

Dec. 2008 Revised Appendix II

From Human Temporal Difference Learning to Monte Carlo Computer Go: Interviews with Rémi Coulom, Olivier Teytaud, and David Fotland

By Peter Shotwell
© Feb.-Dec. 2008

A Short Interview with Rémi Coulom of CrazyStone

Feb. 2008

Recently, there have been stunning developments in computer play that is based on earlier work in human temporal difference learning. What follows does not pretend to be comprehensive, and I don't pretend to be an expert, although Rémi Coulom, developer of Crazy Stone, the world's strongest computer go program, graciously looked this article over and made some comments. However, all mistakes are mine, so please read this only as a portal into what has turned out to be a better method than the classical hand coding that has dominated computer go programming for so long.

Back in 1994, an article appeared in *Advances in Neural Information Processing*: 'Temporal Difference Learning of Position Evaluation in the Game of Go' by N.N. Schraudolph, P. Dayan and T.J. Sejnowski of the Computational Neurobiology Laboratory of The Salk Institute for Biological Studies in San Diego. Their abstract read:

The game of Go has a high branching factor that defeats the tree search approach used in computer chess, and long-range spatiotemporal interactions that make position evaluation extremely difficult. Development of conventional Go programs is hampered by their knowledge-intensive nature. We demonstrate a viable alternative by training networks to evaluate Go positions via temporal difference (TD) learning.

Our approach is based on network architectures that reflect the spatial organization of both input and reinforcement signals on the Go board, and training protocols that provide exposure to competent (though unlabelled) play. These techniques yield far better performance than undifferentiated networks trained by self play alone. A network with less than 500 weights learned within 3,000 games of 9x9 Go a position evaluation function that enables a primitive one-ply search to defeat a commercial Go program at a low playing level.

Coming up to 2007, the Wikipedia article on Temporal Difference Learning discussed the situation.

Temporal difference learning is a prediction method. It has been mostly used for solving the reinforcement learning problem. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. TD resembles a Monte Carlo method because it learns by sampling the environment according to some policy. TD is related to dynamic programming techniques because it approximates its current estimate based on previously learned estimates (a process known as bootstrapping). The TD learning algorithm is related to the Temporal difference model of animal learning.

The TD algorithm has also received attention in the field of Neuroscience. Researchers discovered that the firing rate of dopamine neurons in the ventral tegmental area (VTA) and substantia nigra (SNc) appear to mimic the error function in the algorithm. The error function reports back the difference between the estimated reward at any given state or time step and the actual reward received. The larger the error function the larger the difference between the expected and actual reward. When this is paired with a stimulus that accurately reflects a future reward the error can be used to associate the stimulus with the future reward.

Dopamine cells appear to behave in a similar manner. In one experiment measurements of dopamine cells were made while training a monkey to associate a stimulus with the reward of juice. Initially the dopamine cells increased firing rates when exposed to the juice, indicating a difference in expected and actual rewards.

Over time this increase in firing back propagated to the earliest reliable stimulus for the reward. Once the monkey was fully trained the dopamine cells stopped firing. This mimics closely how the error function in TD is used for reinforcement learning.

The relationship between the model and potential neurological function has produced research attempting to use TD to explain many aspects of behavioral research. It has also been used to study conditions such as schizophrenia or the consequences of pharmacological manipulations of dopamine on learning.

The 'Computer Go' article in Wikipedia explains that:

One major alternative to using hand-coded knowledge and searches is the use of Monte-Carlo methods. This is done by generating a list of potential moves, and for each move playing out thousands of games at random on the resulting board. The move which leads to the best set of random games for the current player is chosen as the best move. The advantage of this technique is that it requires very little domain knowledge or expert input, the tradeoff being increased memory and processor requirements. However, because the moves used for evaluation are generated at random it is possible that a move which would be excellent except for one specific opponent response would be mistakenly evaluated as a good move. The result of this are programs which are strong in an overall strategic sense, but are weak tactically. This problem can be mitigated by adding some domain knowledge in the move generation and a greater level of search depth on top of the random [evaluation]. Some programs which use Monte-Carlo techniques are MoGo and CrazyStone.

In 2006, a new search technique, upper confidence bounds applied to trees (UCT), was developed and applied to many 9x9 Monte-Carlo Go programs with excellent results. UCT uses the results of the play outs collected so far to guide the search along the more successful lines of play, while still allowing alternative lines to be explored. . . .

In 2007, Rémi Coulom developed a new method of generating candidate moves for the UCT algorithm based upon machine analysis of ELO scores/past games. As a result of these changes, CrazyStone

has shown an improvement of roughly 600 ELO points on the CGOS server. . . . [As of December 2007, the rankings given in the rest of the article are outdated— CrazyStone, which won the December 2007 UEC championship with a perfect record was a 2-kyu on KGS.]

Late in 2006, Coulom explained his methods to *Wired* magazine:

Wired News: What makes programming go so much tougher than chess?

Remí Coulom: In Go, you don't capture pieces, and so it's very difficult to say that black is ahead or white is ahead just by looking at the board. In order to survive, a group of stones needs to surround two 'eyes'—empty areas that can't be invaded by the opponent.

On a 19-by-19(-line) board, you'll have plenty of stones whose life or death status is undecided, and this is extremely difficult to analyze statically. This is different from the situation with chess or (checkers), where you can look at the board and say, 'I have one more pawn than you.'

WN: What are 'Monte Carlo' methods and how do they apply to Go?

Coulom: Monte Carlo methods are named after a quarter of Monaco that's famous for its casinos. In the case of Go, the basic idea goes like this: To evaluate a potential move, you simulate thousands of random games. And if black tends to win more often than white, then you know that move is favorable to black.

WN: With 250 moves in a typical game, that must take a lot of computational power.

Coulom: The version of CrazyStone in the Torino Olympiad [in 2006] ran on a four-CPU machine—two dual-core AMD Opterons at 2.2 GHz—and did about 50,000 random games per second. Unlike traditional algorithms, the Monte Carlo approach is extremely easy to parallelize, so it can take advantage of the multi-core architecture of the new generation of processors.

WN: CrazyStone was not the first program to use Monte Carlo methods, but it was successful enough that it started a trend among Go programmers. What was your innovation?

Coulom: Because you can't sample every possible random game, the Monte Carlo algorithm can easily fail to find the best moves. For instance, if most of the random games resulting from a certain move are losses, but one is a guaranteed win, the basic algorithm would take the average of those games and still evaluate it as a bad position.

CrazyStone is clever enough to avoid this problem. When it notices that one sequence of moves looks better than the others, it tends to play it more often in the random games.

WN: Why have people like Nick Wedd, the moderator of the monthly KGS tournaments, complained that watching games played by Monte Carlo programs can be boring?

Coulom: Monte Carlo programs maximize the probability of winning, not the margin that they win by. When they're very far ahead of the opponent, then they'll always play a safe move, which might look boring compared to more aggressive alternatives. It may be boring to watch, but it's more efficient in winning games.

WN: I've heard that a lot of the top Go programs are written by top Go players. What's your experience with the game?

Coulom: Before I started to write my first Go program, I decided I was going to play well enough to beat the other programs out there. But I don't think being a strong player is important to write a strong program. When I was still programming chess, this was obvious: my program was immensely stronger than me.

Some of the programs out there do use these set sequences of play, called joseki, but I avoid hard-coding this knowledge. I see some programs blunder because they blindly apply a hard-coded pattern.

At the 2007 European Congress, Martin Meuller, one of the leading developers and commentators on computer go, gave a lecture summing up the current state of the art of computers that play

go. His website is

<http://www.cs.ualberta.ca/~mmueller/cgo/villach2007.html>

From the report in the July 23, 2007 *American Go Association E-journal*:

Last year, Guo Juan 5P played a series against CrazyStone on a 7x7 board in which the program always won or got a jigo when playing white against the pro; this year MoGo scored 9 wins and 5 losses against Guo Juan on a 9x9 board. 'Monte Carlo programs play many strange moves,' conceded Mueller, 'but they're very good at winning. All without a single line of programming.' Such programs run as many as 100,000 simulations—or 1 million moves per second—for each move in a 9x9 game.

When asked about these figures, Coulom kindly responded in an email:

A payout for 9x9 is about 100 moves. . . . The fastest MC program I know does indeed do 100,000 simulations per second (that is Libego, by Lukasz Lew). But that means 10 million moves per second. 1 simulation = about 100 moves. 10,000 simulations per second is typical (when running on one core).

The AGA interview continues:

'Why does it work so well?' Mueller asked. 'There's no theoretical explanation, although we have excellent empirical results.' In other words, a broadly grinning Mueller said, 'We don't really know.' Although Mueller said that many researchers now think it's 'just a matter of time before there's a professional-level go-playing program,' he thinks it may be farther off. 'My own feeling is that we need one or two more good ideas, but where they'll come from I don't know.'

* * * * *

Further information about UCT can be found at <http://senseis.xmp.net/?UCT> and at Rémi Coulom's website, <http://remi.coulom.free.fr/Amsterdam2007>.

For an ongoing, day-to-day, bird's eye view of where these new ideas might come from, you can subscribe to the computer go discussion list at

<http://hosting.midvalleyhosting.com/mailman/listinfo/computer-go>

An Interview with Olivier Teytaud of MoGo

June 2008

At the beginning of the year, I posted an appendix to my Cognitive Psychology article in the Bob High Library on the AGA website. It ended with a discussion of how human learning studies spurred on the Monte Carlo method of programming, which is now being successfully used in many fields. Like the game of Battleship is played, in go the position of the next move is largely based on the win-lose results at the end of the playouts of almost random searches of future moves. These choices are modified by introducing a certain amount of previous 'history,' meaning the factoring in of the proximity to the last move, adding off-line pattern values taken from thousands of pro games, and estimating the amounts that can be captured or escape. If a certain move leads to 80% wins or more in the search tree, it is played, and can lead to usually impressive (or, sometimes disastrous) results.

Invented during the Manhattan Project of World War II and now using enormous search speeds and sometimes multi-core or message-passing parallelization, the Monte Carlo approach is much faster than the 'simple' memory-crunching computers that have beaten the best chess players. It also began outdistancing the 'classical' human knowledge-based go programs when the Upper Confidence Trees (UCT) algorithm was added in 2006.

In using UCT, different moves within the game tree search are treated like 'one-armed bandit' slot machines that return random results—but, unlike in casinos, where the machines are finely tuned, the results on some bandits are better than others. Because of this, the famous Exploration-Exploitation Dilemma arises and choices have to be made between making what seems to be a good move now (by pulling the arm of a previously high-paying bandit), or spending time playing other bandits which might (or might not) produce a better pay-off.

In humans, the process of resolving the Dilemma is called Temporal Difference Learning, which combines the Monte Carlo method with dynamic programming (DP) ideas (the 'programming' is mathematical optimization and has nothing to do with computer programming). In Temporal Difference Learning, MRI studies show how different parts of the brain start working against each other to

narrow down our choices about future actions. However, it is important to note that in these studies, the emphasis is not on how the brain makes the best judgment—the traditional approach—but how it reduces its ‘regret’ for making wrong choices.

In the world of ‘thinking’ machines, algorithms called Rapid Action Value Estimations (RAVE) were developed to solve the problem of the Dilemma. And in the business world, Behavioral Targeting algorithms try to figure out profitable ‘moves’ for companies by predicting the behavior of opposing customer ‘players’ who are making choices about such things as what to buy or where to go on the Internet.

Dr. Olivier Teytaud of the University of Paris is one of the developers of MoGo, the first program to beat a professional on 9x9 and the first to achieve a 1-*dan* rank on KGS in 19x19 ten-minute-a-side games. He generously responded to my email questions about what is happening on the front lines of computer go.

MoGo improves everyday. The version of yesterday evening wins with probability 57% against the version of yesterday morning (OK, it's not so nice everyday :-).

Most of the MoGo team are not go players and there is very little math involved. All the efficient programs, as far as I know, use the following elements:

- *A Monte-Carlo simulator for evaluating roughly the quality of a position. This method simulates a random game several times, from an initial position, in order to have a preliminary idea of its value.*
- *An incremental building of a tree of situations, representing possible future states; a main part of the algorithm is the so-called ‘bandit-part,’ which triggers the compromise between exploration (trend to analyze more carefully the situations which are less analyzed than other situations) and exploitation (trend to analyze more carefully the situations which are the most likely).*
- *In all successful programs, the Monte-Carlo part is biased through patterns. In fact, one of the main recent improvements consists in reducing the bias—as we improve the computational power (thanks to parallelization), we must reduce the bias induced by the patterns of the Monte-Carlo part.*

- *In all successful programs also there are correlations between one move and the next move in the Monte-Carlo part. This seems to be absolutely necessary. Some programs include a dependency with respect to the two previous moves. This is an important achievement of the early MoGo team, interestingly analyzed in the Ph.D. thesis of Sylvain Gelly.*
- *At least Leela, MoGo, CrazyStone, and probably some others have an SMP parallelization (the Symmetric Multiprocessing of many computers).*
- *In some programs, some bias is introduced in the tree exploration; we a priori prefer moves which match some known patterns, or some rules (e.g., is it worth considering the possibility of a ko, or an atari?); from this point of view MoGo was quite weak until a recent date, but now it is much better. Mango and CrazyStone are top-level programs from this point of view.*

- *The following elements are less widely used but are also important tricks:*

- *In some programs (e.g. MoGo), a so-called RAVE heuristic approximates the value of a sequence of moves by the value of a permutation of an already analyzed sequence of moves. This part is very efficient in MoGo, and is probably a main reason for the efficiency of MoGo on small boards—for small boards, I guess MoGo is currently the best program whenever we do not use the MPI-parallelization (the Message Passing Interface which keeps the Central Processing Units [the CPUs] from interfering with each other's work).*
- *Some programs also use a parallelization without shared memory. This is far less intuitive but quite efficient, particularly in 19x19. Thanks to this parallelization, MoGo can be quite strong in 19x19; it recently reached 1-dan on KGS with 64 quad-core machines on blitz games (10 minutes per side). We have not yet experimented with longer time settings—the priority for the moment is the ‘merging’ of all the improvements developed by various people into only one version of MoGo.*
- *In 9x9, openings are very important. This part is not yet stable in MoGo, but already provides significant improvement, in particular for fast games. For long time settings, the improvement is negligible, but we're working optimistically on it!*

Many elements are not well understood. Essentially, very important modifications in the Monte-Carlo methods come from trial and error. Even the UCT part, which was the most clearly understood, is now outperformed by new methods. A particularly disappointing point is that we cannot estimate the quality of our moves: we don't know when we are sure that MoGo is going to choose a good move. This is quite important, as this could provide a tool for choosing the time used for a given move: when we are not sure of a move, we might want to spend a few more minutes on it. Humans spend a lot of time on some moves, and not at all on some other moves—whereas MoGo spends almost exactly the same time on all moves—there is just a regular decrease of time per move during the game. This is a strong weakness in all current implementations.

Some recent directions are as follows:

- First, MoGo is sufficiently strong for benefiting from classical opening books in 19x19. This was false until a recent date.*
- Second, we can now tune MoGo, and probably other strong programs as well, so that it solves Tsumego. This is much easier than tuning it by self-play, because self-play is highly time-consuming, and makes sense only now—when MoGo was 10-kyu, it was meaningless to try to make it solve difficult Tsumego.*
- As well as many programs, MoGo was playing a so-called 'cosmic,' center-oriented style; now, it is much more classical (and stronger). But another trouble, which still holds, is that MoGo is too aggressive. It is quite efficient in killing groups, but it also tries this against groups which are clearly alive. Modifying the Monte-Carlo part might be a good solution for that.*
- Large-scale parallelization is possible in 19x19. MoGo would be much, much, much stronger if we had access to huge machines (technically speaking, to big clusters of SMP with good switches) and this looks like a possibility early this fall.*

To explain further, a multi-core machine has several cores accessing the same memory and there is no need for messages between cores, whereas a cluster is made of processors like Disk Sectors, each of them working on its own memory. Multi-core

parallelization is a priori easier, but the number of cores on the same machine is limited to a few cores sharing the same memory because they interfere with each other, whereas a cluster can be made of thousands of cores. The parallelization on a cluster is much more difficult and much more recent in Monte-Carlo go, and can be applied efficiently with much bigger machines. This is the main strength of MoGo; MoGo was the first code with a very efficient parallelization on clusters. I guess many programs will have such a parallelization soon. MoGo combines both parallelizations producing a cluster of multi-core machines. For example, the cluster used for games against the professional Catalin Taranu was made of 32 machines, each of them having eight cores. The eight cores of each machine work on the same memory, but in order to work together, the 32 machines have to communicate by sending messages. This is very slow on standard networks, but some specialized networks, based on fast cards and fast switches, are much better.

As for how all this developed, as far as I know from the published papers, the basic idea of mixing bandit techniques and Monte-Carlo simulations came from Rémi Coulom (the author of CrazyStone); UCT comes from MoGo and is also now in CrazyStone; Progressive widening (the 'unpruning' of possible moves that had been discarded, sometimes because threats have been uncovered from other searches) has been introduced in CrazyStone and Mango.

The RAVE heuristic comes from MoGo and is incorporated in Leela, but not in CrazyStone or Mango; Correlations between successive moves in the Monte-Carlo part were initially introduced in MoGo and is now also used in CrazyStone, Leela, Mango and all successful algorithms; as mentioned, Multi-node Message-Passing Parallelization has been introduced in MoGo and not yet in CrazyStone, Leela, or Mango; Expert knowledge as a bias in the bandit was first introduced in CrazyStone and is not yet very developed in MoGo, but we are working on it.

I want to point out that we do not consider MoGo as only the result of our work. Instead, it's the result of plenty of contributors, including go players, computer-scientists, mathematicians, from several countries, the authors from Mango, CrazyStone and UCT, and members of the University of Alberta in Canada.

Lastly, MoGo is not just a program for computer-go because we are developing several other applications with the same algorithm. For example, the same techniques as those employed in MoGo can

be used in resource management, such as deciding which resource should be used for producing electricity. Choosing the next move in go is analogous to choosing the next reservoir to be used, (such as hydroelectricity, nuclear plants, etc.), and winning a go game is analogous to the saving of money and the reduction of the environmental impact of electricity management.

* * * * *

A free version of MoGo can be downloaded at <http://www.lri.fr/~gelly>

Leela can be bought at <http://www.sjeng.org/leela.html>

See also: <http://www.pascal-network.org/article4.pdf> and <http://www.lri.fr/~teytaud/crmogo.en.html>

For games of MoGo vs. humans, see <http://www.lri.fr/~teytaud/iago>

For a game between MoGo and CrazyStone, see http://project-oz.com/public_html/article.php?story=20080127044215866

Dr. Teyaud has recently updated the Wikipedia article on computer go at http://en.wikipedia.org/wiki/Computer_Go

For a good, detailed explanation of UCT that I became aware of too late to utilize in this paper, along with a visualization of its playing Othello while ‘knowing’ nothing but the rules, see <http://www.hvergi.net/2008/06/visualizing-gameplaying-algorithms>

For the names and authors of the almost 100 go programs running on KGS, go to <http://www.weddslist.com/kgs/names.html>

For Temporal Difference Learning, see the Wikipedia article at http://en.wikipedia.org/wiki/Temporal_difference_learning

For MRI research on human learning, see, for example, <http://neurodudes.com/2006/06/22/softmax-rule-for-exploration-exploitation/>

For a dictionary of computer terms, go to www.webopedia.com

An Interview with David Fotland of Many Faces of Go

Dec. 2008

David Fotland started seriously playing go in 1980 and, since he was interested in artificial intelligence, was soon at work on a go program. His first version couldn't fight, but the second, which he called 'Many Faces of Go,' could, and it evolved over the years into today's 9x9 and 19x19 world champion. This was a remarkable feat, since as late as the end of 2007, his knowledge-based program had fallen considerably behind the new Monte Carlo approaches.

In December 2008, as a continuation of the interviews with Rémi Coulom of CrazyStone and Olivier Teytaud of Mogo, I asked David how he did it. His emailed answers are slightly edited and cobbled with some comments from the computer-go email chat group. Square Brackets [] indicate my additions.

It was a lot of fun working hard on something new that was going to be much stronger. There were a lot of new interesting problems to solve.

I'd been working on computer go since about 1982. As the power and memory of computers increased thousands of times, the optimal algorithms have changed. In the beginning it was important to design very small data structures to fit in about 400 K Bytes on DOS. Now Monte Carlo programs need hundreds of megabytes.

I'd won the Ing world championship in 1998 and the 21st Century cup in 2002. The big world championships were a large part of my motivation, and when they ended I spent much less time on my program. The traditional go programs are knowledge-based and very complex. As Many Faces got stronger it took more and more effort to get additional strength from it. So I took a couple of years off to write the world champion arimaa program.

Monte Carlo go had been tried since Brugmann in 1993 without much success, so I didn't expect much from the new Monte Carlo programs like CrazyStone and Mogo. They got strong quickly at 9x9, but didn't have much success at 19x19.

In 2006 and 2007 I was working to add a full board search to Many Faces. In late 2007 it was about two stones stronger than Version 11 and I was preparing to release it. Then in December

CrazyStone won the UEC cup 19x19 contest. That convinced me that Monte Carlo/UCT was stronger than traditional programs and I decided to replace my full board alpha-beta search with Monte Carlo Tree Search (MCTS).

I studied all the literature and thought about it, and started coding in February 2008. In early April my Japanese publisher said they wanted an MCTS program by the end of June, so I started working on it about 50 hours a week.

I tried more than 400 variations on the basic UCT algorithm or playout strategy during this half-year of intensive development. The engine was written in C and tuned from the start for performance. I couldn't have done this many experiments (often several per day), without a very fast engine, because I used 1000 game contests to see if there was improvement with statistically significant results. Part of the reason I won is because the basic UCT/MC is so fast that I can incorporate the slow Many Faces knowledge and still get over 10K playouts per second per CPU on 9x9.

The machines used in the ICGA World Championship are in Redmond, at Microsoft. It's a 4x8-core XEON, with 16 GB per core and 40 Gbps Infiniband networks. I'm very grateful to John Costello and Shahrokh Mortazavi who work on Microsoft's High Performance Computing operating system for large clusters of CPUs.

In April, John contacted me and asked if I would be interested in creating a scalable version of Many Faces for their operating system as a demo. This supercomputer version of Many Faces will be available free to Microsoft's supercomputer customers. In exchange they agreed to make cluster time available for computer go contests. I started writing the supercomputer code in August. Mogo scales to hundreds of cores, and I had hoped to duplicate that for the world championship in Beijing, but I never got the code to scale beyond 32 cores. Currently with more cores it plays weaker, but that should be fixed soon and, if the test results are good, I'm going to use Windows HPC server 2008 which Microsoft has just released. It is a server operating system for high performance clusters and their MPI implementation is about 10% faster than Linux on the huge machines with thousands of cores. A video of the Demo is at <http://www.youtube.com/watch?v=Qe0o-lvHOa0>.

[Olivier Teytaud commented in the computer-go group about the machine that runs Mogo:

Huygens, has 3328 cores but] ‘ . . . Mogo was allowed to use 800 cores, not more, and only for games against humans. We have no access to so many cores for computer-computer games (if there were only three teams involved, we could :-). For some games Huygens was unavailable at all, and Mogo played with much weaker hardware (some quad-cores, however, it is not so bad :-)]

[David continues] *When I won two gold medals in Beijing I decided to introduce the program immediately, as a preliminary version first, then a final version in early November. Now, in December, I’m back to working on the engine.*

Its strength is due to several things. First, a very fast playout engine. The more random games, the stronger the program. Second, a disciplined engineering approach to development. Third, the publications by the authors of CrazyStone, Mogo, and Mango, explaining their algorithms. My UCT implementation is a little different from any of them, and combines ideas from all of them with some original work of my own. Fourth, effectively using the Many Faces knowledge in the search. On 19x19, Many Faces plays with a more human style than the other programs using MCTS.

In a single machine like a quad core PC, a single search can be shared by all cores, with common data in memory. In a bigger machine, a network is used to share data. Separate searches on each node exchange data as they work.

Evaluation has always been the tough unsolved issue in computer go. Certainly most of go programming time for over ten years was spent working on the evaluation function. MCTS eliminates the evaluation entirely. The old Many Faces evaluated 100 positions a second. The new one plays tens of thousands of full games each second.

Monte Carlo by itself does not make a strong program, however. The tree search, AMAF/RAVE algorithm, local move bias, progressive widening, and many other enhancements that make up MCTS are what makes the new idea work. They would not have been discovered without the collaboration of many researchers, and the encouragement from Don Dailey’s CGOS go server.

What made the modern MCTS systems so strong were four breakthroughs by the French researchers of Mogo, CrazyStone and others. They solved the bandit problem, by recognizing that the random playouts should be more frequent for the best moves; the tree search, by recursively applying the bandit algorithm at each node in the search; by recognizing that the playouts should include local move sequences, and not be pure random; and by biasing the tree search with some prior go knowledge.

Thus, all of the MCTS programs have some means of biasing the UCT part of the search to focus quickly on good moves. Mogo uses RAVE. CrazyStone uses patterns learned from professional games. I use Many Faces' go knowledge. Many Faces knows a lot about shapes and fighting, so this makes it a stronger fighter than the other programs. It also makes its moves look more human-like or natural.

They also all use random games to gather statistics to evaluate win-rates for positions. But these random games are not purely random. For example moves are not allowed to fill eyes. All programs have some way to bias the random move generator to make the random games more sensible, for example favoring moves near the last move. The details are different for each program.

As for bandwidth in the supercomputer version, my approach and Mogo's are similar, but I just exchange less information than they do. They scale better than I do to many cores, and it might be because they share more information. It's more likely that my approach will resemble Mogo's because they scale better. Since they publish their approach and I don't, they can't really adopt mine.

By the way, I don't see much similarity between temporal difference learning and MCTS. It's possible to use TD to learn patterns for use in later games, but I don't do that. All of the Many Faces' knowledge is hand created by me (more like an expert system). Many Faces learns joseki and openings from games it plays, but that is rote memorizations, not TD. Rémi used a learning algorithm to find good patterns for MCTS that worked very well, but it was not TD, either.

Go seems particularly suited for MCTS and I'm sure it will be effective in some other games, but I can't predict which ones. I've thought about using MCTS for arimaa, and it might be better than alpha-beta. arimaa has a huge branching factor, and since the pieces move slowly, human long-term planning works well against

computers. MCTS prunes better, so it should get deeper Principal Variations. AMAF might help too, since, in a long plan, it is often not critical what order the moves are in.

I don't think we will know how well MCTS works with arimaa until someone tries it, but there are some things that make MCTS work well for go that don't apply. The AMAF or RAVE concept helps the search converge quickly. This algorithm assumes that good moves made later in a game are likely to also be good moves now. So when a random game is played, all the moves in the game that lead to a win are weighted higher at the current position. This assumption is often right for go, but doesn't work so well for games like chess or arimaa where the pieces move.

Arimaa is different enough from Chess or go that the techniques used for either don't apply directly. For my program I used a chess-like alpha-beta search, but with a smart go-like evaluation.

As for an @home approach, I don't think this will work for go since the bandwidth between home computers is more than 1000 times lower than in a supercomputer cluster. The @home systems work great for big problems that do not have time constraints but game playing is interactive and people expect reasonably quick replies. The problem with @home computational models is that you never know when the user will want their machine back, so you have the problem of deciding if a result is worth waiting for, or if you will send similar requests to multiple machines just to try and be sure that you get at least one reply.

I was contacted by someone in the Govt. of Singapore about trying exactly this (Go@home) and while it is interesting, it is not nearly as simple as SETI@ home [Search for Extra-Terrestrial Intelligence], where independent problems are being solved on [over three million] different machines. You do not expect to get the answer back to the primary server on any particular schedule. In go the answers are interdependent.

Incidentally, for the record, it seems likely now that the correct komi for 9x9 is 7.0. If so, I'd prefer 6.5 komi to 7.5, since 6.5 would have Black winning most games, and most other games have a first player advantage. This would give 9x9 go a similar first player advantage.

For bibliography, I learned from the papers by the Mogo team and Rémi, the Mango papers at <http://www.cs.unimaas.nl/go4go/mango/index.htm> and the survey papers by Bouzy at <http://www.math-info.univ-paris5.fr/~bouzy/publications.html>.

The computer-go group archives are also very useful at <http://computer-go.org/pipermail/computer-go/2005-December/004193.html>.

The Wikipedia article for arimaa is good and has the URL for playing at <http://en.wikipedia.org/wiki/arimaa>.

* * * * *

David Fotland's home page, where Many Faces of Go can be purchased, is at <http://www.smart-games.com>.

His free 9x9 interactive program can be found at <http://www.smart-games.com/igowin.html>. It is knowledge-based and scaled down so it becomes an excellent graded interactive teaching tool.