MQP CDR#GXS1102

Monte-Carlo Search Algorithms

a Major Qualifying Project Report
submitted to the faculty of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Decree of Bachelor of Science by

____

Chang Liu

____

Andrew D. Tremblay

March 28, 2011

____

Professor Gábor N. Sárközy, Major Advisor

____

Professor Stanley M. Selkow, Co-Advisor

Abstract

We have explored and tested the behavior of Monte-Carlo Search Algorithms in both artificial and real game trees. Complementing the work of previous WPI students, we have expanded the Gomba Testing Framework; a platform for the comparative evaluation of search algorithms in large adversarial game trees. We implemented and analyzed the specific UCT algorithm PoolRAVE by developing and testing variations of it in an existing framework of Go algorithms. We have implemented these algorithm variations in computer Go and verified their relative performances against established algorithms.

Acknowledgments

Levente Kocsis, Project Advisor and SZTAKI Contact

Gábor Sárközy, MQP Advisor

Stanley Selkow, MQP Co-Advisor

Worcester Polytechnic Institute

MTA-SZTAKI

Information Lab, MTA-SZTAKI

And All SZTAKI Colleagues

Contents

Abstract i

Contents iii

Contents iii

List of Figures v

List of Algorithms vi

1 Background 7
1.1 Introduction 7
1.2 Go....8
1.3 Current Search Algorithms 10
1.3.1 Monte-Carlo Tree Search (MCTS)....10
1.3.2 Upper Confidence Bound (UCB) 11
1.3.3 Upper Confidence Tree (UCT) 13
1.3.4 Rapid Action Value Estimation (RAVE) 14
1.3.5 Genetic Programming (GP) 16
1.3.6 Neural Networks (NN) 17
2 Existing Codebases....19
2.1 Gomba 19
2.1.1 Artificial Game Trees....19
2.1.2 Features of Gomba....20
2.1.3 Weakness of Gomba....21
2.2 Fuego....23
2.2.1 Architecture of Fuego....23
3 Gomba 26
3.1 Additions to Gomba....26
3.1.1 Modification of tree generation algorithm 26
3.1.2 Addition to searchData field 26
3.1.3 Correlation and consistency among actions....27
3.1.4 Lazy State Expansion 32
3.2 Experiments in Gomba 34
3.2.1 Comparison of Old and New Gomba Game Tree 34
3.2.2 Gomba tree with different equivalence parameters....38
3.2.3 Gomba Tree with Different Correlation Settings 40
3.2.4 Summary 40
4 Fuego....42
4.1 Additions to Fuego 42
4.1.1 PoolRAVE 42

4.1.2 PoolRAVE(Pass) 45
4.1.3 PoolRAVE(PersistPass) 45
4.1.4 PoolRAVE(SmartPersist)..46
4.2 Experiments in Fuego 46
4.2.1 Basic Fuego vs GnuGo...47
4.2.2 PoolRAVE(Pass) 49
4.2.3 PoolRAVE(PersistPass) 50
4.2.4 PoolRAVE (SmartPersist)...52
4.2.5 Score Correlation between Consecutive Moves 54
4.3 Summary 56
5 Conclusions 57
5.1 The Gomba Testing Framework 57
5.2 Fuego...57
5.3 Future Work 58
6 References 61
Appendix A: Gomba Experiment Parameters...62
Set 1: Comparative on different equivalence parameters..62
Set 2: Comparative on level of correlations...62
Appendix B: Gomba Developer's Primer...64
Using Gomba 64
Example..65
Parsing Results...65
Adding Search Algorithms..66
Appendix C: Fuego Experiment Parameters...68

List of Figures

Figure 1 Outline of a Monte-Carlo Tree Search. 11

Figure 2 Nodes updated using UCT RAVE.. 15

Figure 3 UCT RAVE win rate in the original Gomba framework 21

Figure 4: The Fuego Dependency Tree 24

Figure 5 Algorithm 1 illustration 30

Figure 6 Comparison of lazy state expansions 33

Figure 7 Win rate for old and new Gomba framework 36

Figure 8 Average difficulty for old and new Gomba framework 37

Figure 9 Different Equivalence Parameter Settings 39

Figure 10 UCT RAVE performance using different correlations 40

Figure 11 : Performance Difference Between RAVE(Pool) and Basic RAVE 43

Figure 12 : An Example of a Stagnant Pool 44

Figure 13: Current Fuego performance against varying levels of GnuGo 48

Figure 14 Win Percentage vs. Pool Size (5 sec/move) 51

Figure 15 Win Percentage vs Pool Selection Probability p (5 sec/move) 53

Figure 16: Score Estimate Correlations of Consecutive Moves of a Single Game of Go 55

Figure 17 Winning rate of UCT-RAVE vs UCT. 58

List of Algorithms

Define correlation when generating a new Child Node 28

Modified Lazy State Expansion...32

1 Background

1.1 Introduction

Almost every aspect of the world can be modeled as a sequence of actions and their effects. It is through this model that we can understand our surroundings and what actions to take. From our innate understanding of cause and effect we can extrapolate the conclusions of science and mathematics. By searching through our knowledge of possible events and their branching outcomes we may predict and gauge our actions and future environment with varying degrees of accuracy. It is also this law of cause and effect that allows Artificial Intelligence systems any ability for prediction and behavior.

Through artificial systems of actions and states, many modern artificial intelligence algorithms search for solutions to problems by computing through a series of predictions and assessments. Modern intelligence systems can essentially be considered as search algorithms, though instead of searching for a website on Google an AI search algorithm might try to find the ideal move in a game of Chess or Go. Games are especially interesting in developing AI, as they are provably finite but too large to completely store, making them ideal testing grounds. AI algorithms often use games as benchmarks for performance and springboards into many more practical applications, such as automated car navigation or air traffic control.

The goal of our project is to research and improve current search algorithms in the context of large game trees, specifically within the game of Go. For most conventional problems searches are almost trivial. The search space is loaded and the given object to

find is defined and requested. After traversing the search space the requested object is either found or a confirmation of it not existing in the search space is returned.

Large game trees, on the other hand, are situations that have so many possible actions that one cannot resolve the outcome of every single one. Even visiting all of the actions and assessing them even once can be a challenge for games with sufficiently large branching factors. Also, the objective for a large game tree – finding a winning sequence of moves – is not defined initially, and is only fully defined when (and if) the game concludes with a victory. Such a situation is much different than simply finding a word in a large collection of documents or locating a website through Google. Our situation requires more adaptive and efficient solutions in order to be solved even close to optimally. We consider both artificial game tree systems, like Gomba [1], and actual game tree systems, like Fuego [2], as testing frameworks for our explorations. We have expanded the functionality of both frameworks to implement the latest Monte-Carlo search strategies, and have tested these strategies extensively to show the comparative performance of both artificial and actual game trees as well as the actual performance of these new algorithms with other established ones.

1.2 Go

Go is a board game for two players with more than a 2,000 year history. It is still a popular game around the world today. The rules of Go are simple, two players put stones on the Go board in turn to enlarge their own territory and try to capture the opponent's

stones at the same time. However, it is hard to play well. A good move needs to foresee other future moves, to predict its opponent's possible moves, to interact with stones in distance, to allow tactical loss for a current move, to keep the whole board in mind while fighting locally, and other strategies that involve the overall game.

Victory in a Go game is different from other board games. In a Chess game, it ends when one of the players captures his opponent's king. However in a Go game, no certain move will trigger the game to end immediately. It requires a series of good moves through the game play to earn points and enlarge territory. Usually at the conclusion of a Go game, victory is defined by counting the pieces and its territory for both players. A win by 0.5 or 1 position is very common.

The rules of Go look simple but require rich strategy to play well. For many years Computer Go still could not beat a professional Go player, and none currently can consistently beat one. It remained to be a challenging topic for many Computer Go researchers. The size of a Go board ranged from 9x9 to 19x19, which is much larger than a chess board. The possible moves of Go, or branching factor of a game tree is so large that it is impossible for computers to calculate the best move. The most advanced Computer Go so far can reach the master level only in a 9x9 board [3]. Many new techniques are yet to be discovered in the field of Computer Go.

1.3 Current Search Algorithms

While researching the current landscape of algorithms for assessing large search trees we made every attempt to be as comprehensive as possible. From the most established algorithms (UCT [4]) and their recent variants (UCT-RAVE [5]) to nontraditional approaches (Neural Networks [6] and Genetic Programming [7]), almost all were considered. With each algorithm we also researched any past application to Go game trees specifically. Fortunately, the ubiquity of Go as a performance testing platform led us to Go-based experiments for every algorithm that we found.

1.3.1 Monte-Carlo Tree Search (MCTS)

The idea of using Monte-Carlo algorithms in the context of Computer Go was first proposed by Bernd Brügmann in 1993 [8]. Monte-Carlo Methods are widely used in simulating physical and mathematical systems, which rely on repeated random sampling to compute the result. Brügmann posed the question: "*How would nature play Go?*" [8]. This idea attracted more and more attention after it appeared, and the experimental results on a 9x9 Go board were surprisingly efficient.

Monte-Carlo Tree Search (MCTS) is a best-first search algorithm based on Monte-Carlo methods. The basis idea of MCTS is built on the *playout*. A playout is a fast game of random moves from the start to an end of the game. A win-rate and node visits count statsitics are kept by nodes of a game tree.

The algorithm of MCTS can be divided into separate steps of selection, expansion, simulation, and backpropagation (Fig 1.). After reaching the end of a game play (a leaf node), node visit count and win ratio value is updated along the path. This whole process is repeated numerous times, and the final action chosen is the node which was explored most among all the children nodes.

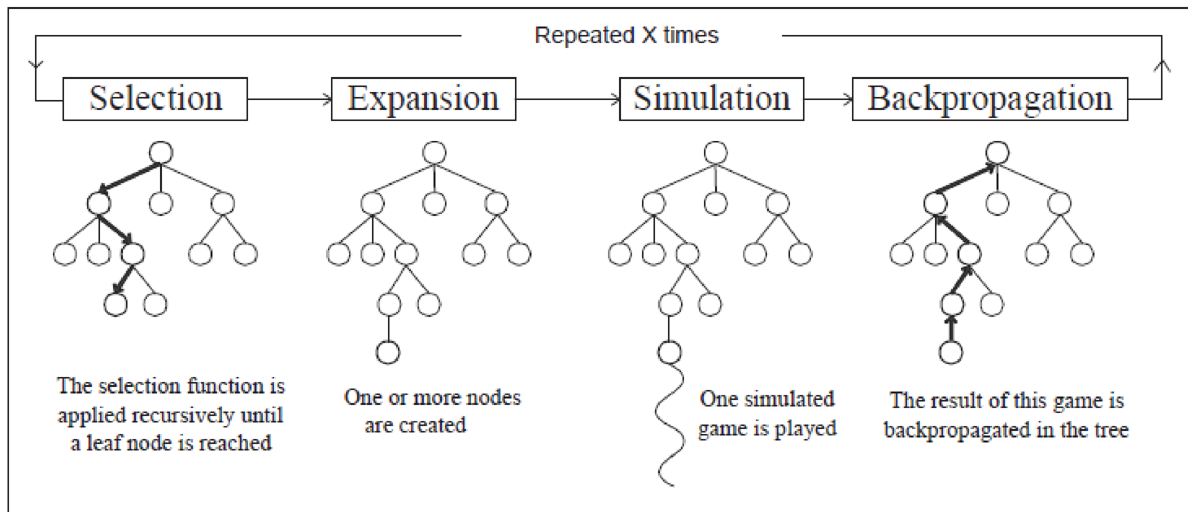


Figure 1 Outline of a Monte-Carlo Tree Search [9].

1.3.2 Upper Confidence Bound (UCB)

The Upper Confidence Bound method is not a tree search method in itself, though the basic principle of it when applied to trees (known as UCT) is the current dominant algorithm in large game trees. The basic model of UCB search is to find the optimal choice in a given set of choices, each with random payoffs, with or without exploring all choices [5]. Also referred to as the "Multi-Armed Bandit" problem, it is similar to a near-infinite row of slot machines, each with a different payout, that you may select in any

order for as many times as possible with the intention of finding the slot machine with the best payout.

UCB uses finite-time regret to keep track of the average of rewards for each of the K visited machines and selects the next slot machine i with the best upper confidence bound, which is a function of the average rewards for that machine plus a hand-tweaked variable $c_{t-1,T_i(t-1)}$ that decays over the number of attempts. I_t then needs to simply select the highest of these values.

$$I_t = \underset{i \in \{1,\cdots,K\}}{\operatorname{argmax}} \left\{ \bar{X}_{i,T_i(t-1)} + c_{t-1,T_i(t-1)} \right\}$$

$$c_{t,s} = \sqrt{\frac{2 \ln t}{s}}$$

Here $\bar{X}_{i,T_i(t-1)}$ is the known average payout for slot machine i at time t and $c_{t,s}$ is the chosen bias sequence, or the tweaked variable that decays over time. In this case $c_{t,s}$ gives preference to unexplored machines, though it averages out as time t grows larger and more nodes are visited, which gives eventual preference to the machine with the highest payout.

This allows UCB much initial exploration while eventually converging towards the optimal choice as the number of attempts approaches infinity [4]. This characteristic of convergence is very important, as it allows searches that cannot be completed in real time (due to the size of the search area or the nature of the scoring values) to be stopped prematurely and yet still produce an answer within range of the optimal one.

1.3.3 Upper Confidence Tree (UCT)

UCT (Upper Confidence bounding applied to Trees) is an algorithm which applies the multi-armed bandit algorithm (UCB1) to trees; consider each node as a bandit and its child nodes as arms. It was developed by Levente Kocsis and Csaba Szepesvari in 2006 [4]. It balances the trade-off between the deep searches of high win-rate moves and the unexplored moves by applying UCB1. UCT is a simple but effective form of MCTS. However, instead of sampling the child nodes uniformly as the regular MCTS does, this algorithm tries to sample actions selectively to reduce the infeasible planning time for large branching factor trees [4]. It descends into the children nodes by applying UCB1 until it finally reaches a terminal, or a leaf node.

$$Q_{UCT}^{\oplus} = Q_{UCT}(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}}$$

$$\pi_{UCT}(s) = \underset{a}{\operatorname{argmax}}\, Q_{UCT}^{\oplus}(s,a)$$

After a playout, it updates the value of the nodes visited (actions played) iteratively from the leaf to the root.

$$n(s_t,a_t) \leftarrow n(s_t,a_t) + 1$$

$$Q_{UCT}(s_t,a_t) \leftarrow Q_{UCT}(s_t,a_t) + \frac{1}{n(s_t,a_t)}[R_t - Q_{UCT}(s_t,a_t)]$$

Here $Q_{UCT}(s, a)$ is the action value function for all (state, action) pairs; the initial value $Q_{UCT}(s, a) = R_t$, $n(s, a) = 1$; $n(s, a)$ counts the number that action a was selected from state s; $n(s) = \sum_a n(s, a)$.

The UCT algorithm is robust in three ways; it can stop at any time of the algorithm, it can smoothly handle uncertainty by computing the mean of the value of all the children weighted by the number of visits, and it builds a tree asymmetrically so that it explores more often in the moves that provide more rewarding outcomes [10]. UCT is guaranteed to converge to the optimal move if enough time is given. It was first used in MoGo [11] and significantly improved the playing strength of Go algorithms. Presently almost every top Go playing algorithm draws from the design of UCT in some way.

1.3.4 Rapid Action Value Estimation (RAVE)

The weakness of the UCT algorithm is that only the first move of a playout determines which node's values and counts are updated which results in slow learning. Rapid Action Value Estimation (RAVE) is a heuristic algorithm which updates the value of all episodes in which an action a is selected at any subsequent time [11, 12]. It is an extension of basic UCT but varies in that it updates the values across multiple states, rather than maintains the value on a per-action-state basis. AMAF (all moves as first) is a general name for this type of heuristic [13]. In RAVE, the action values are updated for every state and every subsequent action following that state (Fig 2).

$$m(s_{t1}, a_{t2}) \leftarrow m(s_{t1}, a_{t2}) + 1$$

$$Q_{RAVE}(s_{t1}, a_{t2}) \leftarrow Q_{RAVE}(s_{t1}, a_{t2}) + \frac{1}{m(s_{t1}, a_{t2})}[R_{t1} - Q_{RAVE}(s_{t1}, a_{t2})]$$

Here $Q_{RAVE}(s, a)$ is the rapid value estimate for action a in state s; $m(s, a)$ counts the number of times that action a has been selected at any time following state s.

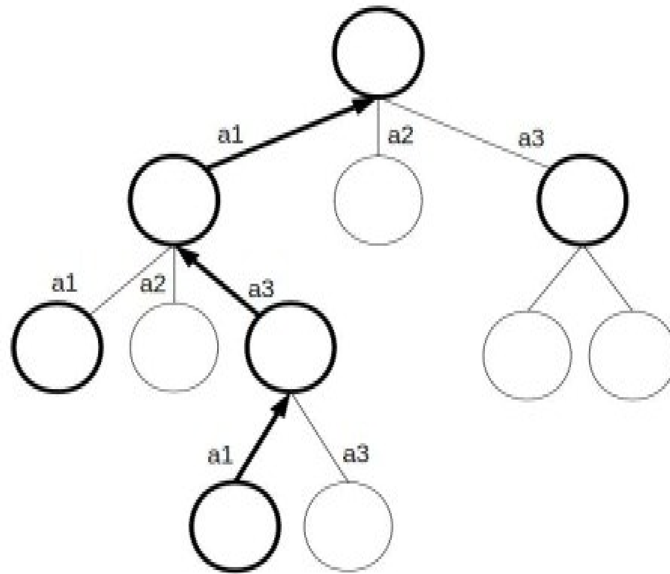


Figure 2 Nodes updated using UCT RAVE.

The value of the bolded nodes are updated along the path in UCT RAVE.

The UCT RAVE algorithm has proved to be extremely effective in Go with high learning speed, low variance at the beginning, and correct move convergence [11]. The success of this algorithm suggests that the value of moves could often be at least partially independent of the order in which they are played.

1.3.5 Genetic Programming (GP)

Genetic Programming, succinctly, is the automatic generation of programs for fulfilling a given task [7]. Like Genetic Algorithms, Genetic Programs undergo a process of generation, mutation, selection, and breeding to automatically improve them. However unlike Genetic Algorithms, which only alter simple variable parameters, Genetic Programs alter deeper patterns of behavior, and some variations of GPs can alter the behavior of itself. Similar to the evolution of a species, Genetic Programs are created with internal variations, or mutations, to their behavior. From these variating individuals, all are assessed by some predefined heuristic, and those that pass assessment are kept for further generations of assessment and mutation.

In the Go codebase MoGo, Hoock and Teytaud have developed Bandit-Based Genetic Programming (BGP), which rather than starting with a genetic program completely from scratch, starts with the MoGo codebase and introduces additional patterns to evaluate [7]. Since the performance of algorithms is difficult to prove outside of testing their culling heuristic was based on the statistics of simulated games.

While the use of genetic programming holds promise in Go, the amount of time required to simulate and evaluate the generated programs to any notable degree easily surpassed our project deadline given our resources. Furthermore, while it would have been intriguing to implement a technique that differed so completely in behavior from our other algorithms, the dynamic nature of Genetic Programs did not coincide with the structure of our existing codebases, which are described in Section 2.

1.3.6 Neural Networks (NN)

The use of Neural Networks, like Genetic Programming, is another biologically inspired approach to algorithms [14]. The advantage of Neural Networks is the ability to function in the same observable way as a biological brain, finding hidden patterns in an environment and concluding on actions on potentially the same level as a human. An NN is a collection of artificial neurons, usually nodes ("neurodes"), connected together in a way to allow the learning of a specific function or task [14]. Neurodes themselves can be very simple, as one neurode needs only to know how to behave towards the neurodes that it is immediately connected to. While this sounds almost too simple to work, the performance of an NN is a result of the emergent behavior caused by the interaction between its neurodes. With certain neurodes receiving input from an environment and sending signals to other neurodes based upon that input, collectively these neurodes produce sophisticated actions as a result of the state of all neurodes, even from very noisy data. With enough processing power a NN is currently the best AI solution to finding and evaluating patterns in an environment [14].

Implementations of Neural Networks in the context of Go, such as NeuroGo [6], have been made with nominal success. Since Go positions are so difficult to evaluate due to the sheer number of outcomes, it would seem obvious that NNs could outperform other algorithms in that task by finding patterns in the stones that other algorithms couldn't see. Previously processed games were fed to NeuroGo in order to teach it game behavior more quickly, as is customary for most Neural Networks, vastly improving the NN's performance compared to untaught NNs. After such preparation NeuroGo was compared against traditional Go algorithms.

While there has been research in Neural Networks in the context of Go, their usefulness in Go and other large search trees are quite limited. The advantage of Neural Networks as assessment algorithms lies primarily in their ability to recognize patterns. Their disadvantage is the amount of time and resources needed to asses and find these patterns, which is magnified under the sheer amount of board assessments required in large Go trees. Added with time limits to searches within the tree, even parallelized, neural networks provide almost no advantage over most algorithms in their current structure [6]. From these disadvantages it was concluded that Neural Networks were not pursuable for this project.

2 Existing Codebases

2.1 Gomba

2.1.1 Artificial Game Trees

In a real Go game, the optimality of a move cannot be calculated in advance; determining when a game has terminated is slow; heuristic evaluations of non-terminal states are both slow and inaccurate [1]. A Computer Go program generally takes an hour to finish a game even on a very powerful hardware. For researchers who want to test new algorithms and conduct statistical analysis with sufficient size, it is infeasible to use real game trees. Also, a result from a real game tree is not how good an algorithm is. It is actually a relative winning rate compared to its opponent algorithm. Therefore, the information gathered from a real game tree is not accurate and will take an extremely long time.

Artificial game trees attempted to give solutions the problems described above. They are used to test new search algorithms before applying them to a real Go game [4], with faster speed and better heuristics. The parameters (branching factor, depth, etc.) of an artificial game tree can be easily modified according to testing needs and the testing results of an algorithm do not depend on any opponent algorithms.

In this report, we used Gomba [1], an Artificial Game Tree testing framework developed by WPI students Daniel Bjorge and John Schaeffer in 2010, and enhanced

more features such that it could be a better and more accurate testing tool for other researchers.

2.1.2 Features of Gomba

Gomba was developed by Daniel Bjorge and John Schaeffer from WPI in 2010 [1]. It is an artificial game tree framework for testing the performance of different Go algorithms. The tree generation algorithm was able to determine minimax-equivalent search entirely, significantly increased the searching speed and provided feasible solutions to test algorithms against trees that were previously too large to consider at all.

Some features of Gomba include:

- Lazy State Expansion
- Deterministic State Expansion
- Pseudorandom State Expansion
- Predetermined State Optimality
- Fast Action Simulation
- Fast Termination Evaluation
- Fast Heuristic Evaluation
- Go-Like Action-Reward Distribution

2.1.3 Weakness of Gomba

In the previous version of Gomba the choice of actions that led to good and bad outcomes was purely random. This violated an important assumption behind how UCT RAVE worked. The reason that UCT-RAVE outperformed regular UCT was that it maintained a global knowledge of the moves and updated the statistics of all the moves along the path selected, as explained in section 1.3.4. However, the Gomba game tree did not have such global knowledge (transposition table [15]) so that the testing result of UCT RAVE vs UCT was not precise. In the previous Gomba framework however, the UCT RAVE actually performed worse than regular UCT algorithm because UCT RAVE added a lot of noise at the beginning of the search.

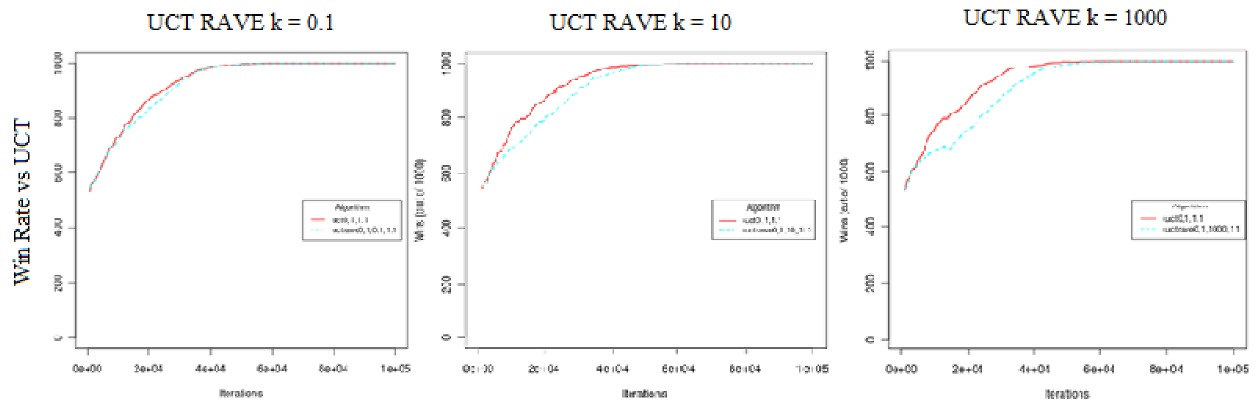


Figure 3 UCT RAVE win rate in the original Gomba framework

As Figure 3 shows above, when we increased the equivalence parameter *k*, the performance (win rate) of UCT RAVE became worse. This was because when evaluating a Game State node, the UCT RAVE value was decided by a linear combination of both regular UCT and RAVE values.

$$Q_{UR}^{\oplus}(s,a) = \beta(s,a)Q_{RAVE}^{\oplus}(s,a) + (1-\beta(s,a))Q_{UCT}^{\oplus}(s,a)$$

The smaller the equivalence parameter k was, the more similarly UCT RAVE behaved to regular UCT. Because the actions were generated randomly and lacked consistency (e.g. action 1 in depth 3 was a good move, but in depth 5 it suddenly turned into a bad move); this violated the way that UCT RAVE worked as described in section 1.3.4.

In this project, we modified the Gomba testing framework such that it maintained a global knowledge of the moves so that it could better simulate a real Go game. This was a continuation from last year's MQP.

2.2 Fuego

The Fuego Codebase is an open-source collection of C++ libraries of existing Go algorithms widely used in the Go Artificial Intelligence community. It was originally developed in 2009 by Markus Enzenberger and Martin Mueller and the most recent version – which we used – is version 1.1 and was released in 2011. Widely regarded as one of the top Go Codebases, Fuego is most notably known for being the first artificial Go system to defeat a 9-Dan professional Go player on a 9x9 size Go board, which occurred in August 2009.

2.2.1 Architecture of Fuego

Fuego contains several substructures of varying abstraction, allowing the implementation of algorithms to be straightforward and generic without being lost in the semantics of Go specifically. It has only one external library, Boost, which it uses for the significant amount of random number generation required in most UCT algorithms as well as unit tests.

Fuego is automatically documented online using a Doxygen-type formatting originally intended for Javadocs. The generated result is acknowledged to be unintuitive [16] and entire projects in the past have been undergone to simply describe the process that Fuego uses to build and implement its searches [16]. Fuego is also built to handle multithreading, which significantly improves performance although it makes approaching the framework – as well as implementation – all the more difficult.

Fuego uses the Go Text Protocol (GTP) [15] as the accepted base for communicating between processes, allowing it to play against algorithms implemented in other frameworks and even other programming languages, such as GnuGo[16] or MoGo[17]. This communication is usually handled through a third process that arranges games, which in our case was GoGui-TwoGtp, one of the Java executables of the GoGui standalone [18].

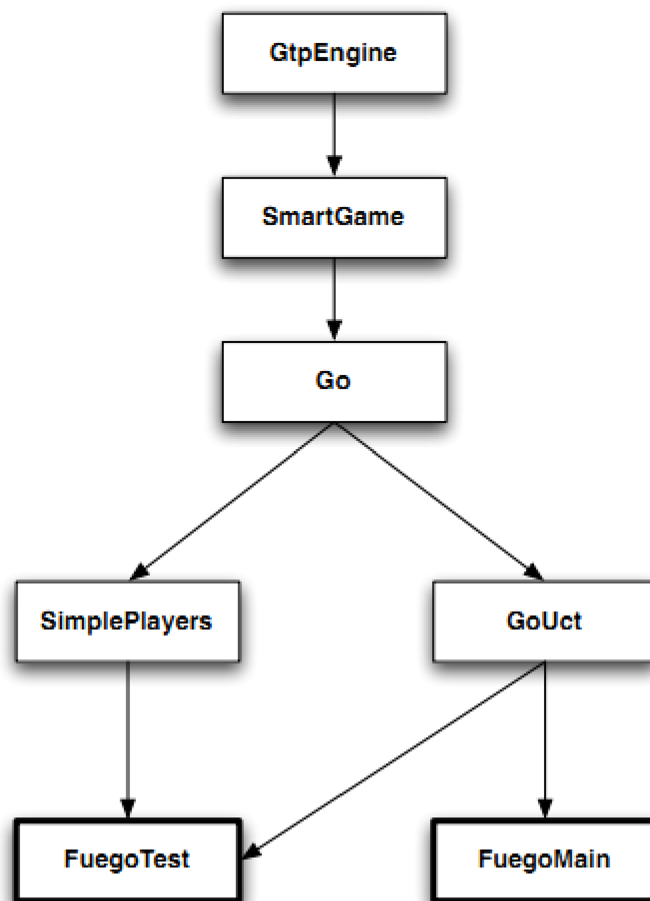


Figure 4: The Fuego Dependency Tree

The GTP protocol, as well as other unique Fuego commands, is at the lowest level library (GtpEngine) and not dependent on any other library. Above that is the SmartGame

library, containing the utility classes for multiple games and where most of our implementation occurred. Above SmartGame are the Go-specific classes in the Go library that handles the basic Go-related rules, and below that is the GoUct library, which handles the behavior for UCT through the GoUctPlayer class, where the remainder of our implementation happened. The main application for Fuego, FuegoMain, allows for a GTP protocol from GoUct to other processes or a human player.

To give a brief example; let us say we want to give the command to generate a move for the black player (or “genmove b” in GTP). The command would first be parsed in the GtpEngine library where most commands (including ours for generating a move) are registered. GtpEngine would then pass the command to the GoUctPlayer class in the GoUct library, where the type of search (which we predefined at the beginning of runtime) would be determined and begun. GoUct would call the Search function in the SgUctSearch class in the SmartGame library, which would in turn start the thread and game initialization. It is within SgUctSearch where the game tree is expanded, assessed, and eventually pruned. It was also within the SmartGame library where we carried out the majority of our implementations, though most of the lower level thread handling was not touched.

There are of course exceptions to this traversal, as Fuego allows for several implementations of UCT searches that implement node and move assessment in many different ways. Most, however, follow this model.

3 Gomba

3.1 Additions to Gomba

3.1.1 Modification of tree generation algorithm

As stated in Section 2.1.3, the weakness of the existing Gomba framework was that it lacked the property of consistency between the actions (moves) along the Game Tree. To better simulate a Go game tree, modifications were made which are discussed in the next three sections.

3.1.2 Addition to searchData field

Two fields which recorded the estimated value and number of visits were added. These two fields were maintained in the searchData data structure in a Game State, which specifically remembered the statistics of UCT RAVE.

When evaluating a Game State, we calculated the upper confidence bounds by mixing the regular UCT value and RAVE value by a linear combination of the two values.

$$Q^{\oplus}_{RAVE} = Q_{RAVE}(s,a) + c\sqrt{\frac{\log m(s)}{m(s,a)}}$$

$$\beta(s,a) = \sqrt{\frac{k}{3n(s)+k}}$$

$$Q^{\oplus}_{UR}(s,a) = \beta(s,a)Q^{\oplus}_{RAVE}(s,a) + (1-\beta(s,a))Q^{\oplus}_{UCT}(s,a)$$

$$\pi_{UR}(s) = \operatorname*{argmax}_{a} Q^{\oplus}_{UR}(s,a)$$

Here parameter c is the bias. The equivalence parameter k controlled the number of episodes of experience when both estimates were given equal weight.

As the above formula suggested, we needed the Gomba testing framework to remember statistics from both UCT and RAVE algorithms and mix them together as the final evaluation value. Therefore, we decided to add two new fields under searchData data structure in a Game State to record the RAVE statistics, separate from the regular UCT statistics.

3.1.3 Correlation and consistency among actions

As stated in Sec 2.1.3, the moves' outcomes were purely random in the existing Gomba framework such that no correlation existed among the moves. This lack of consistency affected the accuracy of the testing results for UCT RAVE and other variations of AMSF heuristics. The success of the UCT RAVE algorithm suggested that the estimate of a move was partially dependent on the order of the moves. This meant that for the same moves in different depths of a game tree, they should be related to each other, rather than purely random. The original Gomba framework violated this assumption.

We introduced a new tree generation algorithm to include consistency in Gomba framework. This algorithm used the nodes one level up as standards, rearranged the statistics for the child nodes so that the children followed the distribution of their parents.

The algorithm worked as follows:

Algorithm 1 Define the correlation when generating a new Child Node

```
1   getChild(action):
2       if  state.children[action] is not defined:
3           state.children[action] := generateChild(action)
4       return state.children[action]
5
6   generateChild(action):
7       childState.depth := state.depth + 1
8       childState.player := OtherPlayer(state.player)
9       childState.prng.seed := GetNthRandom(state.childSeed, action)
10      childState.childSeed := childState.prng.nextSeeed()
11      childState.difficulty := prn.varyDifficulty(state.difficulty)
12
13      childState.winner := state.winner
14
15      if((currentDepth == 0)||(currentDepth == 1) || (action == 0))
16          return childState
17      else
18          NodeAsStd1 := getParent.getChild(action - 1)
19          NodeAsStd2 := getParent.getChild(action)
20          NodeToCompare := getChild(action - 1)
21
22          if((((NodeAsStd1.difficulty < NodeAsStd2.difficulty) &&
23             (NodeToCompare.difficulty < childState.difficulty))||
24
25             ((NodeAsStd1.difficulty > NodeAsStd2.difficulty) &&
26             (NodeToCompare.difficulty > childState.difficulty)))&&
27
28              prng < givenProbability)
29
30              Swap the Seed
31              Swap the RNG
32              Swap the difficulty
33              Swap the winner
34              Swap the childSeed
35              Update the ForcedChild in the ParentNode
36              Swap the ForcedWinner
37
38          return childState
```

Line 7 to Line 11 was the original code when generating a new Child Node. We eliminated some code for deciding the winner in Line 13. The major bulk of modification was from Line 15 and later.

The algorithm took four different nodes:

- A newly generated node, N, with index (action);

- A sibling node of N, NodeToCompare, with index (action – 1)
- Two nodes from one level up of the game tree, with corresponding indices. NodeAsStd1with index (action – 1) and NodeAsStd2 with index (action). These two nodes were set as standard.

Every time we generated a new Node N, the algorithm grasped the relevant information (difficulty of the Game State) from the above four nodes, compared them, and then decided whether to swap the information between N and NodeToCompare by the given probability.

First we checked if the four nodes met the swap criteria. That is, we looked up and compared the difficulty of the four nodes. Because the game tree is a mini-max game tree, your adversary always wants to minimize your gain. So if we want a minimized value in depth d, then in depth (d-1) we want the value to be maximized. As illustrated in figure 4, the difficulty of Std1 is less than the difficulty of Std2 in depth (d-1). In depth d, the difficulty values were the reversed value because the two nodes were minimizing nodes. Even the shown value was 0.4 and 0.6, they actually meant -0.4 and -0.6. Therefore we wanted to swap the value of the two nodes.

Now the swap condition was met, then we decided whether to swap the values of node N and nodeToCompare based on a probability. If the given probability was 1, then we swap every time, this would result the actions are 100% correlated; If the given probability was 0.5, we swap the values of the nodes by 50% chance, the actions were 50% correlated; If the given probability was 0, then we did not swap the values at all – then the game tree behaves exactly as the original game tree, the actions are purely random

(Line 28). By adding this "givenProbability" parameter, we could control the correlation level between moves.

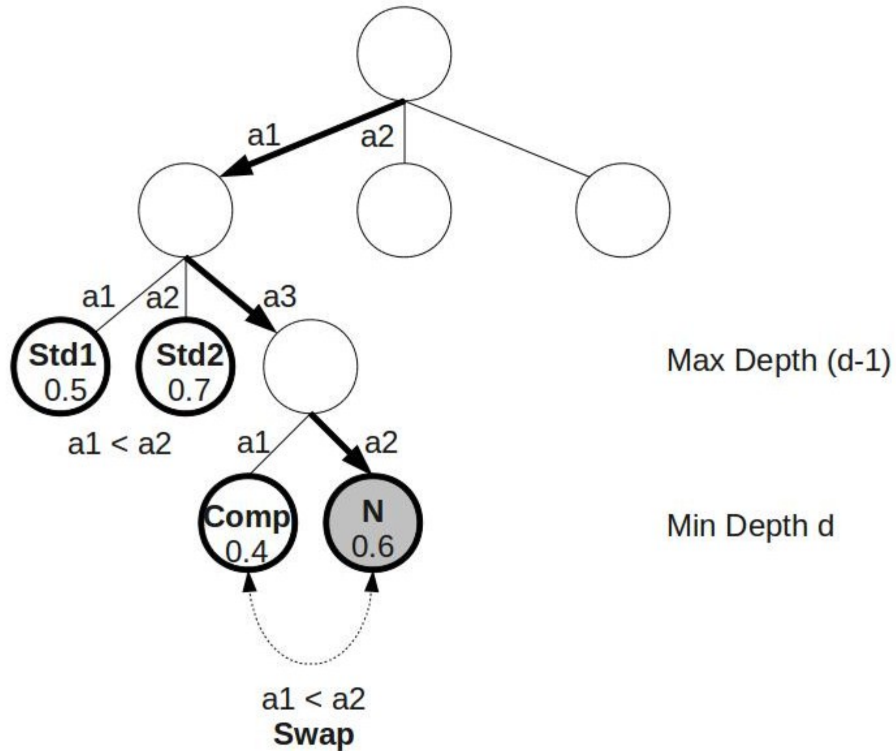


Figure 5 Algorithm 1 illustration

We were interested in seven statistics in a Game Tree node. These values were listed from Line 30 to Line 36.

The first value we swapped was the difficulty of the two nodes. The difficulty measured how the difficult was this node for each player to win. The closer to 0, the easier it would be for player 0 to win, and vice versa. We used the difficulty as the main factor of the correlation among the moves. That is, if move 1 was an easier win at depth 4,

then in depth 10, it would also be relatively easier to win. Therefore, when we decided to swap the nodes, the first value to swap was the difficulty.

The values Seed, RNG, and ChildSeed were basically some random numbers used for generating the child nodes. Because we wanted the moves in the artificial game tree to be correlated with each other, we also wanted this property to be persistent among their child nodes. Therefore, when we decided to swap two nodes, we swapped the Seed and all related random number generators as well.

Winner was the predetermined minimax winner from this Game State node, where both players were to play out the rest of the game tree optimally. If this Game State node required all children to be a particular Winner value, it would be this value. If the value was NEITHER, the choice would not be forced. The values of Winner and ForcedWinner thus also need to be swapped because they were related to a win state of a given node.

The value of ForcedChild was tricky. If this node will requires at least one child to be of a particular Winner value for the sake of minimax tree construction, this value is the action of the random child that would be forced to that value. Now the children of a node were not random anymore because we swapped them upon generation based on the difficulty. Therefore, the ForcedChild would also change. But the change happened in the parent, not in the child node itself. So we could not simply swap this value as the other six values described above. We need to go to the parent and update the value in the parent node.

After all the seven values were swapped and updated, we return the child node requested. This "polished" child node was not purely random any more, and correlations were introduced between it and all its sibling nodes.

3.1.4 Lazy State Expansion

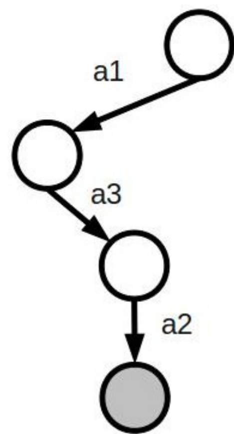
The existing Gomba artificial game tree expanded and generated only one child node (Game State) at a time when needed. In the new version of Gomba, we wanted the same action to be consistent in the game tree. To guarantee this property, we used the algorithm proposed in the previous section. However, with the modification above, the searcher (search algorithm) might look into the node (Game State) statistics before the difficulty and win-rate values were updated (swapped). When a searcher saw the information of a Game State, it would actually be looking at the old information before it was swapped. The testing result would be wrong.

To prevent such a situation, we modified the node generation algorithm such that when the tree decided to descend to a new child node, it expanded all the siblings of that node as well. Upon generation, the tree compared and swapped the difficulty and win probability values when necessary.

Algorithm 2 Modified Lazy State Expansion

```
1  getChild(action):
2      if  state.children[action] is not defined:
3          for (i=0; i<NumChildren; i++)
4              if state.children[i] does not exist
5                  state.children[i] := generateChild(i)
6
7      return state.children[action]
```

This new tree generation algorithm also had weaknesses. It required more memory and took more time than the original design because it expanded all children of a node, especially when the branching factor was very large. Though this new method was less "lazy" than the original generation algorithm, it was still "lazier" than expanding the whole game tree. In order to maintain the global knowledge of the actions, we had to compensate some part of the memory management.



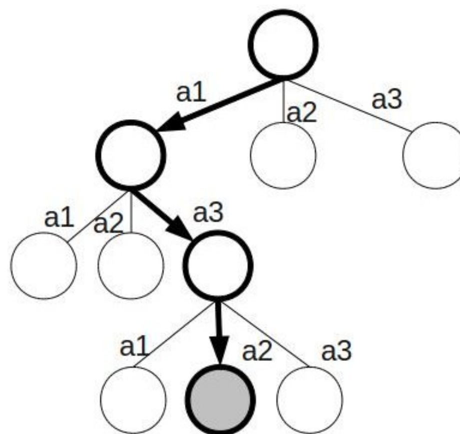


Figure 6 Comparison of lazy state expansions

3.2 Experiments in Gomba

Our experiments were compared for the maximization of two metrics; win-rate and average difficulty. The first is the algorithms' performance in maximizing optimal win rate, which measures how quickly an algorithm can consistently choose moves which are minimax-optimal. The second metric is the average difficulty of the moves that the algorithm chooses.

This value of average difficulty corresponds with the difficulty of a tree node in the artificial game tree. It measures how likely each player is to win when a path is chosen starting from its parent node. A value close to zero means that it is easier for the current player to win in this game state, satisfying our goal that the difficulty value be as low as possible. This is not the same as optimal win rate, and in fact minimizing the difficulty level can sometimes even be at the cost of a worse optimal win rate. This can often occur in adversarial search, as it is often the case that making it harder for your opponent to find good moves is as valuable as finding good moves yourself.

3.2.1 Comparison of Old and New Gomba Game Tree

The old Gomba testing framework only updated the statistics of a node locally. For example, if we selected action 1 at depth 4, only one corresponding node's value would be updated. The old Gomba testing framework ignored a global view of the action in all depths. It was sufficient to provide a decent testing result for most roll-out Monte-Carlo

based algorithms, such as the UCT algorithm. However, for AMAF algorithms which relied on a global view of all the moves, the old testing framework seemed to be deficient.

We modified the Gomba testing framework using the two tree generating algorithms proposed in Section 3 while the basic UCT algorithm was used as a control group. We plotted and compared the performance of the UCT RAVE algorithm in both the old and the new Gomba testing framework to see if the new testing framework with global knowledge would improve the performance of the UCT RAVE algorithm.

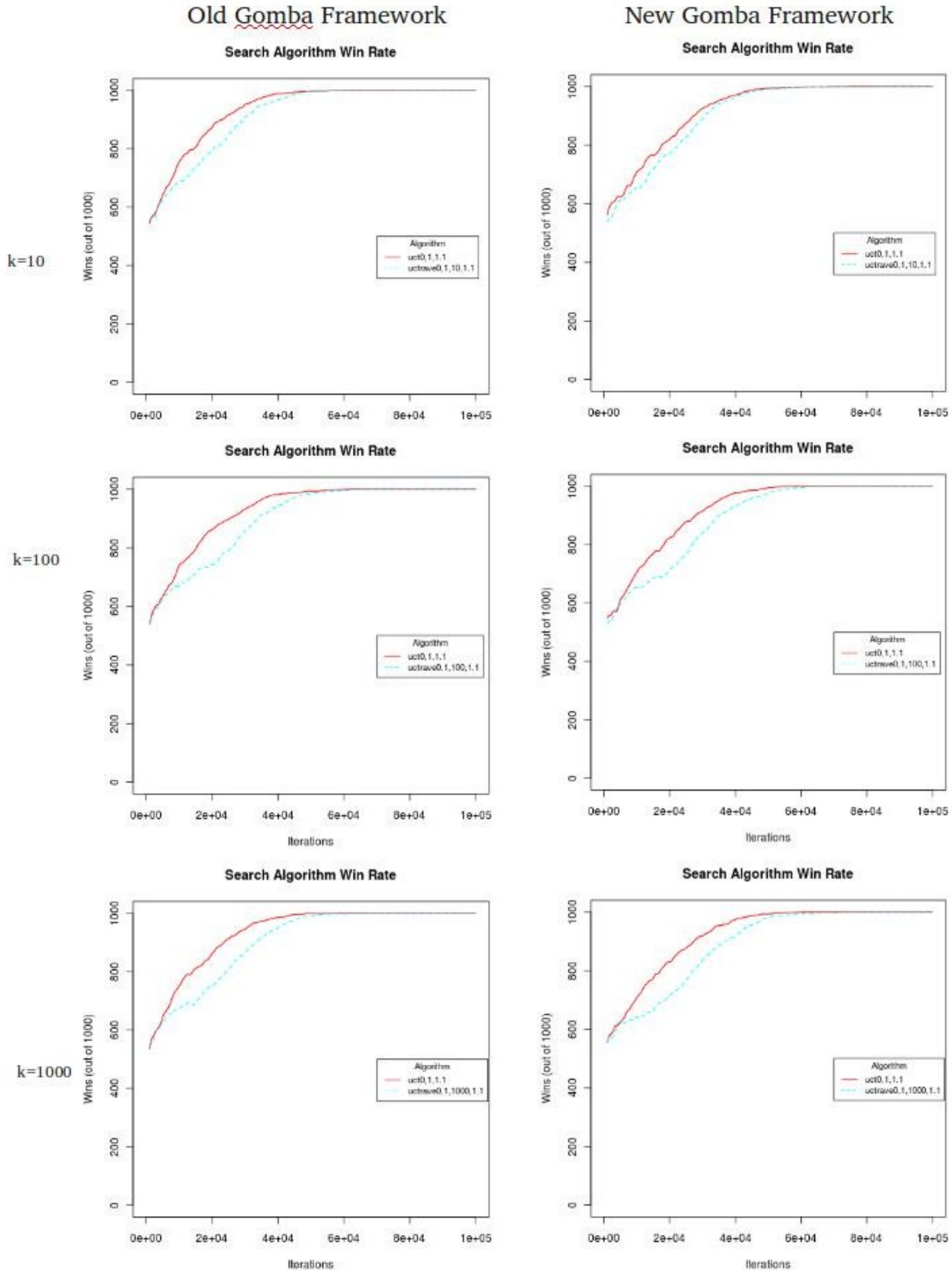


Figure 7 Win rate for old and new Gomba framework

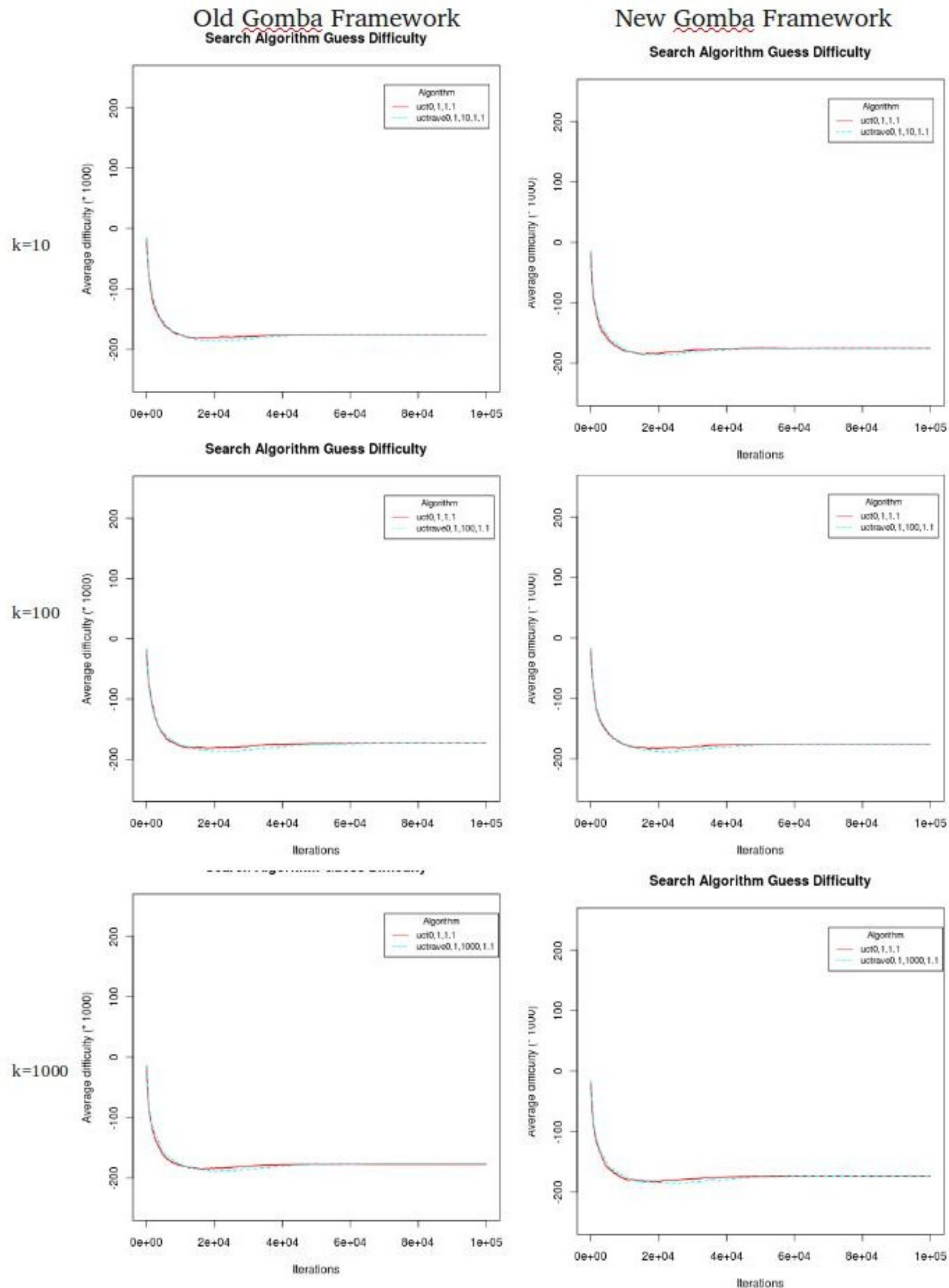


Figure 8 Average difficulty for old and new Gomba framework

The above figures plotted and compared the performance (average difficulty and win rate probability) using the previous version of Gomba framework (left) and the modified version (right). UCT RAVE is represented by blue dotted lines while regular UCT is represented by red solid lines. The modified version of Gomba used the two algorithms proposed in this paper (Section 3.1.3 and section 3.1.4), which added correlations between the moves. We expected this new property of the game tree would improve the performance of UCTRAVE.

In Figure 7 there was strong evidence that the win rate of UCT RAVE was improved in the new Gomba framework over the old version, though in Figure 8 there was not a very significant difference between the average difficulty between UCT RAVE and regular UCT. However, we could still see that the average difficulty for UCT RAVE is lower than the value of regular UCT as we expected. While our measure of the performance of UCT RAVE still could not exceed the regular UCT this is different from what Gelly *et al.* observed in a real game tree [11]. Even though the performance of UCT RAVE was not as good as the performance of the regular UCT as we expected, there was still a certain level of improvement in the new Gomba framework.

3.2.2 Gomba tree with different equivalence parameters

When evaluating a node, the upper confidence bound is calculated using a linear combination of the regular UCT value and the RAVE value. Section 3.1.2 discusses the RAVE value in further detail.

$$Q_{UR}^{\oplus}(s,a) = \beta(s,a)Q_{RAVE}^{\oplus}(s,a) + (1-\beta(s,a))Q_{UCT}^{\oplus}(s,a)$$

. How the two values are mixed is based on equivalence parameter k. In this section, we tested how the equivalence parameter affected the performance of the UCT RAVE algorithm.

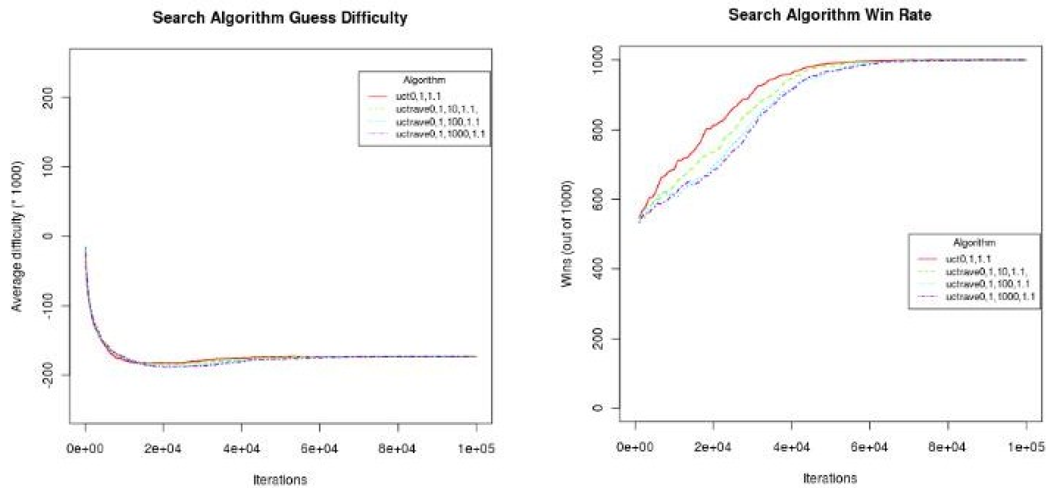


Figure 9 Different Equivalence Parameter Settings

As the above figure presented, the equivalence parameter k did impact how the UCT RAVE algorithm performed, but it was similar to using the old version of Gomba framework – the smaller the k is, the closer the UCT RAVE is to regular UCT. Even though our Gomba testing framework is improved, there are still some hidden factors which have not yet been discovered that seem to affect the correlations, making the UCT

RAVE value still act as “noise” to the regular UCT. No other direct conclusion of the equivalence parameter *k* could be drawn from the result.

3.2.3 Gomba Tree with Different Correlation Settings

As stated in section 3.1.3, we wanted to have control of the level of correlation within the artificial game tree. We tested the performance of UCT-RAVE in different levels of correlations (0%, 50%, and 100% respectively) and we could see a small trend of change from the 0% correlation to the 100% correlation. If we only look at the first and third graph in Figure 10, the change is obvious.

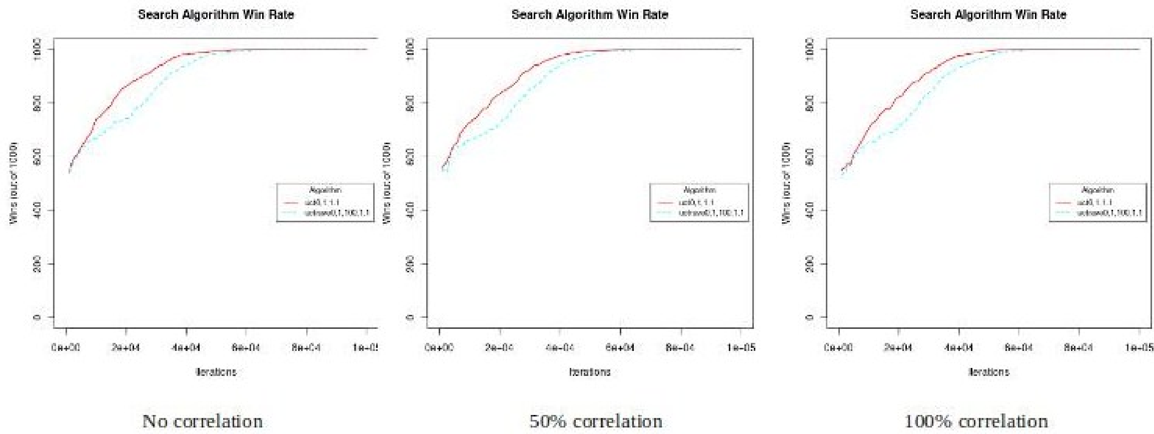


Figure 10 UCT RAVE performance using different correlations

3.2.4 Summary

In this project, we tested the how well the modified version of the Gomba testing framework supported the algorithms relying on move transpositions, namely the UCT-

RAVE algorithm. As shown by our results we improved the accuracy of the testing framework from the previous version of Gomba. The level of correlation in an artificial game tree also affected the performance of the AMAF algorithms. The Gomba testing framework is still under development and more areas still need improvement in order to better support AMAF algorithms and meet the expected result.

4 Fuego

4.1 Additions to Fuego

4.1.1 PoolRAVE

PoolRAVE (or RAVE(Pool)) is a UCT-based search that functions almost identically to basic UCT-RAVE. Developed by Teyatud et. al. in 2010 [5]. The algorithm's main difference is that it bypasses Monte Carlo Search entirely. As the game plays, the unused previous moves with the highest calculated RAVE values are stored in a "pool" of a determined size. Before a Monte Carlo search is run, within a certain probability a move is instead chosen at random from the pool of the recently visited node. The attraction of this algorithm was mainly its ability to recycle old moves and to bypass the expensive Monte Carlo search, significantly reducing the amount of time to reach a conclusion.

A visualization of the hypothetical performances of basic UCT-RAVE and RAVE(Pool) is shown in Figure 11. In it you can see that under the correct circumstances RAVE(Pool) can arrive at the same conclusion as basic RAVE with much less time. While basic UCT-RAVE always requires a time of amount T to perform Monte-Carlo search, RAVE(Pool) can perform the task with similar results at a time of amount t, which is equivalent to the time required to accessing a single random member of a list (which is comparatively instantaneous).

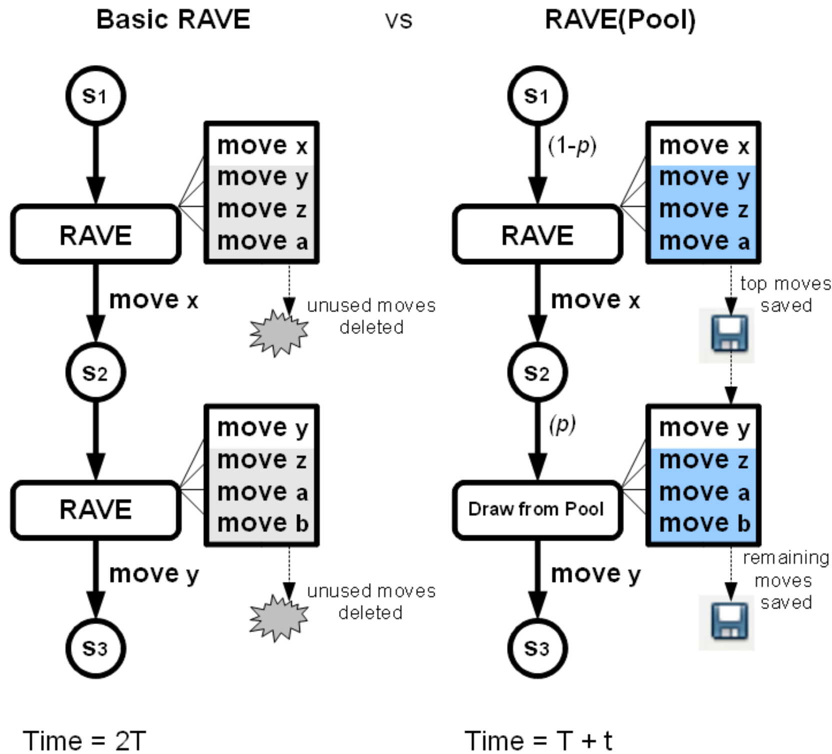


Figure 11 : Performance Difference Between RAVE(Pool) and Basic RAVE

Implementation of PoolRAVE required a simple bypass of the regular functionality of Fuego, in addition to the inclusion of a pool for the stored move values which could be of varying predetermined size. The Boost library was used to handle the randomized selection of the action, which needed to be normally distributed according to Teytaud [5].

While PoolRAVE provides a significant improvement over basic RAVE in terms of time it is not without its caveats, as the pool may stagnate given a long enough time between when the pool is filled and the pool is drawn from. In other words; a move taken from the pool might have been used already between the time when it was allocated into the pool and when it was picked. A used move in Go, in many cases, makes it illegal to play again in the immediately following moves due to an illegal Ko or the simple fact that

the space on the board is already occupied. The likelihood for a certain pool to stagnate relies on a number of factors, though primarily it is determined by the probability p that the pool will be drawn from next and the move score concurrency of the game being played. Move score concurrency being the propinquity of the scores between the top moves that are shared by the two opponents.

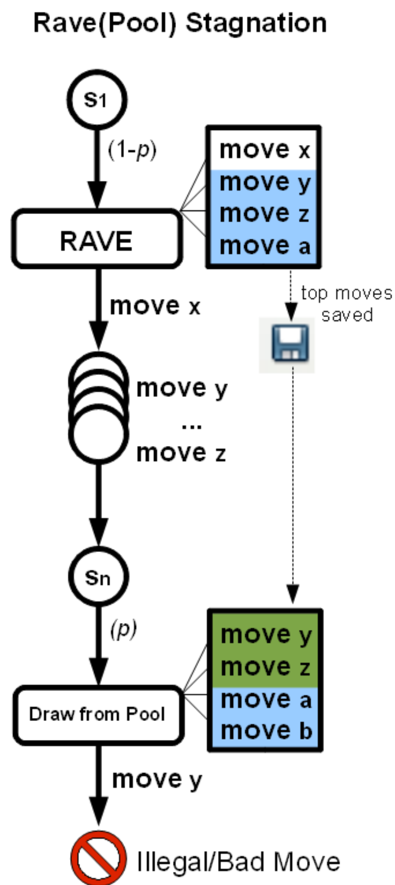


Figure 12 : An Example of a Stagnant Pool

As the behavior of Go makes most good moves that you find also good moves for your opponent if your opponent made them, and the pool is to be always filled with moves with the highest RAVE value, the error of illegal moves being in the pool occurred with enough frequency for it to significantly affect the algorithm's performance. Resultantly, the specific solution we would choose for encountering a stagnant pool would also greatly affect the algorithm.

Teytaud's description of PoolRAVE is not specific on how to select from a stagnant pool, nor was he specific on what his personal solution was [5]. As a result we independently developed and implemented several variations of PoolRAVE with different behaviors based around the occurrence of an illegal move being picked from the pool.

4.1.2 PoolRAVE(Pass)

The easiest solution to encountering an illegal move selected by the pool would be simply for the algorithm to pass its turn. This provided the promptest behavior, and was similar to Fuego's basic approach towards error checking after running plain RAVE. While promptness is not a specific enough criterion to quantify or measure, it is nevertheless important in many real world and testing contexts that involve time-critical decisions. This promptness did mean, however, that the solution would give our opponent several more free moves and advantages early-to-midgame. The advantage to the opponent would eventually minimize later within each simulated game, however, and would also give the opponent the lowest amount of time to ponder. It was decided that testing was required to conclude PoolRAVE(Pass)'s performance.

4.1.3 PoolRAVE(PersistPass)

The next solution after simply passing on an illegal move would be to check the entire pool for a legal move and return the first one. This solution eliminates Teytaud's requirement of normal distribution for selecting the move randomly in the pool, though the manner in which the pool is generated and maintained (inserting and replacing the top moves as they are encountered) provides significant randomness. This approach reduces the number of passes given to the opponent, as the likelihood of every move in the pool being illegal is exceedingly rare under normal circumstances. However, given the proper heuristic values; for example a very low probability of selecting from the pool or a very small pool size, the encounter could easily become very likely. PoolRAVE(PersistPass) gives us similar advantages to PoolRAVE(Pass) during late-game, and tries to remove its

early-and-midgame disadvantages by being more thorough with move selection from the pool.

4.1.4 PoolRAVE(SmartPersist)

PoolRAVE(SmartPersist) follows the behavior of PoolRAVE(PersistPass), though instead of passing after all moves in the pool are found illegal, simply runs through the basic RAVE behavior to find a move. This approach completely eliminates the pass disadvantage of PoolRAVE(Pass) and PoolRAVE(PersistPass) at the cost of additional processing time for potentially checking the entire pool for legal moves. Furthermore, late-game behavior becomes weakened when PoolRAVE(SmartPersist) is compared to PoolRAVE(Pass) and PoolRAVE(PersistPass). During portions of the later part of the game where the best move is to pass, PoolRAVE(Pass) and PoolRAVE(PersistPass) will quickly (though naively) arrive at this conclusion, where PoolRAVE(SmartPersist) must perform a full search of the tree before concluding the same thing. The approach of simply performing a full search after the initial move in the pool was found to be illegal, or PoolRAVE(Smart), was considered, though the additional processing time required for a full search made any advantage of skipping the remaining moves in the pool negligible.

4.2 Experiments in Fuego

In order to properly compare the effectiveness of the implemented algorithms a common comparison was needed. GnuGo 3.8 at difficulty level 6 was used for its compatibility with the Go Text Protocol (GTP), which allows the two programs to

exchange moves easily and efficiently. It was also picked due to its use in the works of Schaeffer and Bjorge of last year, so our results may be compared easily with theirs.

In order to properly compare the effectiveness of the implemented algorithms a common comparison was needed. GnuGo 3.8 was used at difficulty levels 6 and 7 for its compatibility with the Go Text Protocol (GTP), which allows the two programs to exchange moves easily and efficiently. It was also picked due to its use in the works of Schaeffer and Bjorge of last year, so our results may be compared easily with theirs.

A slight modification was made to Fuego during the testing phase in addition to our implementations of the algorithms. We found during initial test runs against GnuGo that Fuego was prematurely resigning with a very high frequency and not giving useful results. While Fuego's method of determining resigns was provably optimal under normal circumstances, we felt that our algorithms may be causing improper conclusions. The default resign threshold was therefore increased appropriately, allowing our algorithms a more thorough playthrough and more conclusive results. The tradeoff to this, however, was that it took much longer for simulated games to be played and resulted in less experiments being possible in a fixed time.

4.2.1 Basic Fuego vs GnuGo

Basic Fuego was compared to GnuGo 3.8 at Difficulty level 6 for 1000 simulated games at default UCT-RAVE values in order to compare to the results of the previous year. Due to improvements in Fuego since the results of Schaeffer and Bjorge, basic Fuego was also compared to GnuGo level 7 for 1000 games to coincide with our own results.

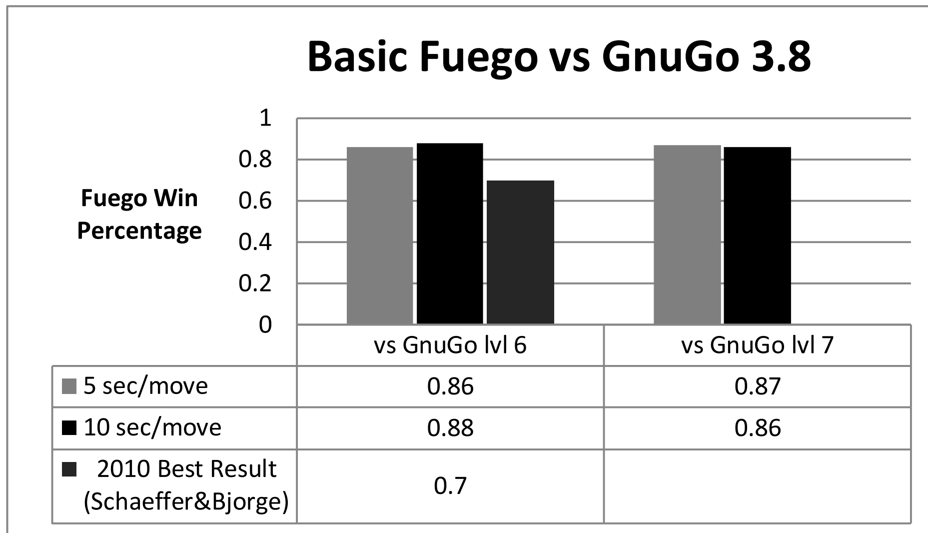


Figure 13: Current Fuego performance against varying levels of GnuGo

Already we have a significant improvement over last year's best results, which were with basic UCT at 5 seconds per move [1]. It is important to note that having a longer time to calculate moves did not produce a significant improvement in Fuego's already impressive win percentage, and as the GnuGo level increased more time actually resulted in slightly decreased performance. This counter-intuitive result can be justified by the behavior of different difficulty levels of GnuGo as well as server-side interference.

While GnuGo performs the same task as Fuego, it operates much differently. Rather than using a specific Monte-Carlo search, GnuGo generates many moves from several different move generators at runtime. Multiple generators allow GnuGo to create moves quickly and evaluate them based on the given situation [1]. Opening moves, for example, are difficult to create initially as you have the entire tree to traverse. Using pre-stored responses to the first few opening moves allows for much faster performance. The set levels of GnuGo affect the generation and assessment of these moves, making it react

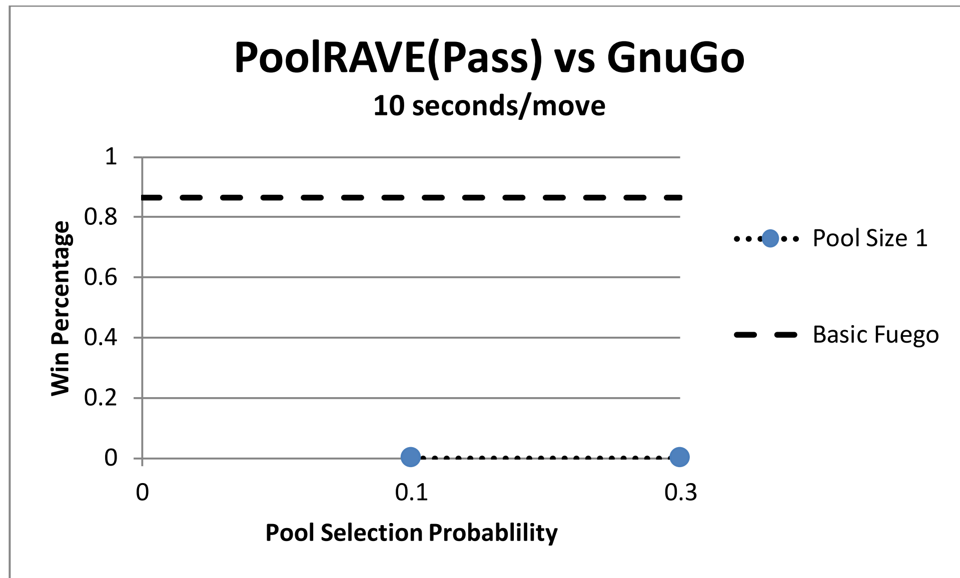
to specific situations less or more appropriately. Thusly, GnuGo set at a lower level could perform more poorly on average against every type of game-playing AI but could have factors that make it better against Fuego specifically in that circumstance.

In order to describe server-side interference a brief description of our server cluster is required. SZTAKI allows its researchers the use of large clusters of machines for faster results in their testing. Tests that are run on this cluster, each called a "job," are submitted and monitored by a separate process called a submission system which notifies you when a job has finished and what machines on the cluster are available to you. It was learned during testing that our submission system, Condor, was not the only submission system on the SZTAKI cluster, which made the resources that Condor stated as available different from those which were actually available. Depending on additional activity on the cluster at the time of simulation the processes run by Condor could potentially be much more inefficient and inaccurate. Many times the processes would simply grind to a halt and Condor would cancel them when only partially finished. This was a phenomenon which Condor could not manage or compensate for, and is regarded as unavoidable noise in the results. Regardless of such noise or other interference, it is nonetheless communicated here that that basic Fuego performs slightly better against higher levels of GnuGo when given shorter time to think.

4.2.2 PoolRAVE(Pass)

PoolRAVE (Pass) was understandably the weakest of the tested algorithms. Any Go algorithm that has a larger likelihood to pass on a move will undoubtedly perform weaker than one that finds a move with any semblance of assessment. Even with very low

selection probability and a modified resign threshold the win percentage was zero against level 7 GnuGo.



	PoolRAVE (Pass) Pool Size 1	Basic Fuego (No Pool)
p = 0.1	0.0	0.864
p = 0.3	0.0	0.864

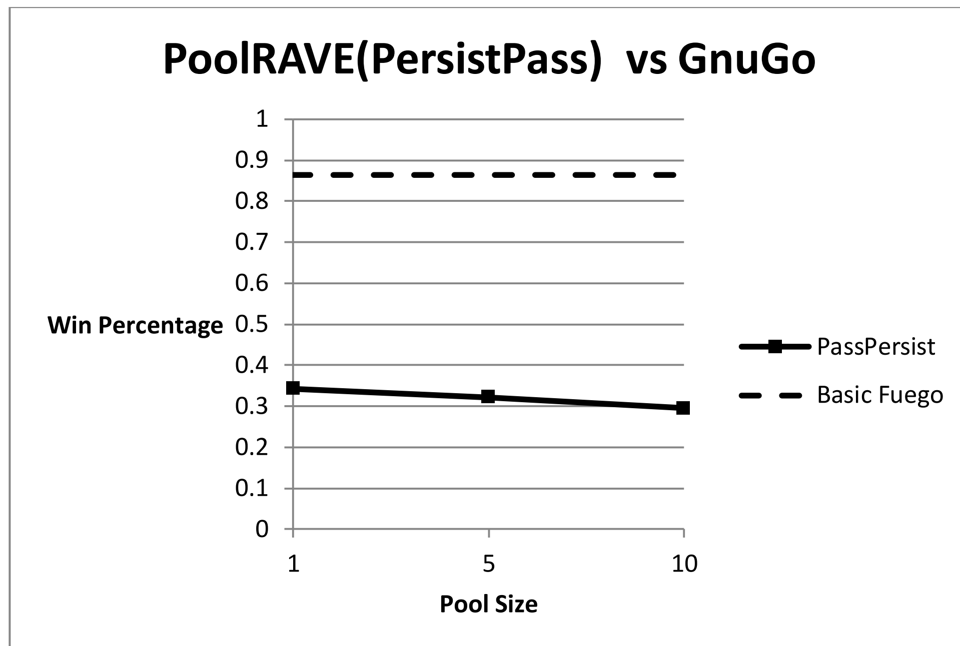
Win Percentage vs. Pool Selection Probability *p* (10s/move)

It was clear that PoolRAVE(Pass) was not going to be a very good algorithm regardless of parameters. There was an intrinsic flaw in its strategy. We felt, however, for the sake of completeness that we test it at least twice at a pool size of one. The results were so discouraging that we continued immediately to PoolRAVE (PersistPass).

4.2.3 PoolRAVE(PersistPass)

Pool size had much greater influence on PoolRAVE(PersistPass) than its predecessor. Rather than pass if the first selected move is invalid, the algorithm iterates through each

item in the pool until it finds a move that can be played, making larger pool sizes much less likely to cause the lethal passes that occurred in PoolRAVE(Pass). Rather than compare the pool selection probability, we chose to explore pool size to a greater degree since pool size would be a much more significant factor in performance. We kept the selection probability p constant at 0.1 and varied with pool sizes of size 1, 5 and 10.



	PoolRAVE(PassPersist) (*p* = 0.1)	Basic Fuego (No Pool)
Size 1	0.34313	0.864
Size 5	0.32222	0.864
Size 10	0.29490	0.864

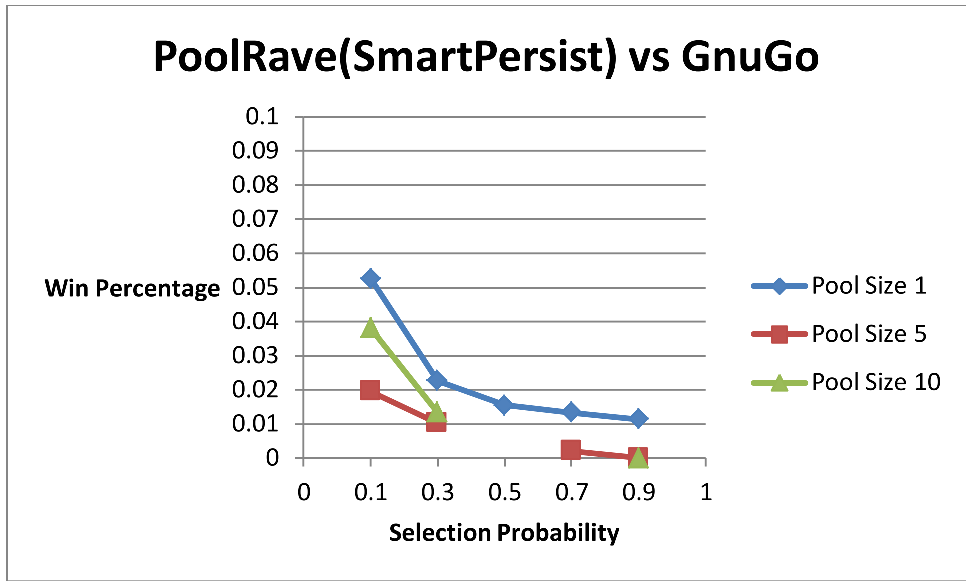
Figure 14 Win Percentage vs. Pool Size (5s/move)

PoolRAVE(PersistPass) shows a clear advantage over the nothing that was PoolRAVE(Pass), winning almost a third of its games on average. An interesting occurrence to note would be how the win percentage decreases as our pool size increases.

One would think that at a low probability of pool selection a larger pool size would help avoid the passing of a turn. It would appear, however, since the move is randomly drawn from the pool by normal distribution that there is a larger probability of drawing a less-than optimal move if the pool contains more moves. Granted these results were only found for a single probability whose effect, though assumed less important than pool size, is still unexplored.

4.2.4 PoolRAVE (SmartPersist)

PoolRAVE(SmartPersist) was our most thoroughly tested of the three variations. We knew that Teytaud had already implemented PoolRAVE in MoGo, but had used a pool selection probability of 100% and focused on pool sizes ranging from five to sixty [5]. We decided to explore smaller pool sizes at varying pool selection probabilities. We tested against GnuGo level 7 with varying selection probabilities and pool sizes ranging from one to ten.



	SmartPersist (Pool Size 1)	SmartPersist (Pool Size 5)	SmartPersist (Pool Size 10)	Basic Fuego (Not Graphed)
p = 0.1	0.052577	0.019588	0.038144	0.864
p = 0.3	0.02268	0.010309	0.013402	0.864
p = 0.5	0.015464	N/A	N/A	0.864
p = 0.7	0.013388	0.002088	N/A	0.864
p = 0.9	0.01134	0	0	0.864

Figure 15 Win Percentage vs Pool Selection Probability *p* (5 sec/move)

While only a few data points were able to be collected due to time constraints, an interesting phenomenon emerged. It would appear from the data that there is a nonlinear correlation between pool size and win percentage. Simply having more moves stored does not improve the performance of the picked move, making a relatively effective pool size a difficult number to find.

4.2.5 Score Correlation between Consecutive Moves

In addition to analyzing the performance of specific algorithms in Fuego, broader exploration was made into computer Go behavior. As Go games progress, many moves are analyzed again and again, while only a few are selected. If there was a better way to show how these moves were incrementally altered, or if they were altered at all, more efficient move selection algorithms could be performed. This could allow for more effective exploration of relevant moves and better score updating for computer Go applications.

Finding and storing the estimated values of non-selected moves was not supported in Fuego, so we altered basic Fuego to print out all of the immediately available moves (and their estimated values) that Fuego looked at during a single turn to an external file. This process was then repeated in a game where two copies of the altered Fuego played against each other and printed out their non-selected moves for a variable number of turns as they played, resulting in several files of data. These files were then read by a separate program that we wrote in the Processing scripting language[16], which displayed the score of each of these non-selected moves on a 9x9 Go game board for each file. This allowed us to visually analyze the variation between moves as opponents updated their scores and explored moves more completely, giving us insight towards the details of the performance of an algorithm.

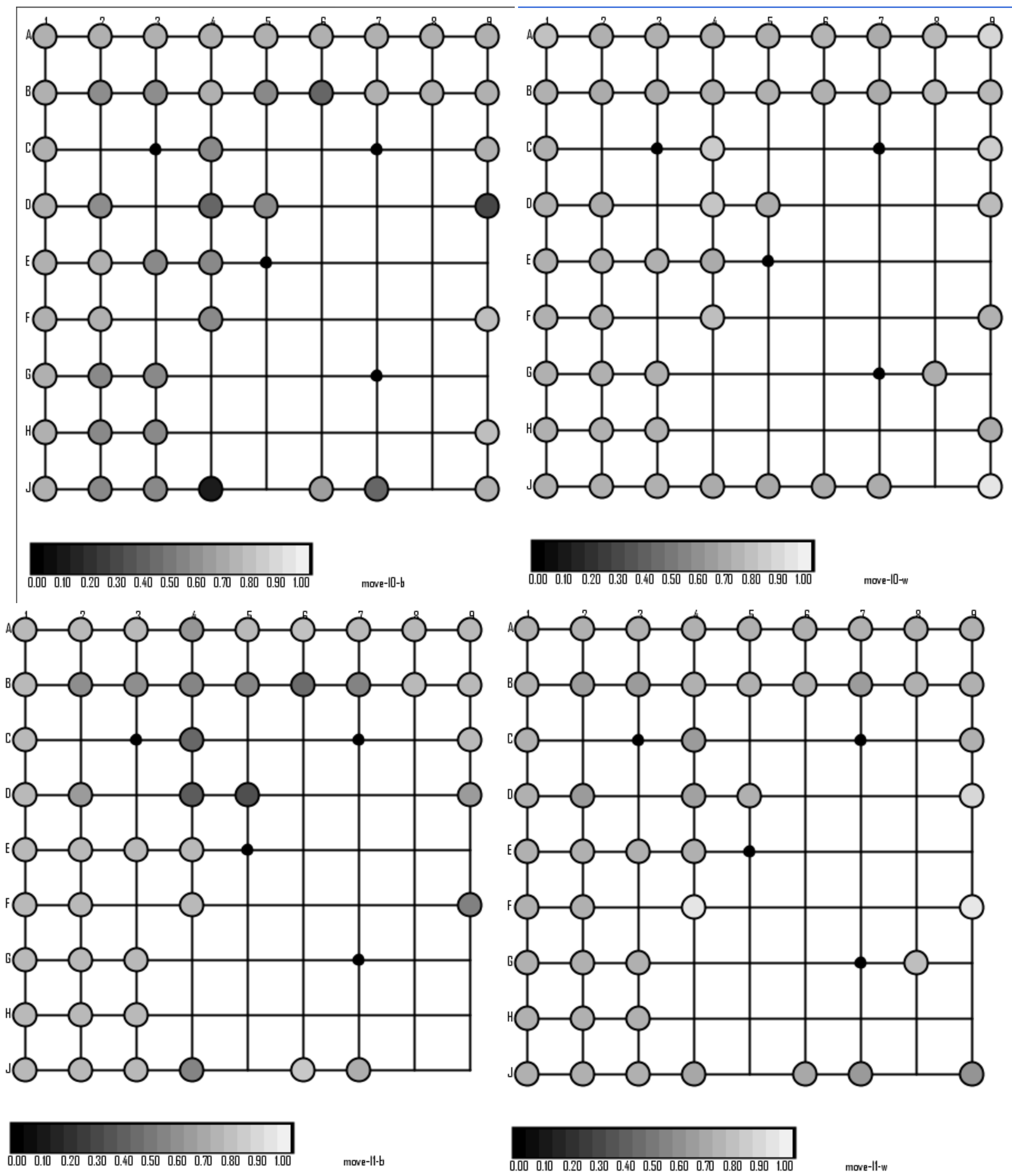


Figure 16: Score Estimate Correlations of Consecutive Moves of a Single Game of Go (Top-Left: Black's 10th move (19th actual move), Top-Right: White's 10th move (20th actual move), Bottom-Left: Black's 11th move (21st), Bottom-Right: White's 11th move (22nd))

4.3 Summary

In this project, we implemented the PoolRAVE algorithm in Fuego and tested variations of its behavior against established benchmarks. The performance of Fuego has improved since the work of Schaeffer & Bjorge, and while our variations were not comparable to that improvement, we still supplied several insights to the details of Rave-Pool performance in Fuego. We also developed a new method of how to explore move-score correlation so that the intricacies of algorithm behavior could be better understood.

Further exploration is always necessary, and the many untested parameter settings of PoolRAVE are no exception. Higher resolution into smaller pool selection probabilities could lead to interesting further results, as well as large pool sizes. PoolRAVE still has the potential to be a very useful iteration of UCT-RAVE, which is itself a very successful algorithm, and such potential should not be ignored.

5 Conclusions

5.1 The Gomba Testing Framework

The first major contribution of our project was our improvement of the accuracy of the Gomba Testing Framework, specifically for algorithms which rely on move transpositions. The Gomba Testing framework now keeps and updates a global knowledge of the moves in the Game State nodes, along with the correlations between the moves which are considered as a new property of the Gomba tree. Gomba is an open, simple framework to allow for the testing of massive game trees, and though it is still under development we hope that it will be a useful tool for future research into the performance of new search variants.

5.2 Fuego

Fuego is already an established framework which has long since proven its usefulness, and our contribution to it was to simply build upon that recognition. Our implementation of PoolRAVE(Pass), PoolRAVE(PassPersist), and PoolRAVE(SmartPersist) brought useful insight into the use of smaller move pools as a method to improve game performance.

Additional exploration into the parameters of the existing algorithms is always a noble pursuit. Though it is a niche area, exploration into how condor activity affects test results would certainly help fortify or rebuke findings that occurred within its system. Also,

further analysis of Move Correlation would be beneficial to the understanding of move exploration and to the implementation of better move recycling.

5.3 Future Work

While the Gomba testing framework using the new tree generation algorithm improved the performance of the AMAF algorithms (UCT RAVE) at a certain level, the result did not meet our ambitions. The conclusion from Gelly *et al.* indicated that the UCT-RAVE algorithm always outperformed the regular UCT algorithm in a real Go game [11], and the optimal value of the equivalence parameter would be around 100.

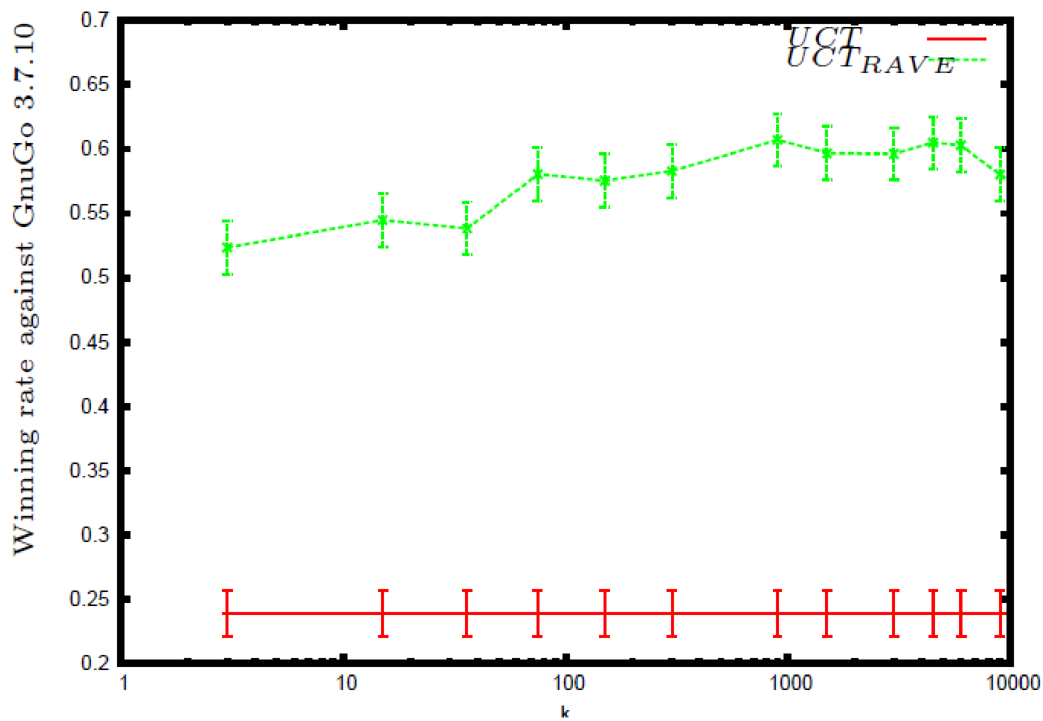


Figure 17 Winning rate of UCT-RAVE vs UCT [11].

Though our testing result using the modified Gomba testing framework improved the performance of UCT RAVE algorithm at a certain level and also suggested that the optimal value of the equivalence parameter was around 100, the overall performance of UCT RAVE was still inferior to the regular UCT algorithm. The exact reason causing this problem was unclear. We thought there might be hidden factors other than the values suggested in section 3.1.3 that we didn't discover which could affect the correlation between the moves.

Some potential future work includes:

- Continue work on correlation between the moves. In this project we adjusted the structure and improved the Gomba testing framework on AMAF-type algorithms at a certain level. However, there was still potential improvement in the testing framework itself.

- Find possible factors that may relate to the correlation between the moves. In this modified version of Gomba framework, we did not use a transposition table to remember the global statistics of the moves. Instead, we let the Game State nodes remember the information itself and swap the values when necessary. We were not clear whether this may affect the correlations, but one thing for sure was that there were other factors which we had not discovered yet that were also related to the correlation of the moves. Identifying these factors would allow more effective move transposition and improve the accuracy of the testing framework.

- Use Fuego to explore how PoolRAVE improves compared to the number of simulations per move, rather than a flat time limit per move.

- In this project, we mainly focused on the AMAF type of algorithms (UCT RAVE) which relied on the move transpositions. It would be also valuable to adjust the structure of testing framework to test other new types of search algorithms for future research.

6 References

[1] D. Bjorge, and J. Schaeffer, *Monte-Carlo Search Algorithms*, Worcester Polytechnic Institute, 2010.

[2] G. I. Lin, *Fuego Go: The Missing Manual*, 2009.

[3] S. Gelly, and D. Silver, "Achieving Master Level Play in 9x9 Computer Go," *Proceedings of AAAI*, pp. 1537-1540, 2008.

[4] L. Kocsis, and C. Szepesvari, "Bandit based Monte-Carlo planning," *15th European Conference on Machine Learning*, pp. 282-293, 2006.

[5] A. Rimmel, F. Teytaud, and T. Olivier, "Biasing Monte-Carlo Simulations through RAVE Values," in The International Conference on Computers and Games 2010, 2010.

[6] M. Enzenberger. "The Integration of A Priori Knowledge into a Go Playing Neural Network," 2011; http://www.cgl.ucsf.edu/go/Programs/neurogo-html/neurogo.html.

[7] O. Teytaud, and J.-B. Hoock, "Bandit-Based Genetic Programming," in 13th European Conference on Genetic Programming, 2010.

[8] B. Brügmann, "Monte Carlo Go," 1993.

[9] C. Chaslot, S. Bakkes, I. Szita *et al.*, "Monte-Carlo Tree Search: A New Framework for Game AI," 2009.

[10] S. Gelly, Y. Wang, R. Munos *et al.*, "Modification of UCT with patterns in MonteCarlo go," *Technical Report RR-6062, INRIA*, 2006.

[11] S. Gelly, and D. Silver, "Combining Online and Offline Knowledge in UCT," *Proceeding 24th International Conference on Machine Learning (ICML)*, pp. 273-280, 2007.

[12] T. David, and M. Müller, "A Study of UCT and its Enhancements in Artificial Game," in ACG, 2009, pp. 55-60.

[13] D. P. Helmbold, and A. Parker-Wood, "All-Moves-As-First Heuristics in Monte-Carlo Go," *Proceedings of the 2009 International Conference on Artificial Intelligence, ICAI*, pp. 605-610, 2009.

[14] B. Wilson. "The Machine Learning Dictionary," http://www.cse.unsw.edu.au/~billw/mldict.html.

[15] B. Childs, J. Brodeur, and L. Kocsis, "Transpositions and Move Groups in Monte Carlo Tree Search," *IEEE Symposium on Computational Intelligence and Games*, pp. 389-395, 2008.

[16] *B. Fry*, and *C. Rea*s. "Processing.org," http://processing.org/.

Appendix A: Gomba Experiment Parameters

Our experiments used Condor cluster graciously provided by MTA-SZTAKI. We used many runs of many trials each to provide statistically sufficient results. The parameter listings that follow are divided into related sets. For details on how the parameters defines the resulting experiments, see Appendix B.

Set 1: Comparative on different equivalence parameters

- 2000 trials
- 100,000 iterations
- 0 base difficulty
- Algorithms
  - UCT (1.1 FPU)
  - UCT RAVE (1.1 FPU, 1 equivalence parameter)
  - UCT RAVE (1.1 FPU, 5 equivalence parameter)
  - UCT RAVE (1.1 FPU, 10 equivalence parameter)
  - UCT RAVE (1.1 FPU, 100 equivalence parameter)
  - UCT RAVE (1.1 FPU, 1000 equivalence parameter)
- Tree sizes:
  - Branching factor 30, depth 6

Set 2: Comparative on level of correlations

- 2000 trials

- 100,000 iterations
- 0 base difficulty
- Algorithms
    - UCT RAVE (FPU 1.1, 100 equivalence parameter, 0 correlation)
    - UCT RAVE (FPU 1.1, 100 equivalence parameter, 25% correlation)
    - UCT RAVE (FPU 1.1, 100 equivalence parameter, 50% correlation)
    - UCT RAVE (FPU 1.1, 100 equivalence parameter, 75% correlation)
    - UCT RAVE (FPU 1.1, 100 equivalence parameter, 100% correlation)
- Tree sizes:
    - Branching factor 30, depth 6

Appendix B: Gomba Developer's Primer

The Gomba search framework is freely available for general use under the Apache 2.0 license. A current copy can be found at http://www.apache.org/licenses/LICENSE-2.0. The source for the project can be found at Worcester Polytechnic Institute Library e-Project Catalog. This developer's primer is written by Daniel J. Bjorge and John N. Schaeffer from last year's MQP [1]. We included it here for the convenience of the reader.

Using Gomba

The general syntax to simulate trials with the Gomba framework is as follows:

gomba-mqp <options> <algorithm_1> [<alg_param_1>[,<alg_param_2>]…] <algorithm 2>

Some common Gomba options include:

-n: Number of trials (distinct trees) to test the algorithms against.

-i: Number of the iterations to run each algorithm for.

-b: The branching factor to construct the tree with. Non-terminal nodes will have this many children.

-d: The depth of the generated tree (the largest distance from any node to the root).

-B: The difficulty bias. This may be any floating point number. Zero means "no bias", lower means better for the minimizing (starting) player, and higher means worse for the starting player.

Algorithm options are specific to each algorithm. The included algorithms which are based on RolloutMonteCarlo will generally have at least two parameters, the first two

being the simulation policy effectiveness and the reward propagation constant. Those based on UCT will generally have at least a third, the first play urgency constant.

Example

gomba-mqp -n100 -i100000 -b2 -d20 random uct0,1,1.1 uct0,1,10000

This would run 100 trials of 100000 iterations each on a tree of branching factor 2 and depth 20. The results would compare a purely random search and two UCT searches, one with a first play urgency value of 1.1 and one with no first play urgency (represented by the massive FPU value of 10000).

Parsing Results

Gomba outputs a series of comma-separated-value text les named <algorithm>.dat for each algorithm specified as input. Each of these files contains one line per iteration. Each line contains the following values from its respective iteration:

1. The number of trials in which an optimal move was chosen
2. The sum of the difficulties at each chosen node
3. The total number of tree nodes expanded
4. The total number of elapsed clock cycles

Each of these values is simply the sum of the respective value from each trial. This was chosen over using averages primarily to ease the merging of multiple output files - it allows for multiple runs of the program with the same parameters to be combined simply by adding the values in the output les component-wise, which is very useful in splitting jobs across nodes in a computing cluster.

The data files can be parsed and analyzed by any program which can read CSV text files. We primarily used the R statistical programming language to generate the statistics used in this report, and have included several example R scripts in the scripts directory of the Gomba source tree. Like the framework itself, these are released under the Apache 2.0 license and may be freely used under its terms and conditions.

Adding Search Algorithms

The general strategy for the introduction of a new algorithm into the existing code-base is as follows:

1. Create a new class which derives from the SearchAlgorithm class, located in search/SearchAlgorithm.h. We recommend that variants of existing algorithms derive from those existing algorithms where feasible. In particular, the RolloutMonteCarlo class, on which most of the algorithms presented in this report are based, provides a great deal of groundwork code which is shared between all algorithms using a Rollout-based Monte-Carlo strategy. This includes, for example, most UCT variants. We highly recommend using the existing variants (such as UCT) as examples, and we also highly recommend checking the documentation within the RolloutMonteCarlo class for descriptions of how to modify the parts of the algorithm your particular variant changes. However, if you are implementing a truly novel new algorithm, it is only necessary that it adhere to the function documentation specified by the SearchAlgorithm interface.

2. Register your class in the algorithms/AllAlgorithms.h header. You can do this by calling any of the registration macros available from search/AlgorithmRegistration.h in between the BEGIN_REGISTER_ALGS; and END_REGISTER_ALGS; calls. Use the existing registrations as a template. The most common usage pattern for a search algorithm with a constructor that takes K arguments is REGISTER_ALG_K(string_name, ClassName), which will allow the framework to map a command line algorithm specication of the form "string_name1,2,...,K" to a search algorithm constructed with the call "Class-Name(1, 2, ..., K)".

3. Recompile the Gomba framework with your new algorithm in place.

4. Run the resulting executable (by default, "gomba-mqp") with the command line algorithm specification you deined by registering your algorithm. For example, if your algorithm class MyAlgorithm has a constructor of two arguments which you registered in step two with "REGISTER_ALG_2(myalg, MyAlgorithm)", you can start a simulation with the command "gomba-mqp myalg0,1". You can use whatever parameters you like for myalg (they are treated as doubles), run it against any other algorithms in the framework (others of your creation or any of the standard included ones), and modify any of the standard framework parameters (-n, -b, -d, etc.) for the run. In short, after registration, it is treated exactly like any one of the standard included algorithms.

Appendix C: Fuego Experiment Parameters

The following experiments used the Condor cluster provided by MTA-SZTAKI. The listings are divided into related parameter settings and organized by order of appearance in the report. Both Black and White results are given not for verbosity but to note the number of games lost to error. The test opponent is GnuGo level 7 unless otherwise noted. Time per move is measured in seconds.

Basic Fuego (RAVE)

Opponent	Time per Move	Number of Games	Wins for Fuego	Wins for GnuGo	% Wins for Fuego
GnuGo 6	5	1000	866	134	86.6
GnuGo 7	5	1000	877	123	87.7
GnuGo 6	10	1000	885	115	88.5
GnuGo 7	10	1000	864	136	86.4

PoolRAVE (Pass)

Pool Size	Pool Prob.	Time per Move	Number of Games	Wins for Fuego	Wins for GnuGo	% Wins for Fuego
1	0.1	10	1000	0	153	0
1	0.3	10	1000	0	916	0

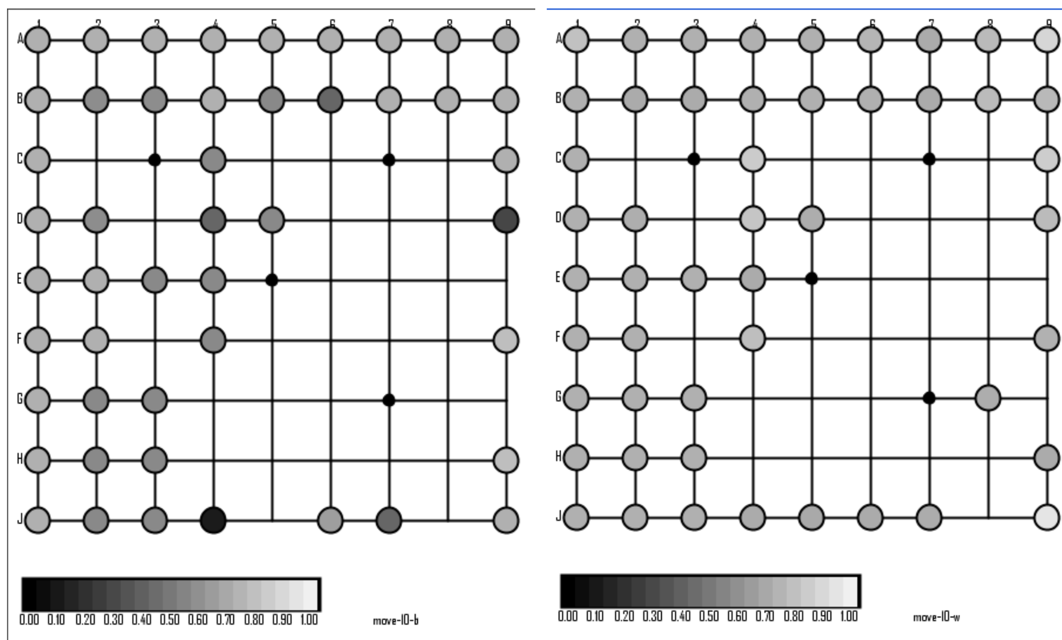
PoolRAVE (PersistPass)

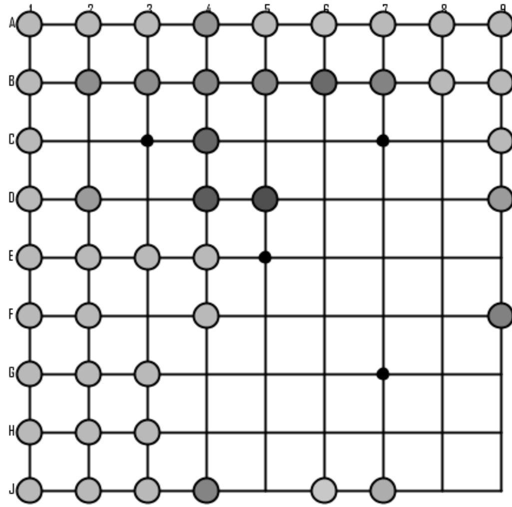
Pool Size	Pool Prob.	Time per Move	Total Games	Wins for Fuego	Wins for GnuGo	% Wins for Fuego
1	0.1	10	510	175	311	0.343
5	0.1	10	450	145	304	0.322
10	0.1	10	373	110	259	0.294

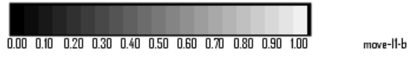
PoolRAVE (SmartPersist)

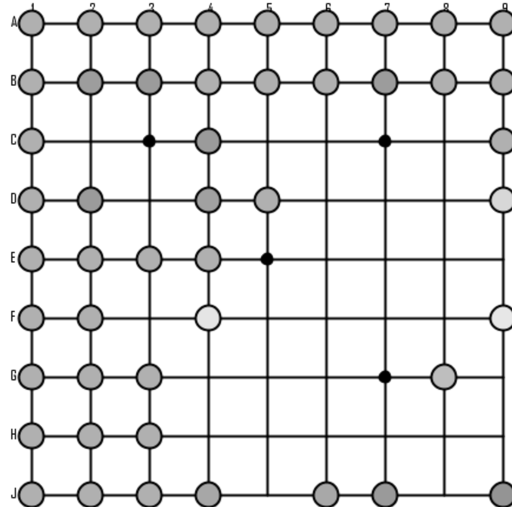
Pool Size	Pool Prob.	Time per Move	Games	Wins for Fuego	Wins for GnuGo	% Wins for Fuego
1	0.1	5	970	51	919	0.052577
1	0.3	5	970	22	948	0.02268
1	0.5	5	388	6	382	0.015464
1	0.7	5	971	13	958	0.013388
1	0.9	5	970	11	959	0.01134
5	0.1	5	970	19	951	0.019588
5	0.3	5	970	10	960	0.010309
5	0.7	5	958	2	955	0.002088
5	0.9	5	901	0	901	0.0
10	0.1	5	970	37	933	0.038144
10	0.3	5	970	13	957	0.013402
10	0.9	5	970	0	970	0.0
1	0.1	10	1000	52	948	5.2
10	0.5	10	997	5	991	0.5015

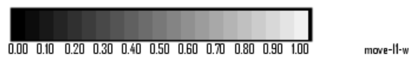
Move Concurrency




1
2
3
4
5
6
7
8
9
A
B
C
D
E
F
G
H
J


0.00 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 1.00
move-11-b


1
2
3
4
5
6
7
8
9
A
B
C
D
E
F
G
H
J


0.00 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 1.00
move-11-w