FUEGO—An Open-Source Framework for Board Games and *Go* Engine Based on Monte Carlo Tree Search

Markus Enzenberger, Martin Müller, Broderick Arneson, and Richard Segal

***Abstract*—FUEGO is both an open-source software framework and a state-of-the-art program that plays the game of *Go*. The framework supports developing game engines for full-information two-player board games, and is used successfully in a substantial number of projects. The FUEGO Go program became the first program to win a game against a top professional player in 9×9 *Go*. It has won a number of strong tournaments against other programs, and is competitive for 19×19 as well. This paper gives an overview of the development and current state of the FUEGO project. It describes the reusable components of the software framework and specific algorithms used in the *Go* engine.**

***Index Terms*—Computer game playing, *Computer Go*, FUEGO, man-machine matches, Monte Carlo tree search, open source software, software frameworks.**

I. INTRODUCTION

RESEARCH in computing science is driven by the interplay of theory and practice. Advances in theory, such as Monte Carlo tree search (MCTS) and the upper confidence bounds applied to trees (UCT) algorithm [1], [2], have led to breakthrough performance in *Computer Go*, as pioneered by the programs CRAZY STONE [1] and MOGO [3], [4]. In turn, attempts to improve the playing strength of these programs have led to improved algorithms, such as RAVE and prior knowledge initialization [5]. Another crucial advance has been the development of parallel MCTS algorithms, both for shared and distributed memory architectures [6]–[8]. Recently, the scope of MCTS methods has greatly expanded to include many other games [9]–[13] as well as a number of interesting single-agent applications [14]–[17] and multiplayer games [18].

FUEGO is an open-source software framework for developing game engines for full-information two-player board games, as well as a state-of-the-art *Go* program. 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 that would not be possible otherwise because of the cost of implementing a complete state-of-the-art system. Successful previous examples show the importance of open-source software: GNU GO [19] provided the first open-source *Go* program with a strength approaching that of the best classical programs. It has had a huge impact, attracting dozens of researchers and hundreds of hobbyists. In *Chess*, programs such as GNU CHESS, CRAFTY, and FRUIT have popularized innovative ideas and provided reference implementations. To give one more example, in the field of domain-independent planning, systems with publicly available source code such as Hoffmann's FF [20] have had a similarly massive impact, and have enabled much follow-up research.

FUEGO contains a game-independent, 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, and a flexible and general plug-in architecture for adding domain-specific knowledge in the game tree. The FUEGO framework has been proven in applications to *Go*, *Hex*, *Havannah*, and *Amazons*.

The main innovation of the overall FUEGO framework may lie not in the novelty of any of its specific methods and 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.

Among comparable systems, only an executable is available for MOGO [3], [21] and for commercial programs such as THE MANY FACES OF GO, ZEN, and KCC IGO. The reference MCTS implementations by Dailey [22] and Boon provide a very useful starting point, but are far from a competitive system. Similarly, Lew's LIBEGO [23] provides an extremely fast but limited implementation of MCTS. Recent versions of GNU GO [19] 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. Baudis' PACHI [24] is a strong and relatively new open-source MCTS Go program. All these programs are specialized to play the game of *Go*.

A. Why Focus on MCTS and on Go?

Research in computer *Go* has been revolutionized by MCTS methods, and has seen more progress within the last four years than in the two decades before. Programs are close to perfection on 7×7 and have reached top human level on the 9×9 board. In August 2009, FUEGO became the first program to win an even

Manuscript received April 14, 2010; revised June 28, 2010; accepted September 21, 2010. Date of publication October 14, 2010; date of current version January 19, 2011. This work was supported by the Defense Advanced Research Projects Agency (DARPA), iCORE, the Natural Sciences and Engineering Research Council of Canada (NSERC), IBM, and Jonathan Schaeffer.

M. Enzenberger is an independent software developer.

M. Müller and B. Arneson are with the Department of Computing Science, University of Alberta, Edmonton, AB T6G 2E8 Canada (e-mail: mmueller@cs.ualberta.ca).

R. Segal is with IBM T. J. Watson Research Center, Hawthorne, NY 10532 USA.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2010.2083662

game against a top-level 9 Dan professional player on 9×9. See Section IV-A for details. MoGo, Crazy Stone, and The Many Faces of Go have also achieved a number of successes against professional players on 9×9.

On 19×19, programs are still much weaker than top humans, but they have achieved some success with handicaps of 6–9 stones [25]. The top programs Zen, The Many Faces of Go, and Aya have reached amateur Dan (master) level on 19×19. Many MCTS programs have surpassed the strength of all classical programs. In a four-stone handicap game shown in Section IV-A, Fuego was able to score a lucky win against a strong 6 Dan amateur.

MCTS has greatly improved the state-of-the-art in general game playing (GGP). The first MCTS program, CadiaPlayer by Finnsson and Björnsson, won both the 2007 and 2008 Association for the Advancement of Artificial Intelligence (AAAI) competitions [9]. MCTS has now been adopted by all strong GGP programs.

The MCTS approach is being intensely investigated in several other games as well. Recent games programming conferences are dominated by papers on MCTS and applications. In the games of *Amazons* and *Hex*, MCTS-based programs have surpassed the strength of classical approaches [10], [11]. At the 2008 13th International Computer Games Championship in Beijing, China, the MCTS-based *Amazons* program Invader won a close match against the five time defending champion, the classical alpha–beta searcher 8 Queens Problem, and convincingly defended its title in Pamplona the following year. In *Hex*, the current champion MoHex, built upon Fuego's MCTS implementation, remained unbeaten in Pamplona 2009 [12] after finishing in second place in Beijing 2008 [11]. The recently popular game of *Havannah* is another success story for MCTS [13].

In games where alpha–beta works very well there is naturally less interest in new techniques. A paper on MCTS in Shogi has won the best paper award at the yearly workshop on game programming in Japan, the 2008 Game Playing Workshop (GPW) [26]. An MCTS analysis option is provided in the top *Chess* program Rybka [27].

II. The Fuego Software Framework

A. History

Fuego builds on two previous projects: Kierulf's Smart Game Board [28], [29] and Müller's Explorer [30], [31]. Smart Game Board is a workbench for game-playing programs that has been under development since the mid 1980s. Explorer is a *Go*-playing program built on top of Smart Game Board. Its history goes back to Chen and Kierulf's first version, called Go Explorer, 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, was renamed to Fuego and in May 2008 became an open-source project.

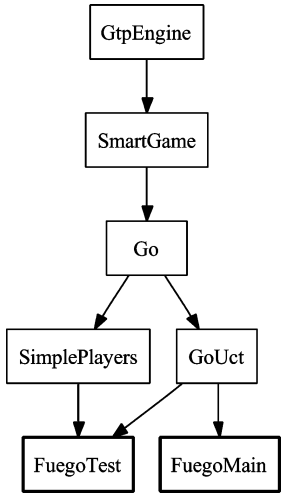

Fig. 1. Libraries and applications of Fuego.

B. Technical Description and Requirements

This paper describes version 0.4.1 of Fuego. File downloads and access to the version control system are available at the SourceForge public open-source hosting service [32]. Fuego is distributed under the terms of the GNU Lesser General Public License [33], version 3 or newer.

Fuego is a software framework. In contrast to a small library with limited functionality and a stable application programming interface (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. Fig. 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 [34], 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 II-D). The default implementation of those classes uses POSIX function calls. No attempt is made to provide the graphical user interface (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 interprocess communication using the Go Text Protocol (GTP).

C. GtpEngine Library

GTP [35] is a text-based protocol for controlling a *Go* engine over a command and response stream, such as the standard input/output (I/O) streams of the process, or network sockets.

It was first used by GNU GO [19] and has gained widespread adoption in the *Computer Go* community. Its many applications include connecting an engine to graphical user interfaces, regression testing, and setting engine parameters through configuration files written in GTP. The protocol has been adapted via game-specific command sets to other games such as *Amazons*, *Othello* [36], *Havannah*, and *Hex*.

FUEGO's 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 keep the engine running in the opponent's time. The pondering function of a game engine is invoked whenever it is waiting for the next command. The base engine sets a flag when the next command has been received to indicate to the game-specific subclass that its 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 nonstandard GTP extension for interrupting commands as used by the GOGUI graphical user interface [37]. The necessary multithreaded implementation is again hidden in the base engine class.

D. *SMARTGAME Library*

The SMARTGAME library contains generally useful game-independent functionality. It includes utility classes, classes that encapsulate nonportable platform-dependent functionality, and classes and functions that help to represent the game state for two-player games on square boards. Further classes represent, load, and save game trees in smart game format (SGF) [38]. The two most complex classes are game-independent implementations of the alpha–beta search algorithm and MCTS. 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.

1) Alpha–Beta Search: SMARTGAME contains a game-independent alpha–beta search engine. It supports standard alpha–beta search with iterative deepening, fractional ply extensions, a transposition table, Probcut [39], and move ordering heuristics such as the history heuristic and preordering the best move from the previous iteration. Tracing functionality can be used to save a search tree as an SGF file, with game-specific text information stored in nodes.

2) Monte Carlo Tree Search: The base MCTS class implements the UCT search algorithm [2] 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 [5], an important enhancement for games such as *Go* that exhibit a high correlation between 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 most moves have only RAVE samples initially, 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 [32], [40]. The mean squared error of the move and RAVE value are modeled with the function $a/N + b$ with the number of samples N and experimentally determined constants a and b. 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. Both move and RAVE values can be initialized by a subclass with game-specific prior knowledge [5].

For efficiency, the tree implementation avoids dynamic memory allocation and stores nodes in a fixed preallocated array. Nodes are never deleted during a search. If the tree runs out of memory, the search can optionally do garbage collection as follows. First, all nodes with a count below a configurable threshold—16 by default—are pruned, then the search resumes, using the reduced tree.

Most search parameters can be changed at runtime to facilitate tuning. The search also collects statistical information such as the average in-tree and total length of simulations, and the number of simulations per second.

3) Multithreading in the MCTS: The search supports parallelization on a single shared memory system using multithreading. Each thread runs the normal MCTS algorithm and has its own game state. All threads share the same game tree; in the default setting, all modifications of the tree are protected by a global mutex. The maximum speedup of this parallelization method is limited 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. Chaslot *et al.* have proposed a more fine-grained locking mechanism that needs a separate mutex for each node in the tree [8], but this did not work well in practice. Usually, a large number of nodes are shared between the in-tree move sequences of different threads; at least the root node is always shared.

FUEGO's lock-free multithreading mode can significantly improve performance [41]. Since this mode depends on hardware support for a specific memory model, it is an optional feature of the base search class that needs to be enabled explicitly. The lock-free implementation does not use any mutexes. Each thread has its own node allocator. If multiple threads expand the same node, only the children created last are used. This causes a small memory overhead. Move and RAVE values are stored in the nodes as counts and incrementally updated mean values, and 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

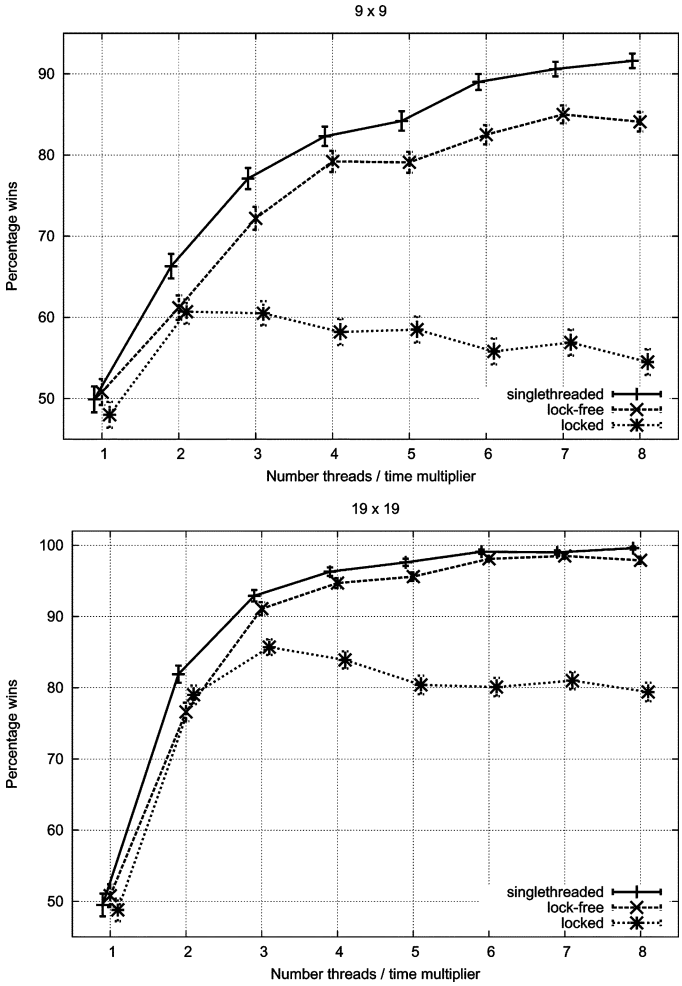

Fig. 2. Self-play performance of locked and lock-free multithreading in comparison to a single-threaded search (1 s/move).

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 architectures guarantee these assumptions and even synchronize central processing unit (CPU) caches after writes [42].

Fig. 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 used the MCTS *Go* engine of FUEGO 0.3 on both 9×9 and 19×19 boards, with 1 s/move, on an Intel Xeon E5420 2.5-GHz dual quad-core system. Unless indicated otherwise, all experiments in this paper use this hardware platform. Each data point shows the percentage of wins in 1000 games against the baseline single-threaded version using 1 s. Locking does not scale beyond two threads on 9×9 and three on 19×19. The lock-free algorithm scaled up to seven threads in this experiment. Preliminary results on a 16 core 2.7-GHz AMD system suggest that the program continues to improve with more cores, though the gain is marginal in the case of 9×9. The measured win rate of 16 versus 8 threads was $69.4\% \pm 1.5\%$ on 19×19 and $54.4\% \pm 1.6\%$ on 9×9.

Other optional features can improve the multithreaded performance depending on the game and the number of threads.

- The *virtual loss* technique [8] is very effective for *Go* in FUEGO. It reduces the chance that different threads explore the same or very similar in-tree sequences.
- Using more than one playout per simulation decreases the percentage of time that is spent in the in-tree phase in each simulation. It is currently not used in *Go*.

4) External and Persistent Knowledge in Lockfree MCTS: Only a very small fraction of nodes in an MCTS receive a large number of simulations [43]. Therefore, relatively expensive external knowledge can be applied to such "heavy" nodes. The additional knowledge can be used to prune moves from further consideration, add new moves in a selective search, or modify the values of existing moves.

FUEGO offers a very general plug-in architecture for applying external knowledge. Multiple sources of knowledge can be applied at different thresholds for number of simulations in a node. In multithreaded search, care is taken to avoid that several threads concurrently compute the same expensive knowledge. Each node stores the threshold at which it last computed knowledge. When a thread discovers that a node needs to compute knowledge, it immediately updates this value. While this does not completely eliminate duplicate work, it works very well in practice.

The current FUEGO GO engine does not use the external knowledge mechanism. It is the main mechanism used in FUEGOEX (see Section II-H).

FUEGO's MCTS engine also provides a mechanism to apply *persistent knowledge* to parts of a game tree. Such knowledge remains true starting from a node N until the end of the game. An example from the game of *Hex* are *dead cells* which can be assigned to a specific player without affecting the game-theoretic value of the game. If a move m is pruned by using persistent knowledge, then the subtrees below the remaining children of N are pruned. Since m can appear in many locations under these children, exploring the entire subtree and removing each node individually would be expensive. The world champion *Hex* program MOHEX uses this feature extensively, to great effect [12]. It is currently not used by the *Go* engines, but could be helpful for an endgame solver that stores persistent safe territories.

E. Go Library

The *Go* library builds on SMARTGAME and provides fairly standard game-specific functionality. 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 can check moves for legality. The commonly used ko rules—simple ko, situational superko, and positional superko—are supported. Different game-ending conventions can be set, such as whether to remove dead stones, and the number of passes required to end a game.

Among others, the *Go* library contains a base class that implements generally useful *Go*-specific GTP commands. Its two methods for detecting safe stones and territories are Benson's

widely used algorithm for unconditional life [44] and Müller's extension for life under alternating play [30].

F. Opening Book

The opening book class has functions to read opening positions from a data file in a simple text format, and match the book against the current position. It handles rotations and mirroring of the position. FUEGO contains one such data file with both 9×9 and 19×19 opening books, plus a few opening moves for 5×5 and 7×7. While the small board books are hand-built, the 19×19 book consists about 1300 moves from popular full-board openings extracted from master games by Ben Lambrechts. Larger books are available for an optional download [45].

Older versions of FUEGO used to be infamous for inefficient opening play. In absence of a book move, on 19×19, FUEGO now always occupies the 4–4 star point in an empty corner. This is consistent with the center-oriented playing style of MCTS programs. In self-play, this simple rule yielded $56.9\% \pm 1.4\%$ wins. FUEGO currently does not contain a *joseki* book with local standard sequences.

The design principles of the hand-built 9×9 book 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 can be exploited by opponents to prepare "killer" variations. Most of the book lines are short.

FUEGO's small human-built book is in stark contrast to MOGO's massive machine-built books [46]. 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 contains a large number of traps designed by professional 6 Dan Yu Ping. These traps are positions that are often misplayed by current MCTS programs.

The FUEGO book is grown by analyzing losses caused by weak openings, and by examining winning statistics of matches against other strong programs. Therefore, it is biased towards lines that have occurred in practice, and still has big "holes" in other lines.

The current 9×9 book contains 1296 positions, which get expanded into 10 126 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 remains roughly constant over a large range of 12–48 000 simulations per move. Many weaknesses addressed by the book seem to be systematic weaknesses in FUEGO's opening play that cannot be discovered easily by more search. A different, less favorable comparison test of FUEGO with and without the opening book against GNU GO 3.8 is given in Section IV-E.

G. Other Libraries and Applications

The FUEGO framework further includes the following libraries: SIMPLEPLAYERS is a collection of simple *Go* playing algorithms, for example, a random player and a player that maximizes an influence function by one-ply search. These players can be used for testing and reference purposes.

GOUCT and FUEGOMAIN are a library and an application that contain the main FUEGO GO engine and a GTP interface to it. The main FUEGO GO engine is described in detail in Section III.

FUEGOTEST is a GTP interface to functionality in the SMARTGAME and *Go* libraries. It provides access to the engine via additional GTP commands which need not be included in the main FUEGO engine, for example, for running regression tests. FUEGOTEST also provides an interface for the players in the SIMPLEPLAYERS library.

H. External Applications and Playing Engines That Use FUEGO

The following projects currently use the FUEGO framework.

MOHEX is the world's strongest *Hex* program [12]. It uses the MCTS framework with persistent knowledge from the current release of FUEGO, along with SMARTGAME utilities such as reading and writing SGF files, random number generation, and timers/clocks.

EXPLORER is a classical knowledge- and local search-based *Go* program [30], [31]. The FUEGO modules GTPENGINE, SMARTGAME, *Go*, and SIMPLEPLAYERS were originally extracted from EXPLORER and are still used there as well. EXPLORER is kept in sync with the development version of FUEGO.

FUEGOEX combines some of the knowledge and local search-based techniques of EXPLORER with the MCTS of FUEGO. It implements three forms of external knowledge as plug-in to FUEGO's MCTS as discussed in Section II-D4. FUEGOEX uses EXPLORER's tactical searches, bad move filters, and patterns to prune and adjust the values for nodes above a threshold of about 200–250 simulations. A detailed discussion of FUEGOEX is beyond the scope of this paper.

BLUEFUEGO is a massively parallel version of FUEGO developed by Richard Segal at IBM (Yorktown Heights, NY). It uses message passing interface (MPI) to share playout information between nodes on a cluster. It has been used successfully in the Pamplona 2009 tournament as well as in the man-machine matches described below. Similarly, BLUEFUEGOEX is a distributed version of FUEGOEX. The design and performance of BLUEFUEGO will be discussed in Section III-D.

SRI has a research team working on a Defense Advanced Research Projects Agency (DARPA, Arlington, VA)-funded *Go* seedling project. Their research explores machine learning techniques for learning new patterns during self-play to improve playing strength, as well as testing an influence function designed by Thomas Wolf. FUEGO 0.2.1 is used as a base to insert/interface their own players and test against other simple players in FUEGO as well as connecting to the *Computer Go* server (CGOS).

TSUMEGO EXPLORER is the world's best life and death solver for enclosed areas [47]. It uses the SMARTGAME and GO libraries. Its code base is kept up to date with FUEGO.

RLGO is a reinforcement learning *Go* program [40] 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 [48]. It uses the classical alpha–beta search and other basic functionality from the

SMARTGAME module. ARROW is kept in sync with the FUEGO development version.

SOS implements a simple abstract sum-of-switches game and is used for experiments with MCTS and RAVE [49]. It currently uses the UCT search framework from FUEGO 0.3.

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 [50], adversarial planning [51], and MCTS in *Amazons* and *Havannah*. Further projects involving FUEGO are currently under way at universities such as Tokyo Institute of Technology, Tokyo, Japan, and the University of California, Santa Cruz.

III. THE FUEGO GO ENGINE

The GOUCT library contains the main *Go* engine of FUEGO, which performs full-board MCTS. The library contains classes that can be reused for other MCTS applications in *Go*, for example, local searches [51]. An extended *Go*-specific GTP engine interfaces to the main FUEGO player, implements commands for setting player and search parameters, and allows querying information about the search. The FUEGOMAIN application contains the top-level main function, which runs the GTP engine over standard I/O streams.

A. Full-Board MCTS

The *Go*-specific MCTS class extends the SMARTGAME MCTS class by providing a *Go*-specific game state. The game state uses the regular *Go* board from the GO library in the in-tree phase, and a specialized *Go* board for the playout phase. The playout board is speed optimized at the cost of reduced functionality: it still updates information about stones and liberties of blocks incrementally. It does not support undoing moves, and supports only the simple ko rule. For code reuse, many board functions are C++ templates taking the board class as a template parameter.

The full-board MCTS class further extends the *Go*-specific MCTS class. It implements a playout policy and prior knowledge.

1) Playout Policy: FUEGO's default playout policy has evolved from the one originally used by MOGO [3]. At the highest priority levels, capture moves, atari-defense moves, and moves matching a small set of hand-selected 3×3 "MOGO" patterns are chosen if they are near the last move played on the board. FUEGO-specific enhancements include a move generator for two-liberty blocks and pattern matching near the second last move. If no move has been selected so far, a global capture move is attempted next. Finally, a move is selected randomly among all legal moves on the board.

The policy for two-liberty blocks is applied both to the last opponent move and to adjacent blocks of the player. It generates moves on *good liberties*, which are points that would gain liberties for that block and are not self-atari. However, moves within *simple chains* such as bamboo and diagonal connections are skipped.

A *move replacement policy* similar to the one in CRAZY STONE [1] attempts to move tactically bad moves to an adjacent point or a better liberty. Moves in one-point eyes are suppressed unless an adjacent stone is in atari. A relatively recent addition to FUEGO's playout policy is a *balancing* rule in conjunction with preventing mutual atari/self-atari moves which would otherwise destroy coexistence (seki) situations. Balancing means that the rule has to be used roughly equally often by both players. This combination achieved about 52% wins in self-play testing.

During a playout, a pass move is generated only if no other moves were produced. A simulation ends either by two consecutive passes, or through a *mercy rule* when the difference in number of captured stones exceeds 30% of the board.

A number of further heuristics is implemented in FUEGO but disabled because of negative test results: MOGO-style rules for nakade and fillboard, as well as several heuristics to avoid bad shape " clumps" of stones.

2) In-Tree Child Selection: Child selection in the in-tree phase of MCTS primarily uses the combined mean and RAVE estimator described above, with the following enhancements.

- **Prior knowledge** [5] initializes moves in the tree with a number of "virtual" wins and losses. Candidate moves from the playout policy receive a boost. Additional bonuses are given globally to moves matching the 3×3 patterns of the playout policy, moves that put the opponent into atari, and moves in the neighborhood of the last move. Self-atari moves are penalized. The number of virtual wins and losses depends on both the move class and the board size.
- A **move filter** completely prunes 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, and can use code that is not thread safe. Currently, unsuccessful ladder-defense moves are pruned in this way, because the result of these ladders is unlikely to become visible within the search tree on large board sizes. Moves on the first line of the board are pruned if there is no stone close by.
- The **SkipRave** heuristic addresses the problem that RAVE effectively prunes moves that are good if played right away, but very bad "on average" if played later. In the child selection computation, one in every $N = 20$ calls ignores RAVE and selects the move based only on the mean value. A 53% winning rate was observed in self-play testing.

3) Evaluation of Terminal Positions: The evaluation of terminal positions in the playout phase is simple, because a pass is only generated when no other move is available. After two consecutive pass moves, the position is assumed to contain only alive blocks and single-point eyes or connection points. Therefore, the score can be easily determined using Tromp–Taylor rules. The win/loss evaluation is modified by small bonuses that favor shorter playouts and terminal positions with a larger score. This plays more human-like moves by preferring wins with fewer moves, and by maximizing the score even if moves are all wins or all losses. It also contributes with a small amount to the playing strength. Playouts with long sequences and wins with a small score have a larger error. Preserving a larger advantage increases the margin of safety against errors later in the game.

Evaluation is more difficult in the in-tree phase, 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 often contains dead blocks. However, 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 of a territory is a lower bound to its Chinese score. The player to move will only generate a pass move if the game is a win in case the opponent terminates the game by also playing a pass, and the resulting "final" position is evaluated with Tromp–Taylor.

B. The Player

The player class of the *Go* engine uses full-board MCTS to generate moves and provides additional functionality. The player sets *search parameters*, which may vary depending on the current board size, such as the ones defining the RAVE weighting function.

By default, the player chooses the *most simulated move* at the root. A few other rules such as highest mean value are also implemented.

If *pondering* is enabled, the player searches the current position while waiting for the opponent to move. The next search of the player can be initialized with the *reusable subtree* of the most recent search, either from pondering or from the last regular search.

With the optional *early pass* feature, the player aborts the search early if the value of the root node is close to a win. In this case, it performs additional searches to check if the position is still a safe win even after passing, and if the status of all points on the board is determined. Determining the status of points uses an optional feature of FUEGO, where the search computes *ownership statistics* for each point on the board, averaged over all terminal positions in the playouts. If passing seems to win with high probability, the player passes immediately. This avoids the continuation of play in clearly won positions and avoids losing points by playing moves in safe territory under Japanese rules. This works relatively well, but is not a full implementation of Japanese rules since playouts are not adapted to that rule set. Point ownership statistics are also used to implement the standard GTP commands for querying the final score and the final status of blocks.

C. Development and Debugging

FUEGO provides a number of GTP commands that are compatible with GOGUI's generic analyze commands [37]. These defined response types of such commands allow the response to be displayed graphically on the GOGUI board. This allows a visualization of the engine state or details of the search results, which helps development and debugging. FUEGO can *visualize the search dynamically* by writing information in GOGUI's LIVEGFX syntax to the standard error stream. Data such as visit counts of the children of the root node, the current main variation of the search, and point ownership can be sent and are regularly updated on GOGUI's display.

Many search parameters can be changed at runtime with GTP commands. This makes it easy to experiment with different search parameters in a given position, and to set up parameter tuning experiments. FUEGO can be started up with a *configuration file* that contains an arbitrary sequence of GTP commands. Such files are also used to configure the engine for tournament play, by setting parameters such as number of threads and memory. Since the GTP commands for setting parameters are compatible with GOGUI's parameter command type, GOGUI can automatically create edit dialogs for them. Fig. 3 shows FUEGO running in GOGUI with a dialog for setting search parameters.

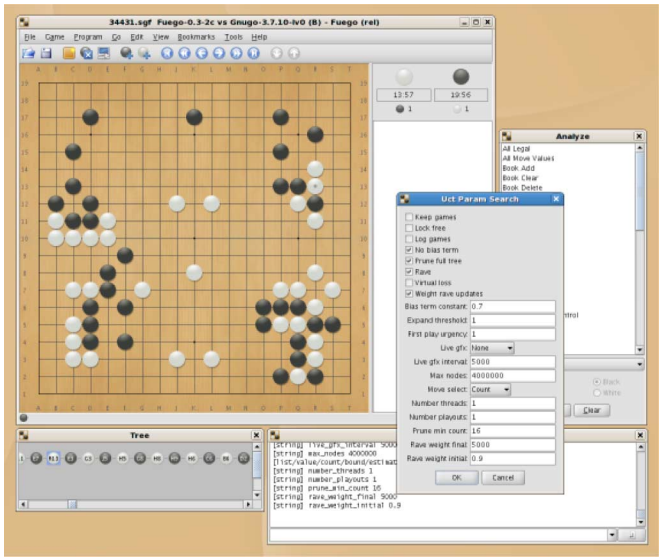

Fig. 3. FUEGO running in GOGUI.

The search tree 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.

D. BLUEFUEGO

BLUEFUEGO is an extension of FUEGO developed at IBM. It uses the standard MPI library to support multimachine clusters. Distributed parallelism is achieved using a new synchronization algorithm called *UCT results synchronization*. Gelly *et al.* [6] considered and rejected *full results synchronization* due to its inability to scale to large clusters. Full results synchronization simulates a large-shared memory machine by broadcasting the results of each UCT trial to all machines in a cluster. On a cluster with N machines, each processor must handle $N-1$ updates for every one of its own. Since the cost of each update for 19×19 FUEGO is about 10% of the total cost of a trial, full results synchronization cannot scale to more than ten machines and would likely scale to far fewer machines due to the increased communication cost.

UCT results synchronization takes advantage of the fact that the cost of updating the UCT tree is dominated by the cost of processing RAVE data. In UCT results synchronization, only updates to the UCT statistics are broadcast. This substantially reduces the cost of processing remote updates and thereby makes scaling to moderately sized clusters possible. Fig. 4 compares the scaling of BLUEFUEGO's UCT results synchronization algorithm with MOGO's tree synchronization

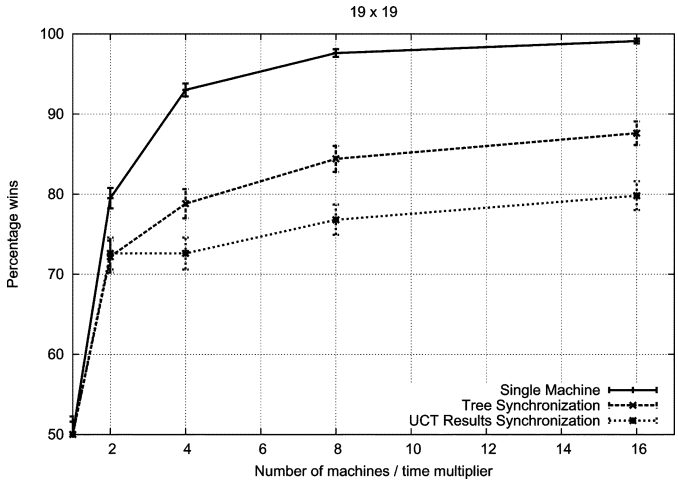

Fig. 4. Scaling of BLUEFUEGO on 60-min 19 × 19 self-play games.

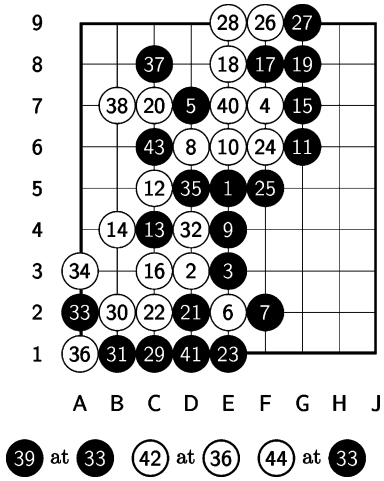

Fig. 5. FUEGO (white) versus Zhou Junxun, professional 9 Dan. White wins by 2.5 points.

algorithm and a single machine with n times more time. The data for each point were collected by running 500 19×19 self-play games on a Blue Gene/P super computer where each player was given 60 min for all moves. The results show that UCT results synchronization provides positive scaling up to at least 16 distributed machines. A separate experiment showed a 72% winning rate over 50 games for BLUEFUEGO running on a cluster of 8×8 core Intel machines against FUEGO running on a single 16 core machine in 9×9 self-play games. These results show that UCT results synchronization can be effective for both 19×19 and 9×9 play.

However, Fig. 4 also shows that UCT results synchronization is outperformed by MOGO's tree synchronization algorithm when running on four or more machines. UCT results synchronization does not appear to offer any advantage over tree synchronization and therefore will likely be replaced in future versions of BLUEFUEGO. Both algorithms with n machines fall far short of the performance of a single machine given n times more time. This suggests that further research on efficient parallel MCTS is needed.

IV. PERFORMANCE

The *Go* program FUEGO is evaluated both in play against humans, and in matches against other programs.

A. Play Against Humans

An official Human vs. Computer Program Competition was held August 21–22, 2009 on Jeju Island, Korea, as part of the 2009 IEEE International Conference on Fuzzy Systems. Four programs, ZEN, MOGO, THE MANY FACES OF GO, and FUEGO, were invited to this event. FUEGO played by using BLUEFUEGO over a cluster of ten nodes. Each node was an 8-core shared memory machine running one copy of FUEGO. For the 19×19 game, an early experimental version of BLUEFUEGOEX was used.

BLUEFUEGO/BLUEFUEGOEX played a total of three official games. Two were 9×9 games against the top-ranked professional 9 Dan Zhou Junxun (his name is also transliterated as Chou Chun-Hsun). On August 22, 2009, BLUEFUEGO won its game playing white against Zhou by 2.5 points, and became the first computer program to win a 9×9 *Go* game on even terms against a world top player. Fig. 5 shows the game. According to the experts, Zhou did not make a mistake in this game, but FUEGO's play was also flawless. Fig. 6 plots FUEGO's score estimate (expressed as a winning percentage) over time. FUEGO's successive moves 8 and 10 created a strong position. Zhou realized early on that he was in trouble, and spent much time on the opening, looking for a way to change the flow of the game. In the end, with normal play, FUEGO would win by 0.5 points, but Zhou tried a line with more potential for confusing the program which resulted in a final score of 2.5 for FUEGO. FUEGO's move 40 inside its own territory, eliminating ko threats and making sure white can win the final ko, is the only path to victory. This game represents a milestone for *Computer Go* research: while programs such as MOGO had previously won 9×9 games against lower ranked professional *Go* players, this is the first time that a player of Zhou's caliber, widely regarded to be among the top 20 players in the world, has lost an even 9×9 game to a computer program.

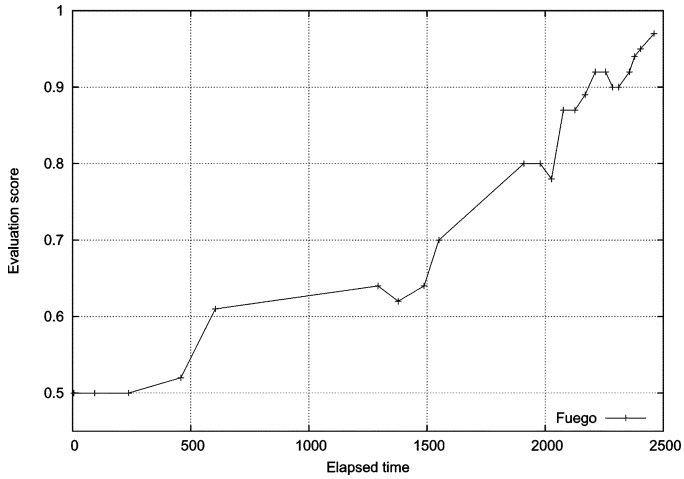

Fig. 6. FUEGO's score estimate versus time in FUEGO (white) versus Zhou Junxun.

BLUEFUEGO easily lost its second game, playing black against Zhou, as shown in Fig. 7. Move 3, played from FUEGO's human-built opening book, was criticized by the experts, and although the program fought well afterwards it did not get any chances in this game.

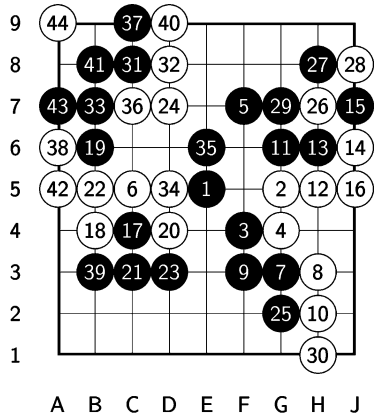

Fig. 7. FUEGO (black) versus Zhou Junxun. White wins by resignation.

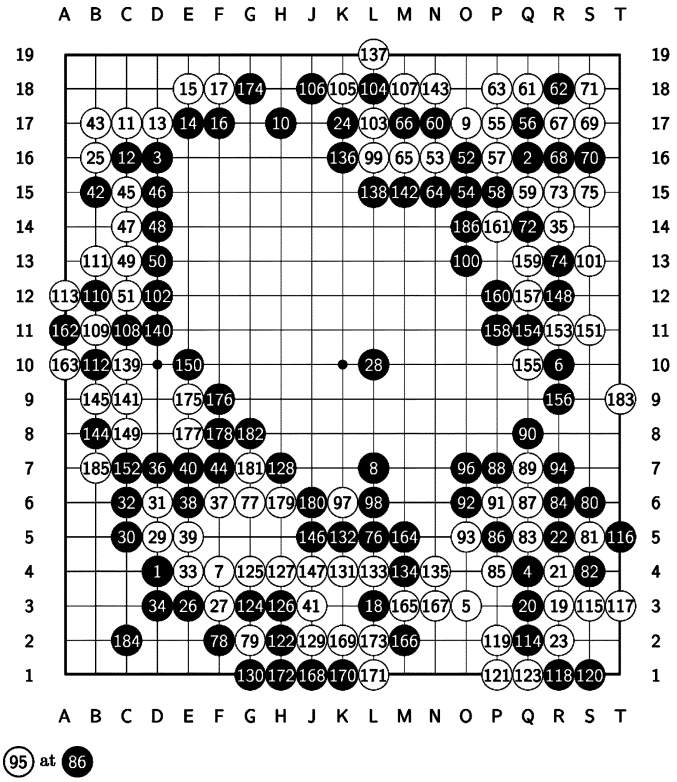

Fig. 8. FUEGO (black, four handicap) versus Chang Shen-Su, amateur 6 Dan. Black wins by resignation.

In the final official game, shown in Fig. 8, BLUEFUEGOEX got an upset win on 19×19 against Chang Shen-Su, a strong 6 Dan amateur player, with only four handicap stones. BLUEFUEGOEX was completely outplayed by Chang in the opening and middlegame. In typical Monte Carlo fashion, FUEGO ignored large losses in the corners and staked everything on a wide open center position. Suddenly, a small mistake by the human in the top center allowed the program to close off a huge center territory on both open sides and thereby win the game.

FUEGO has also been tested in play against a large variety of human players on internet *Go* servers. On the KGS *Go* server, the program has played using the handles `fuego` and `fuego9`. It has achieved a rating of 2 kyu in 19×19 games. 9×9 games are not rated on KGS.

On the turn-based (slow play) online *Go* server (OGS), FUEGO has achieved a rating of 5.7 Dan for 9×9 [52].

B. Selected Tournament Results

FUEGO has achieved the following results in competitions against other programs. In the list, KGS denotes the monthly *KGS Computer Go Tournament*.

- First place: 55th KGS, 9×9, 13 participants, 18 rounds, 16 wins, 2 losses, January 10, 2010.
- Second place: 54th KGS, 9×9, 15 participants, 12 rounds, 9 wins, 3 losses, December 6, 2009.
- First place: 53rd KGS, 19×19, 9 participants, 10 rounds, 9 wins, 1 loss, November 8, 2009.
- First place: 2009 14th Computer Olympiad, 9×9, Pamplona, Spain, 9 participants, 16 rounds double round robin, 15 wins, 1 loss in main tournament, 2 wins against MOGO in playoff, May 11–13, 2009.
- Second place: 2009 14th Computer Olympiad, 19×19, Pamplona, Spain, 6 participants, 10 rounds double round robin, 8 wins, 2 losses, May 14 and 16, 2009.
- First place: 45th KGS, 9×9, 4 participants, 12 rounds, 11 wins, 1 loss, December 7, 2008.
- Fourth place: 13th International Computer Games Championship, 19×19, Beijing, China, October 1, 2008, 13 participants, round robin, 8 wins, 4 losses.
- Fourth place: 13th International Computer Games Championship, 9×9, Beijing, China, October 1, 2008, 18 participants, 9 double rounds, 11 wins, 7 losses.
- Fifth place: 42nd KGS, September 14, 2008. 8 participants, 8 rounds, 4 wins, 4 losses.
- Sixth place: 9×9 *Computer Go* Tournament, European Go Congress, Leksand, Sweden, August 6, 2008. 8 participants, 5 rounds, 2 wins, 3 losses.
- Seventh place: 19×19 *Computer Go* Tournament, European Go Congress, Leksand, Sweden, August 6, 2008. 8 participants, 6 rounds, 2 wins, 3 losses, 1 round missed.

C. FUEGO on CGOS

FUEGO has been playing on CGOS [53] 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.

Fig. 9 shows the performance of FUEGO up to March 2010. The data are based on the Bayes Elo computation of CGOS from March 10, 2010 (16:04 UCT for 19×19; 18:36 UCT for 9×9). Recent FUEGO builds, running on an 8-core system, were ranked as the second-strongest program ever, behind ZEN, and achieved a 2759 top Elo rating on 9×9. The third ranked program on CGOS is MOGO running on multinode supercomputer hardware. The strongest single-core version of FUEGO achieved 2663 Elo, in clear second place behind ZEN among single core versions. Regarding the performance of BLUEFUEGO, only an indirect estimate is currently available. Based on a measured 72% winning rate with BLUEFUEGO on 8×8 cores against FUEGO on a 16 core shared memory machine, and the 54% winning rate of 16 core against 8 core (see above), the Elo rating would be about 2950. However, such self-play experiments tend to exaggerate

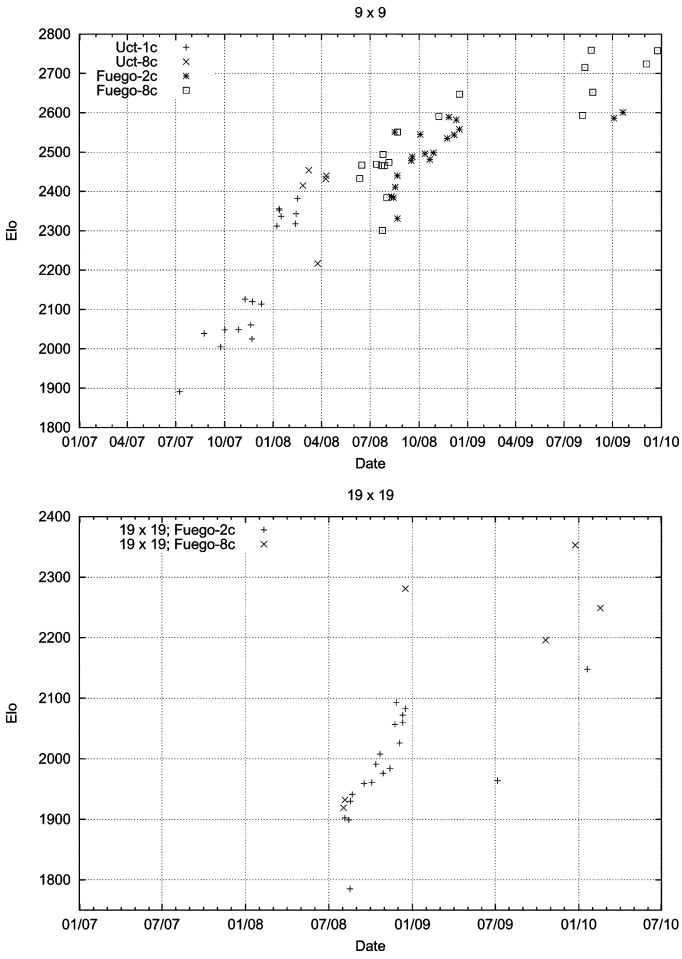

Fig. 9. Performance on CGOS until the release of FUEGO version 0.4.1. Early versions of the program played under the name Uct. 1c, 2c, 8c indicate the number of CPU cores.

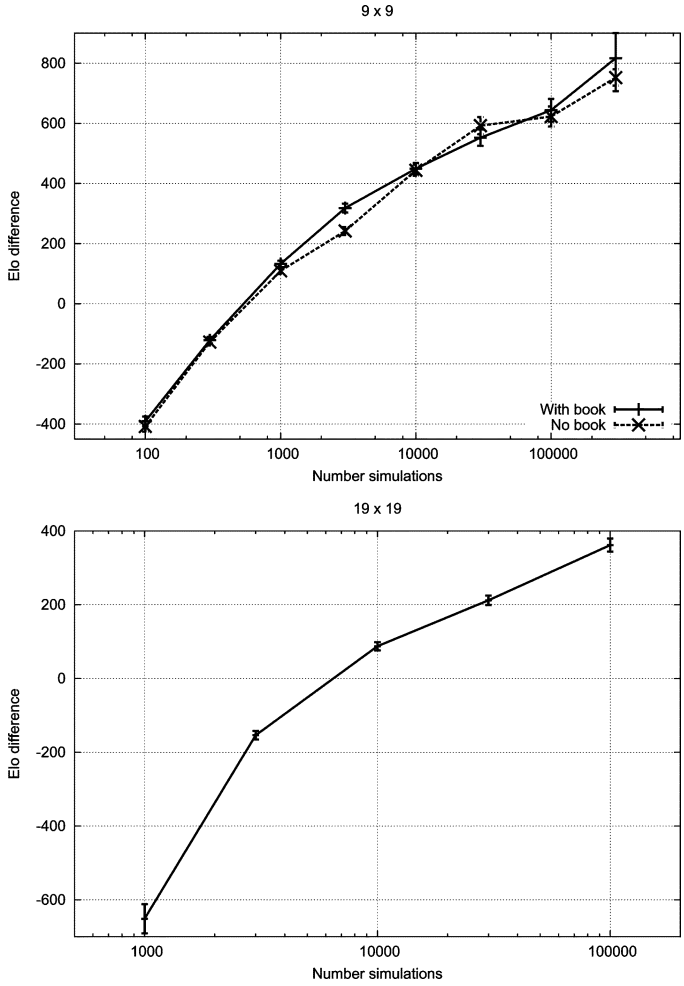

Fig. 10. Performance on 9×9 and 19×19 against GNU GO version 3.8 depending on the number of simulated games per move.

playing strength increases. The performance in man–machine matches is a more solid indicator of performance at this high level where few meaningful comparisons are available.

On 19×19, FUEGO is currently the fifth-ranked program on CGOS with an 8-core rating of 2405 Elo, behind ZEN, THE MANY FACES OF GO, CRAZY STONE, and a multicore version of PACHI. FUEGO's single-core rating is 2148. The performance details of each version including results against individual opponents are archived on the CGOS website [53]. The web page [54] provides access to all this information.

D. Regression and Performance Tests

A different measure of performance is provided by collections of test problems. Such collections cannot measure the overall playing strength, but provide insight into strengths and weaknesses on specialized tasks. The FUEGO project contains a large number of regression and performance test suites which can be run with the *gogui-regress* tool [37]. Some main topics are avoiding blunders played in previous games, and testing MCTS and static safety of territory modules.[1]

[1]For details, please see http://fuego.svn.sourceforge.net/viewvc/fuego/trunk/regression/.

E. Performance Test of FUEGO MCTS Against GNU GO

This test measures the scaling of MCTS in FUEGO SVN revision 1139, a more recent development version with characteristics similar to FUEGO 0.4.1. Fig. 10 shows the performance on 9×9 and 19×19 depending on the number of simulated games per move. The opponent was GNU GO 3.8 level 10 with options `--never-resign--chinese-rules--capture-all-dead`. Two sets of tests were run, with and without opening book. The effect of the book is less pronounced than in self-play but still visible. For reference, on an Intel Xeon E5420 2.5 GHz, FUEGO achieves about 11 000 simulations per second per core on an empty 9×9 board and about 2750 simulations on an empty 19×19 board.

V. FUTURE WORK AND 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 and applications;
- develop a massively parallel version of BLUEFUEGO;
- apply FUEGO's MCTS engine to other games such as *Amazons* or *Havannah*;
- further develop FUEGO on 19×19.

ACKNOWLEDGMENT

A large number of people have contributed to the FUEGO project: A. Kierulf and K. Chen developed the original SMART GAME BOARD and EXPLORER, parts of which still survive in the FUEGO framework in some form. Contributors at the University of Alberta include A. Botea, T. Furtak, A. Kishimoto, X. Niu, A. Rimmel, D. Silver, D. Tom, and L. Zhao. G. Tesauro at IBM was involved in many aspects of this work. B. Lambrechts created and maintains the large opening books as well as the port to Windows. Finally, the authors would like to thank the contributors to the *fuego-devel* mailing list, and the many members of the worldwide computer *Go* community for sharing their ideas, encouragement, and friendly rivalry.

REFERENCES

[1] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Proceedings of the 5th International Conference on Computer and Games*, J. van den Herik, P. Ciancarini, and H. Donkers, Eds. Berlin, Germany: Springer-Verlag, Jun. 2006, vol. 4630/2007, pp. 72–83 [Online]. Available: http://remi.coulom.free.fr/CG2006/CG2006.pdf

[2] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Machine Learning: ECML 2006*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2006, vol. 4212, pp. 282–293.

[3] S. Gelly, "A contribution to reinforcement learning; Application to computer-Go" Ph.D. dissertation, Université Paris-Sud, Paris, France, 2007.

[4] G. Chaslot, C. Fiter, J.-B. Hoock, A. Rimmel, and O. Teytaud, "Adding expert knowledge and exploration in Monte-Carlo tree search," in *Advances in Computer Games*, ser. Lecture Notes in Computer Science, J. van den Herik and P. Spronck, Eds. Berlin, Germany: Springer-Verlag, 2010, vol. 6048, pp. 1–13.

[5] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," in *Proc. 24th Int. Conf. Mach. Learn.*, 2007, pp. 273–280 [Online]. Available: http://dx.doi.org/10.1145/1273496.1273531

[6] S. Gelly, J.-B. Hoock, A. Rimmel, O. Teytaud, and Y. Kalemkarian, "The parallelization of Monte-Carlo planning—Parallelization of MC-planning," in *ICINCO-ICSO*, J. Filipe, J. Andrade-Cetto, and J.-L. Ferrier, Eds. Setubal, Portugal: INSTICC Press, 2008, pp. 244–249.

[7] T. Cazenave and N. Jouandeau, "A parallel Monte-Carlo tree search algorithm," in *Computers and Games*, ser. Lecture Notes in Computer Science, J. van den Herik, X. Xu, Z. Ma, and M. Winands, Eds. Berlin, Germany: Springer-Verlag, 2008, vol. 5131, pp. 72–80.

[8] 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*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2008, vol. 5131, pp. 60–71 [Online]. Available: http://www.cs.unimaas.nl/g.chaslot/papers/parallelMCTS.pdf

[9] Y. Bjornsson and H. Finnsson, "Cadiaplayer: A simulation-based general game player," *IEEE Trans. Comput. Intell. AI Games*, vol. 1, no. 1, pp. 4–15, Mar. 2009.

[10] R. Lorentz, "Amazons discover Monte-Carlo," in *Computers and Games*, ser. Lecture Notes in Computer Science, J. van den Herik, X. Xu, Z. Ma, and M. Winands, Eds. Berlin, Germany: Springer-Verlag, 2008, vol. 5131, pp. 13–24.

[11] B. Arneson, R. Hayward, and P. Henderson, "Wolve wins Hex tournament," *Int. Comput. Games Assoc. J.*, vol. 32, no. 1, pp. 48–53, 2009.

[12] B. Arneson, R. Hayward, and P. Henderson, "Mohex wins Hex tournament," *Int. Comput. Games Assoc. J.*, vol. 32, no. 2, pp. 114–116, 2009.

[13] F. Teytaud and O. Teytaud, "Creating an upper-confidence-tree program for Havannah," in *Advances in Computer Games*, ser. Lecture Notes in Computer Science, J. van den Herik and P. Spronck, Eds. Berlin, Germany: Springer-Verlag, 2010, vol. 6048, pp. 65–74.

[14] F. de Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel, "Bandit-based optimization on graphs with application to library performance tuning," in *ICML*, ser. ACM International Conference Proceedings, A. P. Danyluk, L. Bottou, and M. L. Littman, Eds. New York: ACM, 2009, vol. 382, pp. 92–92.

[15] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, G. Chaslot, and J. W. H. M. Uiterwijk, "Single-player Monte-Carlo tree search," in *Computers and Games*, ser. Lecture Notes in Computer Science, J. van den Herik, X. Xu, Z. Ma, and M. Winands, Eds. Berlin, Germany: Springer-Verlag, 2008, vol. 5131, pp. 1–12.

[16] A. Rimmel, "Bandit-based optimization on graphs with application to library performance tuning," Ph.D. dissertation, Ecole doctorale d'informatique, Université Paris-Sud, Paris, France, 2009.

[17] H. Nakhost and M. Müller, "Monte-Carlo exploration for deterministic planning," in *Proc. 21st Int. Joint Conf. Artif. Intell.*, Pasadena, CA, 2009, pp. 1766–1771.

[18] N. Sturtevant, "An analysis of UCT in multi-player games," in *Computers and Games*, ser. Lecture Notes in Computer Science, J. van den Herik, X. Xu, Z. Ma, and M. Winands, Eds. Berlin, Germany: Springer-Verlag, 2008, vol. 5131, pp. 37–49.

[19] Free Software Foundation, GNU Go, 2009 [Online]. Available: http://www.gnu.org/software/gnugo/

[20] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *J. Artif. Intell. Res.*, vol. 14, pp. 253–302, 2001.

[21] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, "Modification of UCT with patterns in Monte-Carlo Go," 2006 [Online]. Available: http://hal.inria.fr/inria-00117266/en

[22] D. Dailey, "Simple MC reference bot and specification," 2008 [Online]. Available: http://computer-go.org/pipermail/computer-go/2008-October/016624.html

[23] L. Lew, "Library of effective Go routines," 2010 [Online]. Available: http://github.com/lukaszlew/libego

[24] P. Baudis, "Pachi simple Go/baduk/weiqi bot," 2010 [Online]. Available: http://repo.or.cz/w/pachi.git

[25] N. Wedd, "Human-computer Go challenges," 2009 [Online]. Available: http://www.computer-go.info/h-c/index.html

[26] Y. Sato and D. Takahashi, "A Shogi program based on Monte-Carlo tree search," in *Proc. 13th Game Programming Workshop*, 2008, pp. 1–8.

[27] S. Lopez, "Rybka's Monte Carlo analysis," 2008 [Online]. Available: http://www.chessbase.com/newsdetail.asp?newsid=5075

[28] A. Kierulf, "Smart game board: A workbench for game-playing programs, with Go and Othello as case studies" Ph.D. dissertation, Dept. Comput. Sci., ETH Zürich, Zürich, Switzerland, 1990.

[29] A. Kierulf, R. Gasser, P. M. Geiser, M. Müller, J. Nievergelt, and C. Wirth, H. Maurer, Ed., "Every interactive system evolves into hyperspace: The case of the Smart Game Board," *Hypertext/Hypermedia*, vol. 276, pp. 174–180, 1991.

[30] M. Müller, "Computer Go as a sum of local games: An application of combinatorial game theory," Ph.D. dissertation, Dept. Comput. Sci., ETH Zürich, Zürich, Switzerland, 1995.

[31] M. Müller, "Counting the score: Position evaluation in computer Go," *Int. Comput. Games Assoc. J.*, vol. 25, no. 4, pp. 219–228, 2002.

[32] M. Enzenberger and M. Müller, "Fuego homepage," 2008 [Online]. Available: http://fuego.sf.net/

[33] GNU Lesser General Public License, 2008 [Online]. Available: http://www.gnu.org/licenses/lgpl.html

[34] B. Dawes, "Boost C++ libraries," 2008 [Online]. Available: http://www.boost.org/

[35] G. Farnebäck, "GTP—Go text protocol," 2008 [Online]. Available: http://www.lysator.liu.se/~gunnar/gtp/

[36] P. Pogonyshev, "Quarry homepage," 2009 [Online]. Available: http://home.gna.org/quarry/

[37] M. Enzenberger, "GoGui," 2009 [Online]. Available: http://gogui.sf.net/

[38] A. Hollosi, "SGF file format," 2009 [Online]. Available: http://www.red-bean.com/sgf/

[39] M. Buro, "ProbCut: An effective selective extension of the alpha-beta algorithm," *Int. Comput. Chess Assoc. J.*, vol. 18, no. 2, pp. 71–76, Jun. 1995.

[40] D. Silver, "Reinforcement learning and simulation-based search in computer Go," Ph.D. dissertation, Dept. Comput. Sci., Univ. Alberta, Edmonton, AB, Canada, 2009.

[41] M. Enzenberger and M. Müller, "A lock-free multithreaded Monte-Carlo tree search algorithm," in *Advances in Computer Games*, ser. Lecture Notes in Computer Science, J. van den Herik and P. Spronck, Eds. Berlin, Germany: Springer-Verlag, 2010, vol. 6048, pp. 14–20.

[42] Intel Corporation, "Intel 64 and IA-32 architectures software developer's manual—Volume 3A: System programming guide, part 1," Order No. 253668-029US, 2008.

[43] G. Chaslot, M. Winands, J. Uiterwijk, J. van den Herik, and B. Bouzy, "Progressive strategies for Monte-Carlo tree search," *New Math. Natural Comput.*, vol. 4, no. 3, pp. 343–357, 2008.

[44] D. B. Benson, "Life in the game of Go," *Inf. Sci.*, vol. 10, pp. 17–29, 1976.
[45] B. Lambrechts, "Opening books for Fuego," 2010 [Online]. Available: http://gnugo.baduk.org/fuegoob.htm
[46] P. Audouard, G. Chaslot, J.-B. Hoock, A. Rimmel, J. Perez, and O. Teytaud, "Grid coevolution for adaptive simulations; application to the building of opening books in the game of Go," in *Applications of Evolutionary Computing*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2009, vol. 5484, pp. 323–332.
[47] A. Kishimoto, "Correct and efficient search algorithms in the presence of repetitions," Ph.D. dissertation, Dept. Comput. Sci., Univ. Alberta, Edmonton, AB, Canada, 2005.
[48] M. Müller, "Solving 5 × 5 amazons," in *The 6th Game Programming Workshop (GPW 2001)*. Hakone, Japan: Information Processing Society of Japan, 2001, vol. 2001, pp. 64–71.
[49] D. Tom and M. Müller, "A study of UCT and its enhancements in an artificial game," in *Advances in Computer Games*, ser. Lecture Notes in Computer Science, J. van den Herik and P. Spronck, Eds. Berlin, Germany: Springer-Verlag, 2010, vol. 6048, pp. 55–64 [Online]. Available: http://hal.inria.fr/inria-00386477/en/
[50] M. Müller and Z. Li, "Locally informed global search for sums of combinatorial games," in *Computers and Games: 4th International Conference, CG 2004*, ser. Lecture Notes in Computer Science, J. van den Herik, Y. Björnsson, and N. Netanyahu, Eds. Berlin, Germany: Springer-Verlag, 2006, vol. 3846, pp. 273–284.
[51] L. Zhao and M. Müller, "Using artificial boundaries in the game of Go," in *Computer and Games. 6th International Conference*, ser. Lecture Notes in Computer Science, J. van den Herik, X. Xu, Z. Ma, and M. Winands, Eds. Berlin, Germany: Springer-Verlag, 2008, vol. 5131, pp. 81–91.
[52] OGS Online Go Server, "Player profile—Fuego9," × 9-1h 2010 [Online]. Available: http://www.online-go.com/profile.php?user=26504
[53] D. Dailey, "Computer Go server," 2008 [Online]. Available: http://cgos.boardspace.net/
[54] M. Müller, "Fuego—CGOS history," 2010 [Online]. Available: http:// http://webdocs.cs.ualberta.ca/~mmueller/fuego/cgos.html

Markus Enzenberger, photograph and biography not available at the time of publication.

Martin Müller received the Ph.D. degree in computer science from ETH Zurich, Switzerland.

Currently, he is a Professor at the Department of Computing Science, University of Alberta, Edmonton, AB, Canada. He is the leader of the FUEGO project and also acts as the *Go* expert. His research interests are in search algorithms, with applications to *Computer Go* and planning.

Broderick Arneson received the B.Sc. degree in computing science, the B.Sc. degree (honors) in mathematics, and the M.S. degree in computing science from the University of Alberta, Edmonton, Canada, in 2003, 2004, and 2006, respectively.

Currently, he is a Research Associate at the University of Alberta, where he works for the *Go* group.

Richard Segal received the Ph.D. degree in computer science from the University of Washington, Seattle, in 1997.

Currently, he is a member of the research staff at the IBM T. J. Watson Research Center, Hawthorne, NY. His research interests include machine learning, text classification, and parallel algorithms.