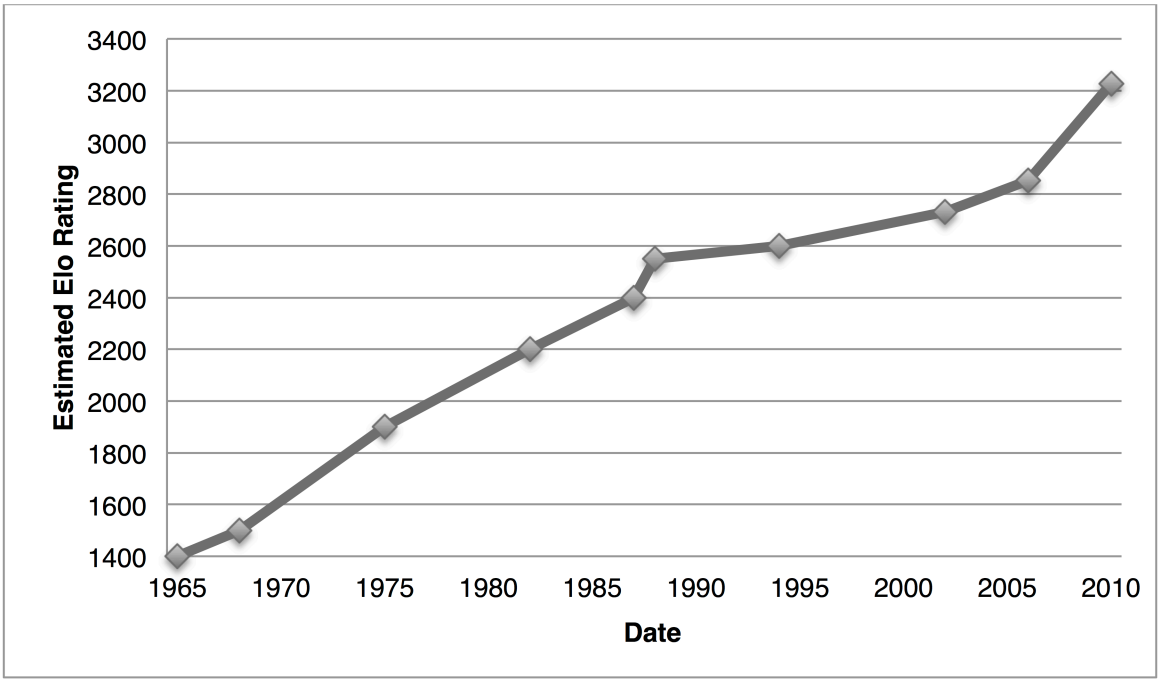

Figure 1: Estimated rankings of best available chess-playing programs during the development of the field, compared to human champion ranking. Rankings are given using the standard chess Elo scale. Estimated rankings from before 2000 include systems using high-end special-purpose hardware[75], while estimated rankings from after 2000 are from the SSDF Rating List which tests programs using commodity computing hardware[8]. The maximum rating achieved by human chess champions is approximately 2900; current computer chess performance is well beyond this level.

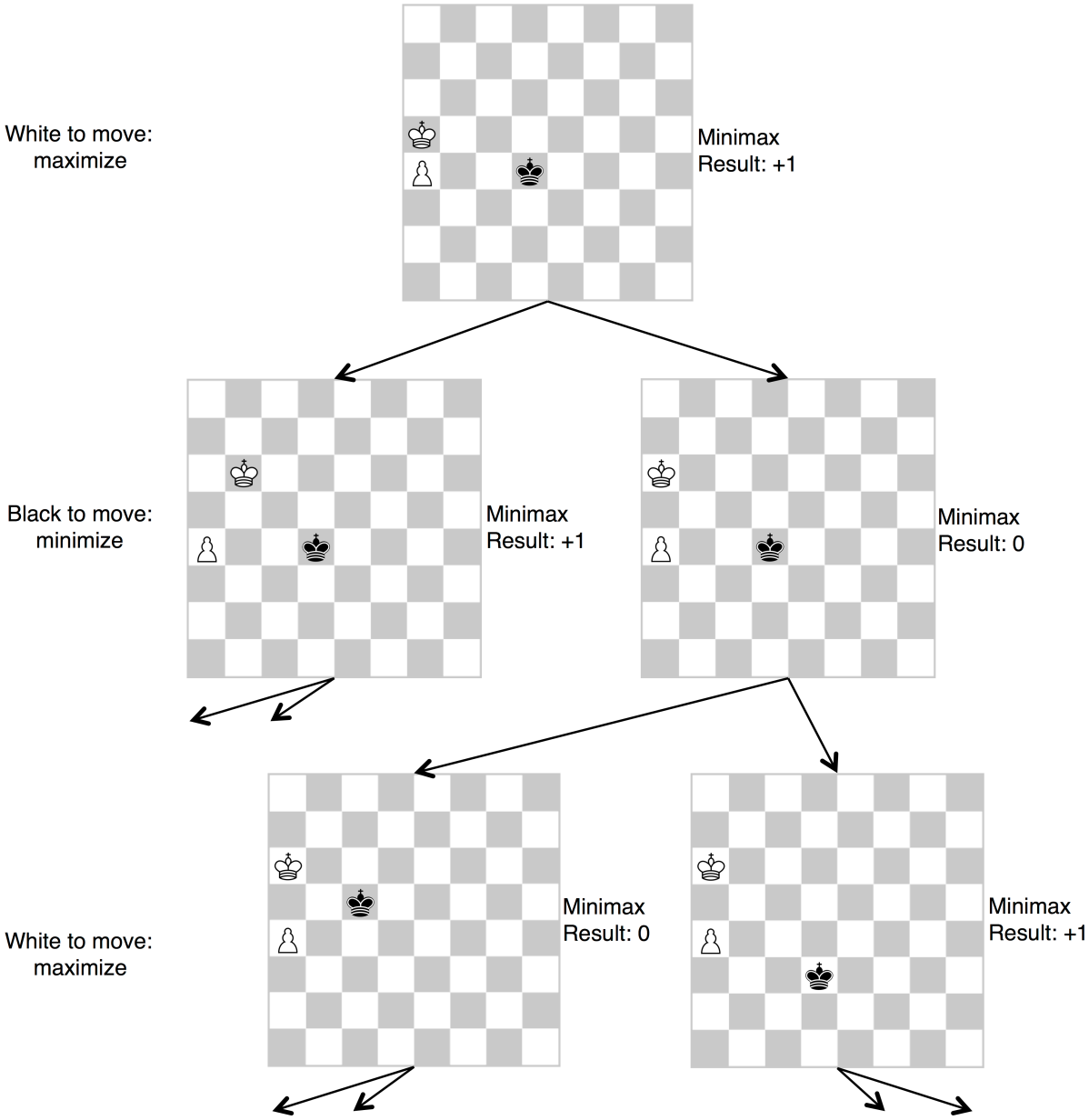

Figure 2: Game tree with minimax search, illustrated for a position near the end of a game of chess. Values here are +1 for white win, -1 for black win, and 0 for draw.

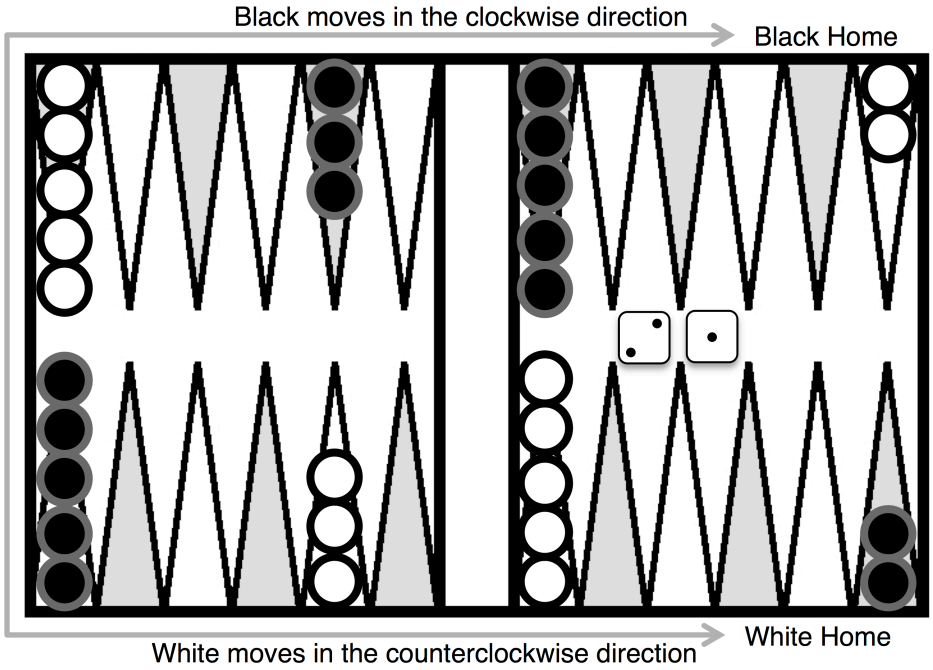

Figure 3: The game of backgammon

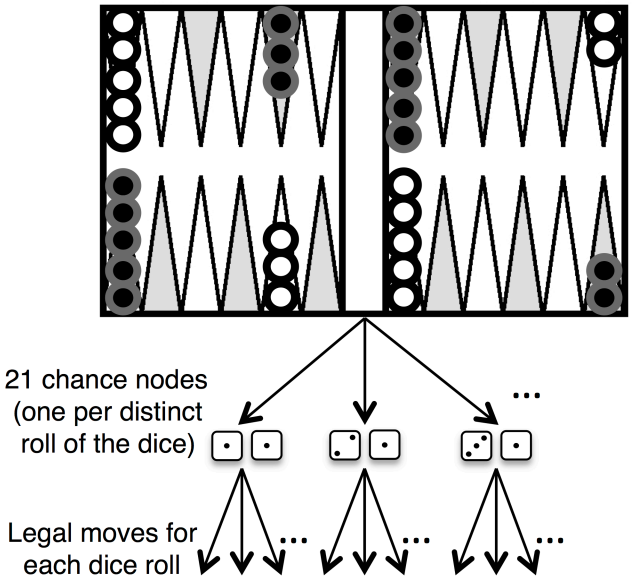

Figure 4: Backgammon game tree with chance nodes

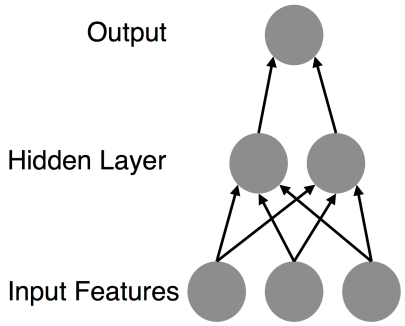

Figure 5: A small neural network. In TD-Gammon, the input features represent the state of the backgammon board, and the output is an evaluation of the position. Each hidden layer and output node computes a weighted sum of its inputs, and applies a sigmoid function to this sum to compute its output. The weights are set by a machine learning procedure using gradient descent. TD-Gammon uses a much larger neural network than is pictured here.

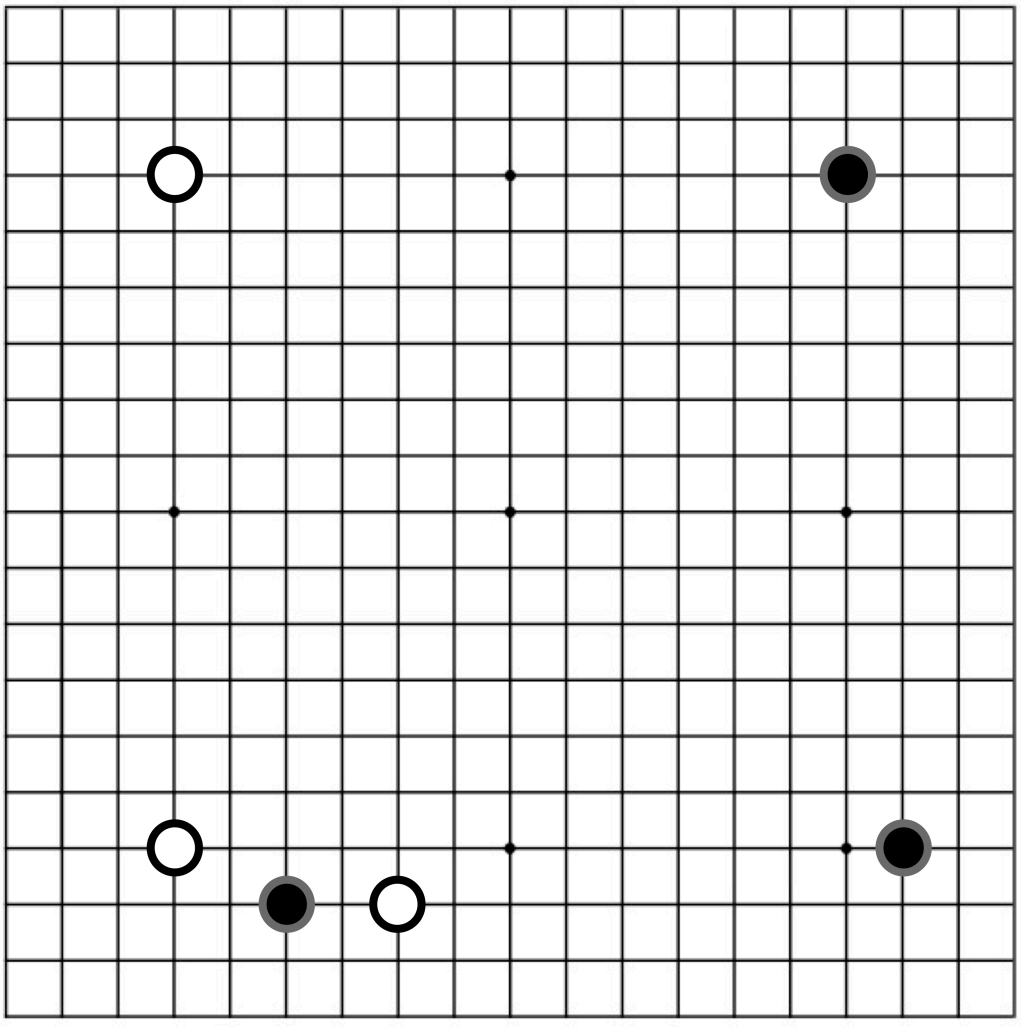

Figure 6: The game of Go

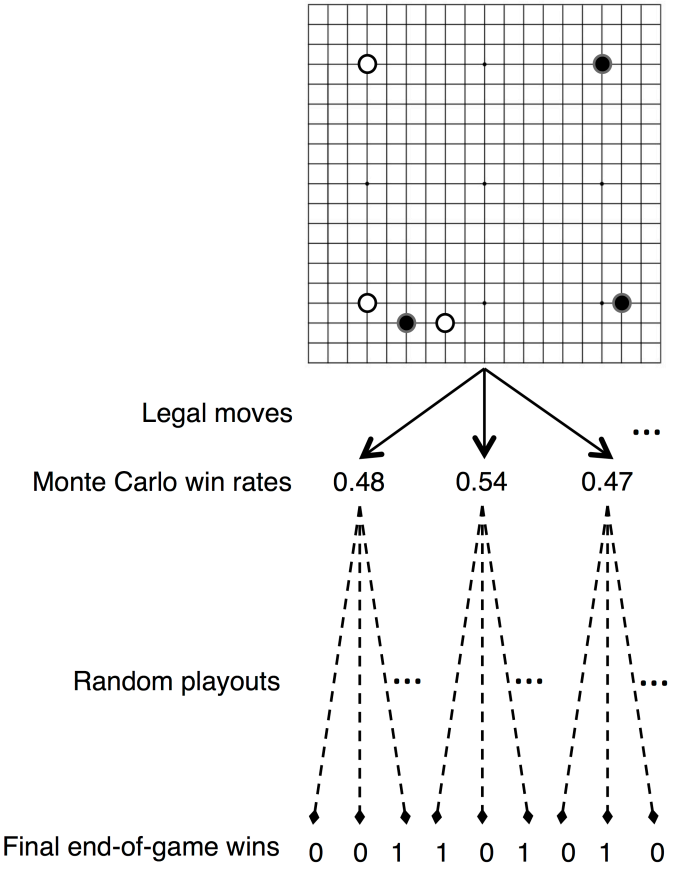

Figure 7: Monte Carlo Tree Search. Each legal move is evaluated by playing out from it many complete games to the end, taking the average win rate of the final positions in these games. The move with maximum win rate is then selected for play.

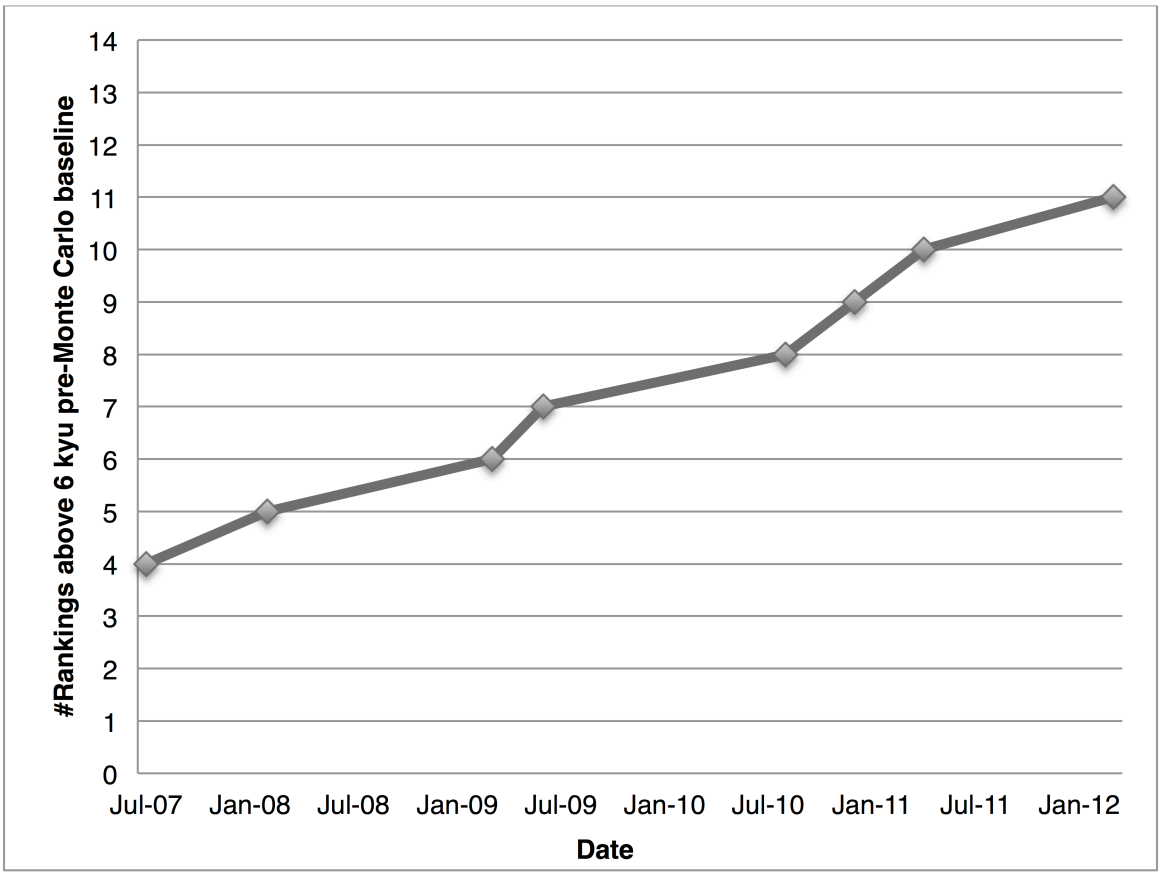

Figure 8: Ranking of Monte Carlo programs for Go; ratings held for at least 20 rated games on the KGS Go server where they compete against a wide variety of human opponents.[50,76] Rankings are relative to the best 6 kyu KGS ranking achieved by the classical pre-Monte Carlo programs GNU Go and Many Faces of Go from 2006[77]; 6 kyu has a relative ranking of 0 on the bottom of the vertical axis. The value 14 at the top of the vertical axis represents a maximum ranking on KGS, which corresponds approximately to low professional Go player rankings.[78]

Tables

Table 1: A chess evaluation function based on commonly understood piece values

Piece	Value for white piece	Value for black piece
Pawn	+1	-1
Bishop	+3	-3
Knight	+3	-3
Rook	+5	-5
Queen	+9	-9

Table 2: Analysis of errors by TD-Gammon and expert human opponents in head-to-head backgammon matches.[27]

Match	**TD-Gammon** Average errors per game	**Expert human opponent** Average errors per game
TD-Gammon 2.1 vs. Bill Robertie 1993	1.67 errors per game	2.12 errors per game
TD-Gammon 3.1 vs. Malcom Davis 1998	0.59 errors per game	1.85 errors per game

Table 3: Game state space sizes and status

Game	**Estimated State Space Size**[10,22,27,72,73,74]
Computer Game Playing Strength Near or Above Human Champion Level	
Chess	10^{53}
Backgammon	10^{20}
Othello	10^{28}
Shogi	10^{71}
Gomoku	10^{105}
Limit Hold'Em Poker (ACPC version)	10^{18}
Computer Game Playing Strength Well Below Human Champion Level	
No Limit Hold'Em Poker (ACPC version)	10^{76}
Go	10^{171}
Arimaa	10^{43}
Stratego	10^{115}

Further Reading/Resources

Russell S, Norvig P. Artificial Intelligence: A Modern Approach (3rd Edition). Prentice Hall, 2009.

Schaeffer J, van den Herik HJ. Chips Challenging Champions: Games, Computers, and Artificial Intelligence, North Holland, 2002.

Browne CB, Powley E, Whitehouse D, Lucas SM, et al. A survey of Monte Carlo tree search methods. IEEE TCIAIG 2012, 4:1-43.

Related Articles

Article ID	Article title
COGSCI-250	Reinforcement Learning, Computational Models
COGSCI-046	Robot Soccer
COGSCI-048	Search, Adversarial