Fuego – An Open-source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search*

Markus Enzenberger Martin Müller

May 1, 2009

Abstract

Fuego is an open-source software framework for developing game engines for full-information two-player board games, with a focus on the game of Go. It was mainly developed by the Computer Go group of the University of Alberta. Fuego includes a Go engine with a playing strength that is competitive with the top programs in 9×9 Go, and respectable in 19×19 Go. This report describes the reusable components of the software framework and specific algorithms used in the Go engine. The description includes a new enhancement for Monte-Carlo tree search: a lock-free multithreaded mode.

1 Introduction

Research in computing science is driven by the interplay of theory and practice. Advances in theory, such as Monte-Carlo Tree Search and the UCT algorithm [4, 24], lead to breakthrough performance in computer Go, as in the programs Crazy Stone [4] and MoGo [15]. In turn, attempts to improve the playing strength of these programs led to improved algorithms, such as RAVE and prior knowledge initialization [16].

Fuego is an open-source software framework for developing game engines for full-information two-player board games, with a focus on the game of Go. Open source software can facilitate and accelerate research. It lowers the overhead of getting started with research in a field, and allows experiments with new algorithms

*Technical Report TR 09-08. April 2009. Dept. of Computing Science. University of Alberta. Canada. A short version of this report will be submitted for publication.

that would not be possible otherwise because of the cost of implementing state of the art methods. Successful previous examples show the importance of open-source software: GNU Go [14], by providing the first open source program with a strength approaching the best classical programs, has had a huge impact, attracting dozens of researchers and hundreds of hobbyists. In planning, Hoffmann's FF [18] has had a similar impact, and has been used as the basis for much followup research.

Fuego contains a state of the art implementation of MCTS with many standard enhancements. It implements a coherent design, consistent with software engineering best practices. Advanced features include a lock-free shared memory architecture. The Fuego framework has been proven in applications to Go, Hex, and Amazons.

The main innovation of Fuego lies not in the novelty of its methods or algorithms, but in the fact that for the first time, a state of the art implementation of these methods is made available freely to the public in form of a consistent, well-designed, tested and maintained open source software.

There is a number of other available systems: For MoGo [15, 17], only an executable is available. The reference MCTS implementation by Dailey [6] provides a very useful starting point, but is far from a competitive system. Recent versions of GNU Go [14] contain a hybrid system adding Monte-Carlo search to GNU Go. However, it is several hundred Elo rating points weaker than state of the art programs.

1.1 Why Focus on MCTS and on Go?

Research in computer Go has been revolutionized by MCTS methods, and has seen more progress within 3 years than in the two decades before. Programs are close to perfection on 7×7 and have reached high amateur dan level on the 9×9 board. The top programs are also dan level on 19×19 and have far surpassed the strength of classical programs. For example, David Fotland's new MCTS-based version of *The Many Faces of Go* is 5 stones stronger than his old program [13].

MCTS is the state of the art in computer Go and General Game Playing [12]. In Go, at the 13th International Computer Games Championship in Beijing 2008, MCTS-based programs dominated the field. In both the 9×9 and the 19×19 competition, the top half dozen programs used MCTS. The programs MoGo, Crazy Stone and Many Faces of Go have achieved successes against professional players in even games on 9×9 and with handicaps of 6-9 stones on 19×19 [33]. In General Game-playing, the MCTS program *CadiaPlayer* by Finnsson and Björnsson has won both the 2007 and 2008 AAAI competitions [12].

In several other games, the MCTS approach is being intensely investigated. In

the games of Amazons and Hex, MCTS-based programs have reached the strength of classical approaches [26, 1]. At the 13th International Computer Games Championship, the MCTS-based Amazons program *Invader* took the title from the five times champion, the classical alpha-beta searcher *8 Queens Problem*. In Hex, *MoHex 2008*, which used an earlier version of the Fuego MCTS engine described in this report, finished in a strong second place and won 3 out of 4 games in its mini-match against the winning program *Wolve 2008* [1].

2 The software framework

2.1 History

Fuego is built on two previous projects: Kierulf's Smart Game Board [21, 22] and Müller's Explorer [27, 28]. Smart Game Board is a workbench for game-playing programs that has been under development since the mid 1980's. Explorer is a Go-playing program built on top of Smart Game Board. It's history goes back to Kierulf and Chen's first version in 1988.

Motivated by the successes of Crazy Stone and Mogo, Enzenberger started to implement an MCTS program in 2007. This program was built on top of the Smartgame and Go core routines but independent of the remaining Explorer code base. This program, simply called Uct at first, became the basis of Fuego. Fuego became an open source project in May 2008.

2.2 Technical Description and Requirements

This paper describes version 0.3 of Fuego. File downloads and access to the version control system are available at a public open-source hosting service [9]. It is distributed under the terms of the GNU Lesser General Public License [25], version 3 or newer.

Fuego is a software framework. In contrast to a small library with limited functionality and a stable API, Fuego provides a large number of classes and functions. The API is not stable between major releases. Porting applications that depend on the Fuego libraries to a new major release can be a significant effort. For each major release a stable branch is created, which is used for applying critical bug fixes that do not break the API, and can be used by applications that do not want or need to follow new developments in the main branch. The code is divided into five libraries, and uses a largely consistent coding style. Figure 1 shows the dependency graph of the libraries and applications.

The code is written in C++ with portability in mind. Apart from the standard C++ library, it uses selected parts of the Boost libraries [7], which are available

for a large number of platforms. The required version of Boost is 1.33.1 or newer. Fuego compiles successfully with recent versions of gcc, from 4.1.2 up. Older versions of gcc will probably work but have not been tested.

Platform-dependent functionality, such as time measurement or process creation, is encapsulated in classes of the SmartGame library (see Section 2.4). The default implementation of those classes uses POSIX function calls. No attempt is made to provide GUI-specific functionality. GUIs or other controllers can interface to the game engines by linking, if they are written in C++ or a compatible language, or by inter-process communication using the Go Text Protocol (see Section 2.3).

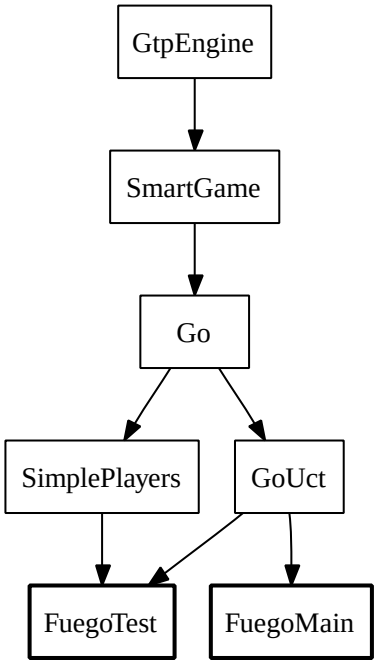

Figure 1: Libraries and applications of Fuego

2.3 GtpEngine library

GTP, the Go Text Protocol [11], is a text-based protocol for controlling a Go engine over a command and response stream, such as the standard I/O streams of the process, or network sockets. It was first used by GNU Go [14] and has gained widespread adoption in the Computer Go community in the last few years. Among other applications, it is used for connecting an engine to graphical user interfaces, for regression testing, and as a simple language to set engine parameters in config-

uration files. The protocol has also been used for other games with game-specific command sets, for example for the games Amazons, Othello [30], and Hex.

The GtpEngine library provides a game-independent implementation of GTP. Concrete game engines will typically derive from the base engine class and register new commands and their handlers. The base engine runs the command loop, parses commands and invokes the command handlers. In addition, GtpEngine provides support for *pondering*, to allow the engine to think during the opponent's time. The pondering function is a function of the subclass that is invoked whenever the engine is waiting for the next command. The base engine sets a flag when the next command was received to indicate to the subclass that the pondering function needs to terminate. This facilitates the use of pondering, because the necessary use of multithreading is hidden in the base engine. The functions of the subclass do not need to be thread-safe, because they are only invoked sequentially. GtpEngine also provides support for the non-standard GTP extension for interrupting commands as used by the GoGui graphical user interface [8]. Again, the necessary multithreaded implementation is hidden in the base engine class.

2.4 SmartGame library

The SmartGame library contains generally useful game-independent functionality. It includes utility classes, classes that encapsulate non-portable platform-dependent functionality, and classes and functions that help to represent the game state for two-player games on square boards. Further, there are classes that can represent, load and save game trees in Smart Game Format [19]. The two most complex classes are game-independent implementations of the alpha-beta search algorithm and Monte-Carlo tree search, which will be described in more detail in the next two sections. Both search classes assume that player moves can be represented by positive integer values. This restriction could be lifted in the future by making the move class a template parameter of the search.

2.4.1 Alpha-beta search

SmartGame contains a game-independent alpha-beta search engine. It supports standard alpha-beta search as well as iterative deepening with fractional ply extensions, a transposition table, and move ordering heuristics such as the history heuristic and preordering the best move from the previous iteration. It also provides an implementation of Buro's Probcut [2]. Tracing support allows saving a search tree as an sgf file, complete with game-specific text information in nodes.

2.4.2 Monte-Carlo tree search

The base Monte-Carlo tree search class of the SmartGame library implements the UCT search algorithm [24] with a number of commonly used enhancements. Concrete subclasses need to implement pure virtual functions for representing the game state, generating and playing moves in both the in-tree phase and the playout phase, and evaluating terminal states.

The search can optionally use the Rapid Action Value Estimation (RAVE) heuristic [15], which was first used in the MoGo Go program and is an important enhancement for Go and other games with a high correlation of the values of the same move in different positions. RAVE is implemented in a slightly different form than originally used by MoGo. Instead of adding a UCT-like bias term to the move value and the RAVE value before computing the weighted sum, the weighted sum of the values is computed first, and the UCT bias term is added after that. Since initially most of the moves will have only RAVE samples, and the number of move samples is zero, the UCT bias term is slightly modified by adding 1 to the number of move samples. This avoids a division by zero with minimal impact on the value of the bias term for larger numbers of move samples. The weights of the move and RAVE value are derived by treating them as independent estimators and minimizing the mean squared error of the weighted sum. The mean squared error of the move and RAVE value are modeled with the function $a/N + b$ with N being the number of samples and constants a, b that need to be determined experimentally. The RAVE values can optionally be updated by weighting the samples with a function that decreases with the number of moves between the game states.

The move and RAVE values can be initialized with prior knowledge from game-specific static heuristics [15], if the subclass can provide such heuristics.

For efficiency, the tree implementation avoids dynamic memory allocation and allocates new nodes from an array of pre-allocated memory. Nodes are never deleted during a search. If the tree runs out of memory, the search can optionally be restarted after pruning leaf nodes with a count below a certain threshold.

Most of the parameters of the search class can be changed at run-time to facilitate tuning. The search also collects statistical information, for example the average in-tree and total length of simulations and the number of simulations per second.

2.4.3 Multithreading in the Monte-Carlo tree search

The search supports parallelization on a single system using multithreading and shared memory. Each thread runs the normal Monte-Carlo tree search algorithm and has its own game state. All threads share the same game tree; modifications of

the tree are protected by a global mutex. The maximum speedup of this parallelization method is bounded by the inverse of the time that is spent in the phases of the simulation that needs locking (the in-tree play phase and the tree update phase) as a percentage of the total time for the simulation. Other researches have proposed a more fine-grained locking mechanism that needs a separate mutex for each node in the tree, but this did not work well in practice [3]. The problem here is that usually a large number of nodes are shared in the initial in-tree move sequences of the different threads; at least one node, the root node, is always shared.

The Monte-Carlo tree search implementation of Fuego introduces a new lock-free multithreading mode that can improve the performance significantly [10]. Lock-free mode depends on hardware support for a specific memory model. Therefore it is an optional feature of the base search class that is switched off by default and needs to be enabled explicitly. The lock-free implementation does not use a mutex at all. Each thread has its own node allocator. If multiple threads expand the same node in the tree, only those children are used that are created last, which causes a small memory overhead. The move and RAVE values are stored in the nodes as counts and incrementally updated mean values. They are updated without protection of a mutex. This can cause updates of the mean to be lost with or without increment of the count, as well as updates of the mean occurring without increment of the count. In practice, these wrong updates occur with low probability and are intentionally ignored. The requirements on the memory model of the platform are that writes of the basic types size_t, int, float and pointers are atomic and that writes by one thread are seen in the same order by another thread. The popular IA-32 and Intel-64 architecture guarantees these assumptions and even synchronizes CPU caches after writes [20].

Figure 2 shows the performance of locked and lock-free multithreading with n threads in comparison to a single-threaded search given n times more time. The experiment was performed using the Monte-Carlo tree search Go engine of Fuego (see Section 3) on both 9×9 and 19×19 boards, with one second per move, on a Intel Xeon E5420 2.5 GHz dual quadcore system. Each data point shows the percentage of wins in 1000 games against the single-threaded version. The version using locking does not scale beyond two threads on 9×9 and three on 19×19. The lock-free version scales up to seven threads in this experiment. Preliminary recent results on a 16 core system suggest that the program continues to improve with more cores, though the gain is marginal in the case of 9×9.

The search supports other optional features that can improve the multithreaded performance depending on the game and the number of threads.

- Using more than one playout per simulation decreases the percentage of time that is spent in the in-tree phase in each simulation.

- The *virtual loss* technique [3] can reduce the chance that different threads explore the same or very similar in-tree sequences.

2.5 Go library

The Go library builds on the SmartGame library and provides functionality specific to the game of Go.

The most important class is an efficient implementation of a Go board, which keeps a history of all moves played and allows to undo them. It incrementally keeps track of the stones and liberties of all blocks on the board and allows to check moves for legality. All commonly used ko rules, simple ko, situational superko and positional superko, are supported.

The library contains a base class for generally useful but Go-specific GTP commands. Two classes implement widely used algorithms for detecting unconditional life: the Benson algorithm and Müller's extension for life under alternating play [27].

An opening book class allows to read opening positions from a data file using a simple text format, and matches them on the current position independent of rotations and mirroring of the position. One such data file is included.

2.5.1 Opening Book

Fuego contains both a 9×9 and a 19×19 opening book. The 9×9 book is hand-built. The design principles are to avoid bad moves, and to keep the game as simple as possible in favorable positions. To keep the book small, only one move is selected per position. This makes the book easier to maintain, but has one potential disadvantage - it could be exploited by opponents to prepare "killer" variations. Most of the book lines are short. A few longer lines were added in cases where a position is equal or favorable in the human's judgment but the program "panics" and starts self-destructing. However, these long lines are problematic and will probably be removed in the future.

Fuego's human-built book is in contrast to MoGo's massive machine-built book. It would be very interesting to compare these books, for example by swapping the books used by Fuego and MoGo. Another contrasting approach is Yogo's book, which apparently contains a large number of traps designed by a professional player. These traps are positions that are often misplayed by Monte-Carlo based programs.

The book is grown by analyzing losses, and by examining winning statistics of matches against other strong programs. Therefore the book is biased towards lines

occurring in practice, and still has big "holes" in other lines. The current book contains 1491 positions, which get expanded into 11598 positions by adding rotated and mirrored variations for fast matching. Most lines are of length 5 or shorter. In self-play the 9×9 opening book gives about a 60-70 Elo points gain. Interestingly, this gain seems to remain roughly the same, even when varying the number of simulations per move over the large range of 12 - 48000. Many weaknesses addressed by the book seem to be systematic weaknesses in Fuego's opening play that cannot be discovered easily by more search.

The 19×19 book consists of a small database of popular full-board openings extracted from dan-level amateur games. Its effect is largely cosmetic, to play more natural-looking moves in the very beginning. In contrast to most classical programs, the current Fuego does not contain a *Joseki* book with local standard sequences.

2.6 Other libraries and applications

SimplePlayers is a library with a collection of simple Go playing algorithms, for example a random player and a player that maximizes a simple influence function. They can be used for testing and reference purposes.

GoUct and FuegoMain are a library and application that contain the main Fuego engine and a GTP interface to it. The main Fuego engine is described in detail in Section 3.

FuegoTest is a GTP interface to functionality in the SmartGame and Go library. It exists to provide access to the engine via additional GTP commands, for example in regression tests, which need not be included in the main Fuego engine. FuegoTest can also be used to play against any of the players of the SimplePlayers library.

2.7 External Applications and Playing Engines that use Fuego

The following programs currently use the Fuego framework:

MoHex is a world-class Hex program [1]. It currently uses the UCT search framework from Fuego 0.3, along with some utilities such as reading and writing sgf, random number generation, and timers/clocks.

SRI has a research team working on a DARPA funded Go seedling project. They use Fuego 0.2.1 as a base to insert/interface their own players and test against other simple players in Fuego as well as entering it into CGOS. Their research explores machine learning techniques for learning new patterns dur-

ing "self-play" to improve playing strength, as well as testing of an influence function designed by Thomas Wolf.

Explorer is a classical knowledge- and local search-based Go program [27, 28]. The Fuego modules GtpEngine, SmartGame, Go and SimplePlayers were originally extracted from Explorer and are still used there. Explorer is kept in sync with the most recent subversion development version of Fuego.

TsumeGo Explorer is the world's best Life and Death solver for enclosed areas [23]. It uses the SmartGame and Go libraries and is kept in sync with subversion.

RLGO is a reinforcement learning Go program [31]. RLGO is built on the SmartGame and Go libraries from Fuego 0.1. In addition, the tournament version of RLGO uses the default playout policy from Fuego 0.1 to rollout games to completion.

Arrow is an Amazons-playing program. It uses the classical alpha-beta search and other basic functionality from the SmartGame module. Arrow is kept in sync with subversion.

SOS implements a simple abstract Sum-of-Switches game and is used for experiments with Monte-Carlo Tree Search [32]. It currently uses the UCT search framework from Fuego 0.3.

Student Projects Several other current and previous research projects by students at the University of Alberta have used the Fuego framework for topics such as combinatorial games and adversarial planning. Examples are [29, 34].

3 The Go engine

The main Go engine of Fuego uses a full-board Monte-Carlo tree search. It is contained in a library named GoUct. The library contains classes that can be reused for other Monte-Carlo tree search applications in Go, for example local searches, and a GTP engine with the main Fuego player and commands for setting player and search parameters and querying information about the search. The application FuegoMain contains the top-level main function, which runs the GTP engine using the standard I/O streams of the process.

3.1 Full-board Monte-Carlo tree search

The Go-specific Monte-Carlo tree search class extends the Monte-Carlo tree search class of the SmartGame library by providing a Go-specific game state. The game-state uses the Go board of the Go library in the in-tree phase and a special Go board for the playout phase. The playout board is speed-optimized at the cost of reduced functionality: it still provides incrementally updated information about stones and liberties of blocks, but does not support undoing moves, and supports only the simple ko rule.

The full-board Monte-Carlo tree search class extends the Go-specific Monte-Carlo tree search class and implements a playout policy and prior knowledge.

The playout policy is similar to the one originally used by MoGo [15]. Capture moves or atari-defense moves, or moves that match a small set of hand-selected 3×3 patterns, are selected if they are adjacent to the last move played on the board. Fuego-specific enhancements include a move generator for 2-liberty blocks. If no move is selected so far, a global capture move is selected. If no global capture move exists either, a move is selected randomly from all legal moves on the board. A replacement policy attempts to move tactically bad moves to an adjacent point. Moves are never selected if all adjacent points are occupied by stones of the color to move, unless one of them is in atari. A pass in the playout phase is generated only if no other moves were produced.

The prior knowledge class initializes moves in the tree, that would have been played by the playout policy, with positive values for the player to move. An additional bonus is given to global matches of the 3×3 patterns of the playout policy, moves that set the opponent into atari, and all moves that are in a neighborhood up to a certain distance to the last move. Self-atari moves are penalized. The values and counts used in the initialization depend on the occurrence of moves in these move classes and on the board size.

A move filter is implemented to completely prune moves in the root node. This filter can run algorithms that would take too much time if they were run at every node of the search, or that use code that is not thread-safe. Currently, it prunes unsuccessful ladder-defense moves, because the result of the ladder would likely not become visible within the search tree on large board sizes. Also, moves to points on the edge of the board are pruned if there is no stone in a neighborhood of the point up to a certain distance.

The evaluation of terminal positions in the playout phase is simple, because passes are only generated if no other moves are generated. After two consecutive pass moves, the position contains only alive blocks and single-point eyes or connection points, such that the score can be easily determined. The win/loss evaluation is modified by a small bonus that favours shorter playouts and terminal posi-

tions with a larger score. This not only plays more human-like moves by preferring wins with fewer moves and maximizing the score, even if all moves are losses. It also contributes with a small amount to the playing strength, because playouts with long sequences or wins with a small score have a larger error.

The evaluation in the in-tree phase is more difficult, because pass moves are always generated here to avoid losing sekis in zugzwang situations. A terminal position after two passes in the in-tree phase will usually contain dead blocks, but the search does not have information about the status of blocks. Therefore, the score is determined using Tromp-Taylor rules: every block is considered to be alive. Together with the additional requirements that the two passes are both played in the search, this will still generate the best move if Chinese rules are used, in which dead blocks may remain on the board, because the Tromp-Taylor score is a lower bound to the Chinese score. The player to move therefore will only generate a pass move if he can afford that the opponent terminates the game by also playing a pass, and the position is evaluated with Tromp-Taylor.

3.2 The player

The player class of the Go engine uses the full-board Monte-Carlo tree search to generate moves and provides additional functionality on top of it. The player sets search parameters depending of the current board size, because some parameters, for example the ones defining the RAVE weighting function, have different optimal values on different board sizes.

If pondering is enabled, the player will also perform a search on the current position while waiting for the opponent to move. The next search for the player will then be initialized with the reusable subtree of the pondering search. To a lesser extent, reusing the subtree can also be useful if pondering is disabled, because a subtree of the last regular search of the player can be used.

The player aborts the search early if the value of the root node is close to a win, and performs additional searches to check if the position is still a win after passing and the status of all points on the board is determined. For determining the status of points, the search can compute statistics for the ownership of points averaged over all terminal positions of the search. If passing seems to win with high probability, the player passes immediately. This avoids the continuation of play in clearly won positions and losing points if Japanese rules are used, in which playing moves in safe territory can cost score points. This works well in most games, but is not a full implementation of Japanese rules. The statistics for the ownership of points are also used for the implementation of the standard GTP commands for querying the final score or final status of blocks.

3.3 Development and debugging

Fuego provides a number of GTP commands that are compatible with GoGui's generic analyze commands [8]. These commands have defined response types and the response can be displayed graphically on the board. This allows a visualization of the engine state or details of the search results, which helps development and debugging. Fuego can also visualize the search dynamically by outputting information to the standard error stream, which uses GoGui's LiveGfx syntax. For example, the visit counts of the children of the root node or the current main variation of the search can be displayed on the board using a configurable interval of simulations.

Many parameters of the search can be changed at runtime with GTP commands. This makes it easy to experiment with different search parameters in a given position or to set up experiments to tune the parameters. The GTP commands for setting parameters are compatible with GoGui's parameter command type, such that GoGui can automatically create dialogs to edit these parameters in the graphical user interface. Figure 3 shows Fuego running in GoGui with a dialog for setting search parameters.

The search tree of a search and the history of all games played in the simulations can be saved in SGF format. The saved search tree contains the position and RAVE counts and values as SGF label properties or as text information in comment properties.

3.4 Performance

3.4.1 CGOS

Fuego has been playing on the Computer Go Server (CGOS) [5] since July 2007. CGOS is a game playing server for Computer Go programs, which computes an incremental Elo rating of the programs, as well as a Bayes Elo rating based on the complete history of all games played.

Figure 4 shows the performance of Fuego up to the release of version 0.3 in December 2008. The data is based on the Bayes Elo computation of CGOS from January 15 2009 (16:03 UCT for 19×19; 18:19 UCT for 9×9). Version 0.3 running on eight cores (Intel Xeon E5420 2.5 GHz dual quadcore system) in lock-free mode was ranked as the 3rd-strongest program ever on 9×9 with a rating of 2664 Elo. The two top-ranked programs are versions of MoGo running on multinode supercomputer hardware. On 19×19 Fuego was the 2nd-strongest program ever with a rating of 2290 Elo[1].

[1] At the time this report was updated, April 30, 2009, the program Zen is also ranked above Fuego

3.4.2 Tournaments

Fuego has achieved the following results in competitions:

- 1st place, KGS 9×9 tournament, December 7, 2008. 4 participants, 12 rounds, 11 wins, 1 loss.
- 4th place, 13th International Computer Games Championship, 19×19, October 1, 2008. 13 participants, round robin, 8 wins, 4 losses.
- 4th place, 13th International Computer Games Championship, 9×9, October 1, 2008. 18 participants, 9 double rounds, 11 wins, 7 losses.
- 5th place, KGS tournament, September 14, 2008. 8 participants, 8 rounds, 4 wins, 4 losses.
- 6th place, 9×9 Computer Go Tournament, European Go Congress, August 2008. 8 participants, 5 rounds, 2 wins, 3 losses.
- 7th place, 19×19 Computer Go Tournament, European Go Congress, August 2008. 8 participants, 6 rounds, 2 wins, 3 losses, 1 round missed due to technical problems.

3.4.3 Play against Humans

Fuego has been tested in play on KGS against humans. It has achieved a rating of 2 kyu in 19×19 games. 9×9 games are not rated on KGS, but in informal games, Fuego has beaten several high-dan amateur players on even.

3.4.4 Regression tests

The Fuego project contains several regression test suites which can be run with the *gogui-regress* tool [8]. Main topic are blunders from games played, and tests for the UCT and static safety of territory modules. For details, please see `http://fuego.svn.sourceforge.net/viewvc/fuego/trunk/regression/`.

3.4.5 Performance tests

Figure 5 shows the performance on 9×9 against GNU Go version 3.6 depending on the number of simulated games per move. No opening book was used for the experiment. On an Intel Xeon E5420 2.5 GHz, Fuego achieves about 11000 simulations per second per core on an empty 9×9 board and about 2750 simulations on an empty 19×19 board.

on both board sizes, and Crazy Stone is now ranked ahead of Fuego on 9×9 as well

4 Future work / outlook

Fuego is a work in progress. Some of the bigger goals for future improvements include:

- Grow the user community and increase the number of developers.
- Develop a massively parallel version.
- Apply Monte-Carlo tree search to other games such as Amazons or Dots and Boxes.
- Establish Fuego as a top program on 19×19.

Acknowledgements

A large number of people have contributed to the Fuego project: Ken Chen and Anders Kierulf developed the Smart Game Board and Explorer, parts of which still survive in the Fuego framework in some form. Contributors at the University of Alberta include Broderick Arneson, Adi Botea, Tim Furtak, Akihiro Kishimoto, Xiaozhen Niu, David Silver, David Tom and Ling Zhao. Ben Lambrechts created and maintains the port to Windows. DARPA, iCORE and NSERC provided financial support. Finally, thanks to the members of the worldwide computer Go community for sharing their ideas, encouragement and friendly rivalry.

References

[1] B. Arneson, R. Hayward, and P. Henderson. Wolve wins Hex tournament. `http://www.cs.ualberta.ca/~hayward/papers/rptBeijing.pdf`, 2008. To appear in ICGA Journal.

[2] M. Buro. ProbCut: An effective selective extension of the alpha-beta algorithm. *ICCA Journal*, 18(2):71–76, June 1995.

[3] G. Chaslot, M. Winands, and J. van den Herik. Parallel Monte-Carlo tree search. In *Proceedings of the 6th International Conference on Computer and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 2008.

[4] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In J. van den Herik, P. Ciancarini, and H. Donkers, editors, *Proceedings of the 5th International Conference on Computer and Games*, volume

4630/2007 of *Lecture Notes in Computer Science*, pages 72–83, Turin, Italy, June 2006. Springer.

[5] D. Dailey. Computer Go Server. http://cgos.boardspace.net/, 2008. Date retrieved: January 19, 2009.

[6] D. Dailey. Simple MC reference bot and specification. http://computer-go.org/pipermail/computer-go/2008-October/016624.html, 2008. Date retrieved: April 30, 2009.

[7] B. Dawes. Boost C++ libraries. http://www.boost.org/, 2008. Date retrieved: December 22, 2008. Date last modified: April 23, 2008.

[8] M. Enzenberger. GoGui. http://gogui.sf.net/, 2009. Date retrieved: January 2, 2009.

[9] M. Enzenberger and M. Müller. Fuego homepage. http://fuego.sf.net/, 2008. Date of publication: May 27, 2008. Date retrieved: December 22, 2008.

[10] M. Enzenberger and M. Müller. A lock-free multithreaded Monte-Carlo tree search algorithm, 2009. Accepted for ACG12.

[11] G. Farnebäck. GTP - Go Text Protocol. http://www.lysator.liu.se/~gunnar/gtp/, 2008. Date retrieved: January 2, 2009.

[12] H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In D. Fox and C. Gomes, editors, *AAAI*, pages 259–264. AAAI Press, 2008.

[13] D. Fotland. The Many Faces of Go, version 12. http://www.smart-games.com/manyfaces.html, 2009. Date retrieved: April 1, 2009.

[14] Free Software Foundation. GNU Go. http://www.gnu.org/software/gnugo/, 2009. Date retrieved: January 2, 2009.

[15] S. Gelly. *A Contribution to Reinforcement Learning; Application to Computer-Go*. PhD thesis, Université Paris-Sud, 2007.

[16] S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007.

[17] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo Go, 2006. Technical Report RR-6062.

[18] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 14:253–302, 2001.

[19] A. Hollosi. SGF file format. `http://www.red-bean.com/sgf/`, 2009. Date retrieved: January 2, 2009.

[20] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 3A: System Programming Guide, Part 1*, 2008. Order Number: 253668-029US.

[21] A. Kierulf. *Smart Game Board: a Workbench for Game-Playing Programs, with Go and Othello as Case Studies*. PhD thesis, ETH Zürich, 1990.

[22] A. Kierulf, R. Gasser, P. M. Geiser, M. Müller, J. Nievergelt, and C. Wirth. Every interactive system evolves into hyperspace: The case of the Smart Game Board. In H. Maurer, editor, *Hypertext/Hypermedia*, volume 276 of *Informatik-Fachberichte*, pages 174–180. Springer, 1991.

[23] A. Kishimoto. *Correct and Efficient Search Algorithms in the Presence of Repetitions*. PhD thesis, University of Alberta, 2005.

[24] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning (ECML)*, pages 282–293, 2006.

[25] GNU Lesser General Public License. `http://www.gnu.org/licenses/lgpl.html`, 2008. Date of publication: June 29, 2007. Date retrieved: December 22, 2008.

[26] R. Lorentz. Amazons discover Monte-Carlo. In J. van den Herik, X. Xu, Z. Ma, and M. Winands, editors, *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2008.

[27] M. Müller. *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory*. PhD thesis, ETH Zürich, 1995. Diss. ETH Nr. 11.006.

[28] M. Müller. Counting the score: Position evaluation in computer Go. *ICGA Journal*, 25(4):219–228, 2002.

[29] M. Müller and Z. Li. Locally informed global search for sums of combinatorial games. In J. van den Herik, Y. Björnsson, and N. Netanyahu, editors, *Computers and Games: 4th International Conference, CG 2004*, volume 3846 of *Lecture Notes in Computer Science*, pages 273–284, Ramat-Gan, Israel, 2006. Springer.

[30] P. Pogonyshev. Quarry homepage. `http://home.gna.org/quarry/`, 2009. Date retrieved: January 2, 2009.

[31] D. Silver, R. Sutton, and M. Müller. Reinforcement learning of local shape in the game of Go. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1053–1058, Hyderabad, India, 2007. Acceptance rate (oral presentation): 212/1353 = 15.7%.

[32] D. Tom and M. Müller. A study of UCT and its enhancements, 2009. Accepted for ACG12.

[33] N. Wedd. Human-computer Go challenges. `http://www.computer-go.info/h-c/index.html`, 2009. Date retrieved: April 1, 2009.

[34] L. Zhao and M. Müller. Using artificial boundaries in the game of Go. In J. van den Herik, X. Xu, Z. Ma, and M. Winands, editors, *Computer and Games. 6th International Conference*, volume 5131 of *Lecture Notes in Computer Science*, pages 81–91, Beijing, China, 2008. Springer.

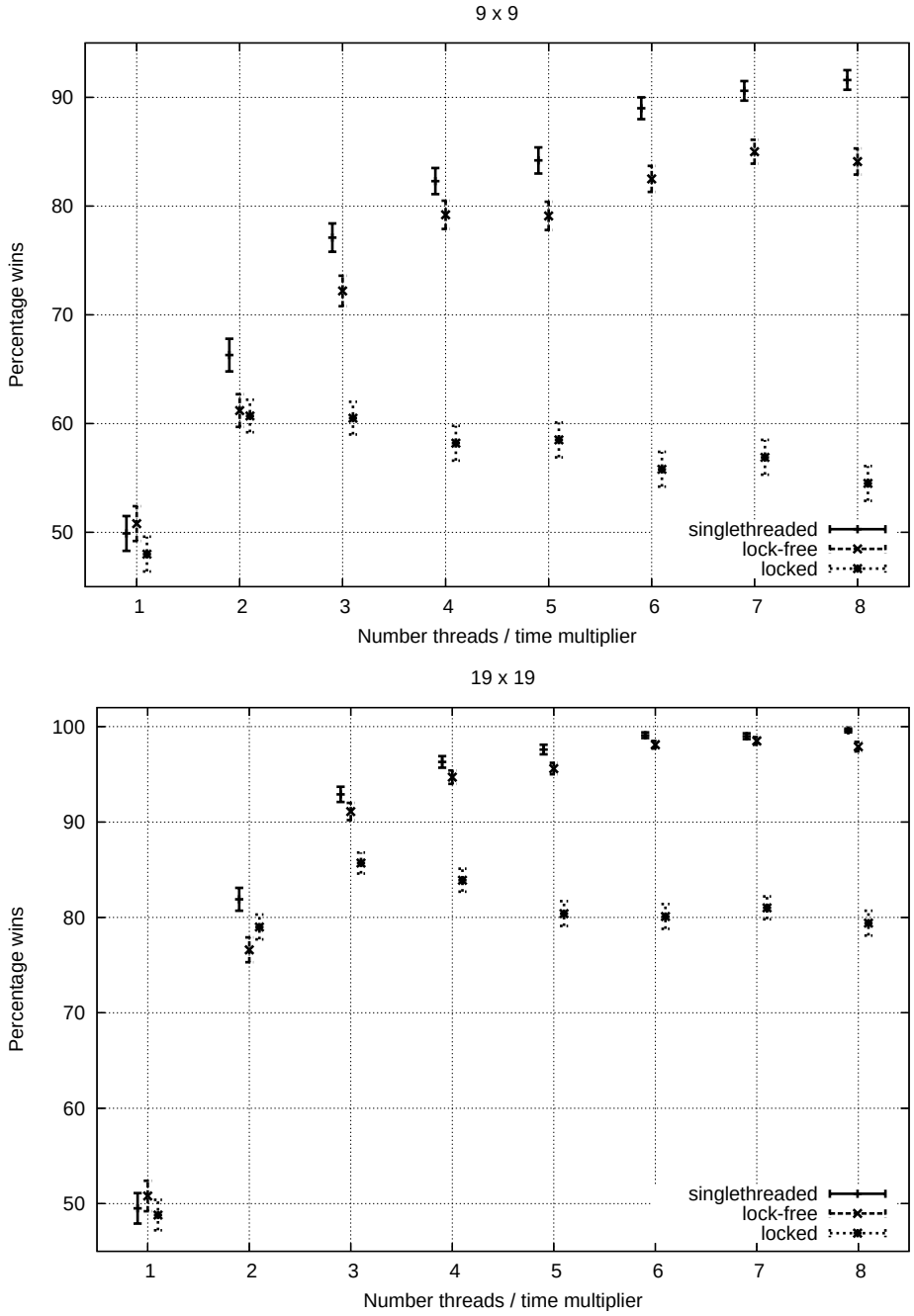

Figure 2: Self-play performance of locked and lock-free multithreading in comparison to a single-threaded search (1 sec per move)

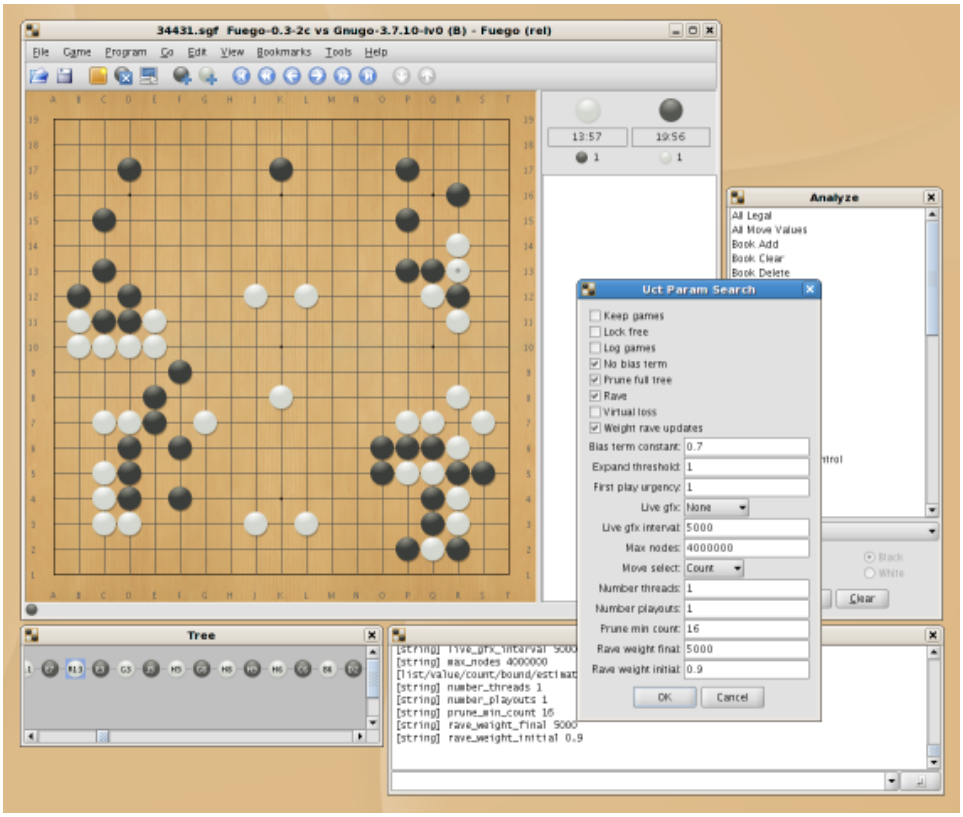

Figure 3: Fuego running in GoGui.

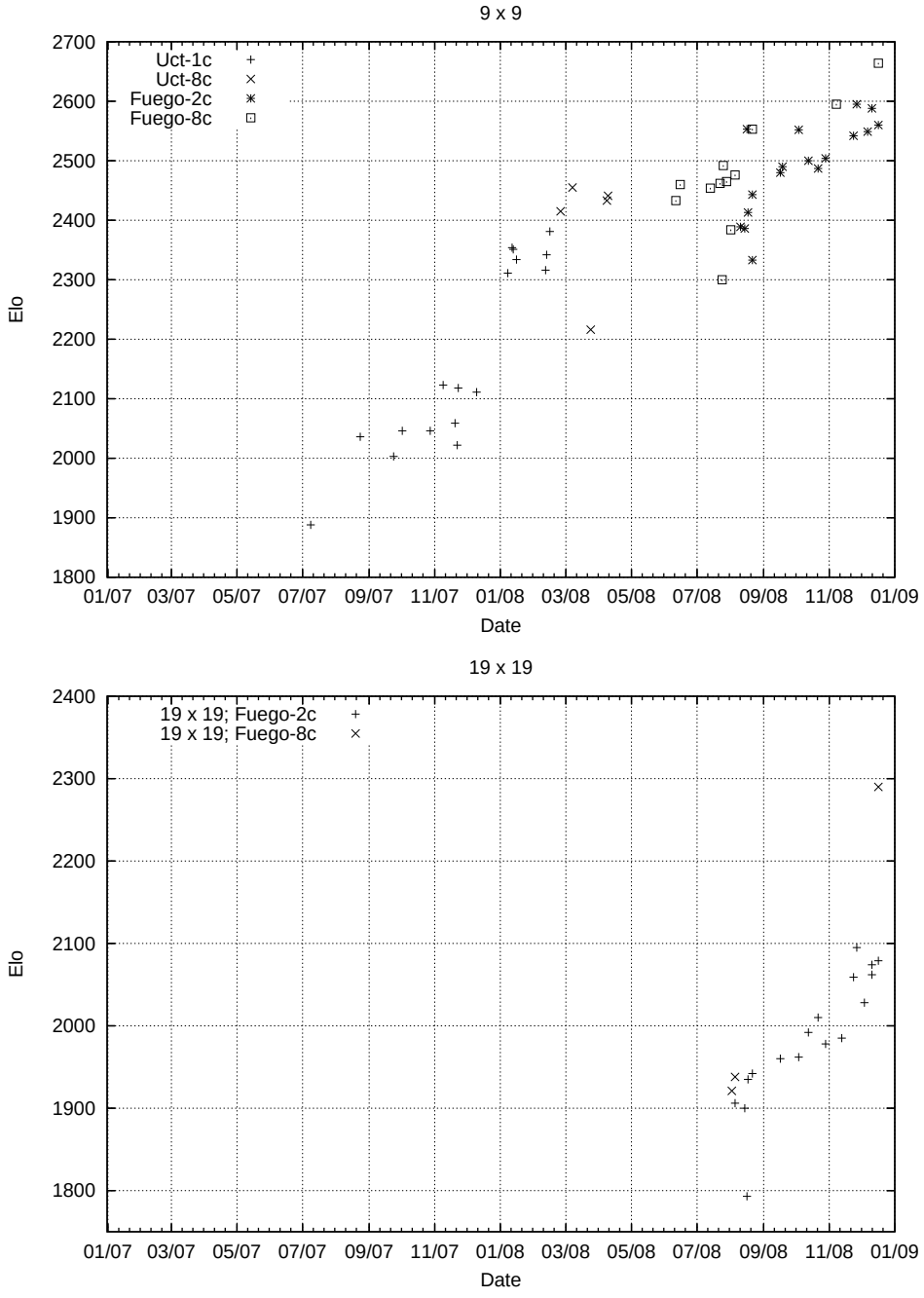

Figure 4: Performance on CGOS until the release of version 0.3. Older versions played under the name Uct, newer versions as Fuego. Before the implementation of multithreading, Fuego used only 1 core (1c), later 2 or 8 cores (2c, 8c).

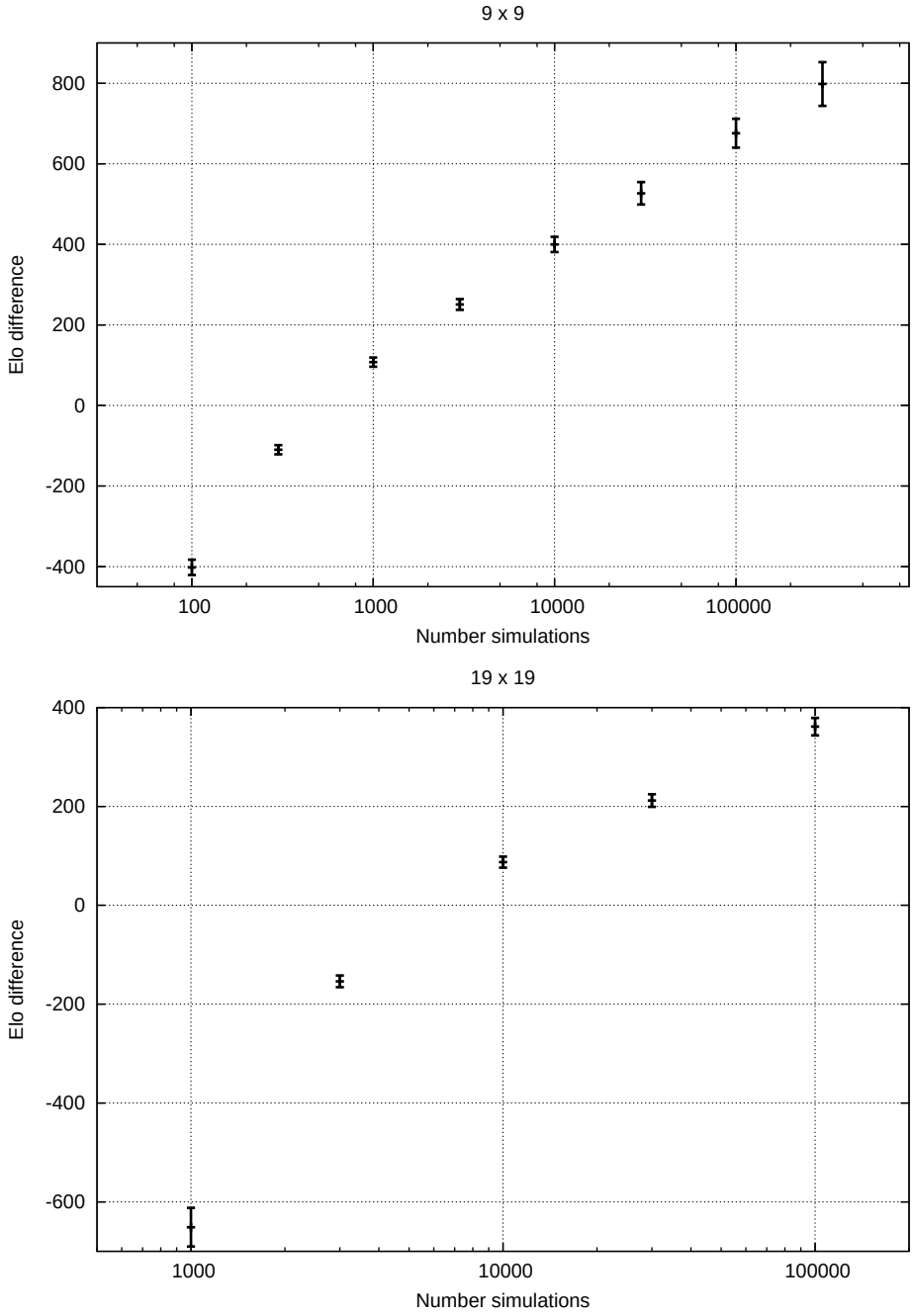

Figure 5: Performance on 9×9 and 19×19 against GNU Go version 3.6 depending on the number of simulated games per move.