MQP CDR#GXS1002

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 Degree of Bachelor of Science by

Daniel J. Bjorge

John N. Schaeffer

April 28, 2010

Professor Gábor N. Sárközy, Major Advisor

Professor Stanley M. Selkow, Co-Advisor

Abstract

We have developed the Gomba Testing Framework, a new platform for the comparative evaluation of search algorithms in large adversarial game trees. Gomba is simple, fast, extensible, objective, and can scale to larger trees than previous frameworks have been able to test against. We have implemented and tested a variety of Monte-Carlo based search algorithms using the framework and have analyzed their performance in relation to known metrics in computer Go. Finally, we have taken several solutions to the infinitely-many-armed bandit problem and adapted them to tree search. We have tested these variants in both Gomba and computer Go, and have shown that they can be effective in both cases.

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
- *Fuego* Development Team
- Gábor Recski and Attila Zseder, SZTAKI Colleagues

Contents

Abstract **i**

Acknowledgements **ii**

Table of Contents **iii**

List of Figures **v**

List of Algorithms **vi**

1 Background **1**
1.1 Introduction . . . 1
1.2 Adversarial Search . . . 2
1.2.1 Performance Improvements . . . 3
1.3 Go . . . 4
1.3.1 Current Techniques . . . 5
1.4 Multi-Armed Bandit Problems . . . 6
1.4.1 Searching in MABs: UCB1 and UCT . . . 7
1.5 Artificial Game Trees . . . 8

2 Artificial Game Trees **11**
2.1 Requirements . . . 11
2.1.1 Lazy State Expansion . . . 11
2.1.2 Deterministic State Expansion . . . 11
2.1.3 Pseudorandom State Expansion . . . 11
2.1.4 Predetermined State Optimality . . . 12
2.1.5 Fast Action Simulation . . . 13
2.1.6 Fast Termination Evaluation . . . 13
2.1.7 Fast Heuristic Evaluation . . . 13
2.1.8 Go-Like Action-Reward Distribution . . . 13
2.2 Realization . . . 14
2.3 Properties . . . 14
2.3.1 Lazy Deterministic Expansion . . . 15
2.3.2 Predetermined Minimax Values . . . 16
2.3.3 Speed . . . 19
2.4 Weaknesses . . . 20
2.4.1 Weak Randomness . . . 20
2.4.2 Random Action-Reward Distribution . . . 21
2.5 Conclusions . . . 22

3 Gomba **23**
3.1 Framework Requirements . . . 23
3.1.1 Modularity . . . 23
3.1.2 Simplicity . . . 24
3.1.3 Rapid Node Generation and Access . . . 24
3.1.4 Scalability to Go-Sized Trees . . . 25

3.1.5 Simple Parametrization . . . 25
3.1.6 Reliable, Deterministic Results . . . 25
3.2 Implementation . . . 26
3.2.1 Node Structure . . . 26
3.2.2 Memory Management . . . 27
3.2.3 Random Number Generation . . . 28
3.2.4 Monte-Carlo Simulation . . . 29
3.3 Conclusions . . . 35

4 Experiments **37**
4.1 Previously Tested Algorithms . . . 37
4.1.1 Random Search . . . 38
4.1.2 Minimax Search . . . 38
4.1.3 Random Monte-Carlo Search . . . 39
4.1.4 UCT . . . 39
4.1.5 UCT-Tuned and UCT-V . . . 42
4.1.6 UCT-FPU . . . 45
4.1.7 Heuristic UCT . . . 50
4.1.8 UCT-RAVE . . . 52
4.2 Tree Properties . . . 52
4.2.1 Difficulty Bias . . . 53
4.3 Untested Algorithms . . . 57
4.3.1 k-Failure . . . 57
4.3.2 k-Failure Variants . . . 59
4.3.3 UCT-V(∞) . . . 59
4.3.4 Meta-EvE . . . 63
4.3.5 Probabilistic Minimax Monte-Carlo . . . 63
4.4 Conclusions . . . 67

5 Fuego **68**
5.1 The Existing Codebase . . . 69
5.1.1 UCT Player . . . 69
5.1.2 GTP Engine . . . 69
5.2 Implementation Details . . . 70
5.3 Experiments . . . 71
5.3.1 GNU Go . . . 71
5.3.2 Testing Parameters . . . 72
5.3.3 Results . . . 72
5.4 Conclusions . . . 73

6 Conclusions **75**
6.1 The Gomba Search Framework . . . 75
6.2 Infinite-Armed-Bandit Based Search . . . 75
6.3 Future Work . . . 75

References **77**

Appendix A: Gomba Experiment Parameters **79**

Appendix B: Gomba Developer's Primer **86**

Appendix C: Fuego Experiment Parameters **90**

List of Figures

1 UCT Performance in Small Gomba Trees 41
2 UCT Performance in Large Gomba Trees 41
3 Variance-Based UCT Performance in Small Gomba Trees 44
4 Variance-Based UCT Performance in Large Gomba Trees 44
5 UCT-FPU Performance in Small Gomba Trees 46
6 UCT-FPU Performance in Large Gomba Trees 46
7 FPU Performance (b=2, d=20) 47
8 FPU Performance (b=6, d=6) 47
9 FPU Performance (b=8, d=8) 48
10 FPU Performance (b=30, d=6) 48
11 FPU Performance (b=50, d=50) 49
12 FPU Performance (b=200, d=50) 49
13 Simulation Policy Effectiveness in Small Gomba Trees 51
14 Simulation Policy Effectiveness in Large Gomba Trees 51
15 Effects of Difficulty Bias in Gomba Trees (b=2, d=20) 55
16 Effects of Difficulty Bias in Gomba Trees (b=8, d=8) 55
17 Effects of Difficulty Bias in Gomba Trees (b=30, d=6) 56
18 K-Failure Performance in Small Gomba Trees 58
19 K-Failure Performance in Large Gomba Trees 58
20 UCT-AIR Performance in Small Gomba Trees 62
21 UCT-AIR Performance in Large Gomba Trees 62
22 Probabilistic Minimax Monte-Carlo Effectiveness in Small Gomba Trees 66

23 Probabilistic Minimax Monte-Carlo Effectiveness in Large Gomba Trees 66

List of Algorithms

1 Gomba Tree Generation 14
2 Probabilistic Minimax Update 64

1 Background

1.1 Introduction

Intelligent search forms the basis of much of modern artificial intelligence research. Most AI problems can be reduced to searching for an ideal solution from a known set of potential solutions to a problem. Speech recognition is a search for the phrase which best represents a particular sequence of sounds. Proof generators are effectively search engines for sets of logical premises. Game playing, too, may be reduced to a search problem where the goal is to find a sequence of moves which will result in a winning game state.

Our project's broad goal is the improvement of computer search performance in adversarial game trees. In particular, we consider performance in trees which are too large for the application of traditional exhaustive search methods, as is the case in most "interesting" games such as chess and Go. We first present the Gomba (Go-based Monte-Carlo Bandit Analysis) Testing Framework, a new testing platform which we have developed to measure the performance of search algorithms for two-player adversarial game trees. This framework has been designed with extremely large trees in mind, and is capable of evaluating new search strategies simply, quickly, objectively, and scalably. This combination has previously only been feasible with relatively small game trees, which makes our framework a significant new contribution not just for our own testing but for future researchers to use in the evaluation of their own strategies.

With this new framework in hand, we showcase a variety of experimental results obtained by testing several Monte-Carlo based search strategies which have proved to be effective in computer Go playing against the Gomba framework. We compare the results in our framework to those claimed in previous research and to those seen in current practical computer Go engines. After completing these control experiments, we proceed to implementing several new variant search strategies in our framework.

We then take the most promising of these new variants and implement them in an existing successful computer Go engine, Fuego, to compare their performance in our artificial game tree engine with performance in a real-world application.

1.2 Adversarial Search

In artificial intelligence, some of the oldest problems revolve around adversarial search. In a regular search process, an agent is simply attempting to search through a (non-hostile) environment for a solution to a problem, but adversarial search extends this concept to include aspects of the environment seeking to hamper the agent's progress. Thus, the agent's goal becomes not just to find a solution, but to find a solution where it wins against its adversary or adversaries.

To use such techniques in a game, the entire set of states and actions of that game may be represented as a tree structure where each node corresponds to a game state and each edge corresponds to an action from that game state. The overall root of the tree is the current state of the game, thus providing a convenient representation of all states and actions that can occur within this game. Additionally, states may have a notion of value attached to them such that one can determine the winning player's and losing player's scores. Once the game is encoded in this fashion, the goal of the search algorithm is clear: it must find the move which will place itself in a position such that it will maximize its value and, ideally, win. Doing this for every move is known as playing optimally.

The current fastest strategy for optimal play, known as minimax [14], is based on a fairly simple recursive process of *mini*mizing the opponent's reward while *max*imizing the current player's reward at any given time. This is done by choosing the node with the lowest minimax value when the opponent plays and the node with the highest minimax value when the current player plays. From there, the optimal move is determined for not just the current time step, but for all subsequent time steps, effectively solving the game. However, all is not as well as it might seem: the minimax strategy requires $O(b^d)$ time to complete for a game with branching factor b and depth d due to visiting all nodes in the tree. For many games, this approach is simply infeasible.

1.2.1 Performance Improvements

It is not typically feasible to use a true minimax policy in practice because of the exponential time constraint involved in obtaining a usable result. There are several known optimizations to the basic minimax algorithm which can improve drastically on the amount of time required, but there is no known method which can generate perfect results in less than exponential time. For example, Alpha-Beta Pruning [14] is a common approach in the early stages of move computation in chess AI - it is a minimax policy which keeps track of the best and worst subnodes currently found underneath nodes earlier in the tree. Because a node is suboptimal if all of its child nodes are suboptimal, this data can be used to determine that certain nodes are suboptimal before fully expanding their children. This resulting in a time complexity of $O(b^{\frac{d}{2}})$ instead of normal Minimax's $O(b^d)$. This is a significant improvement, but not so significant as to make a fully optimal search of this nature feasible for a game with a branching factor as large as Go's. Other strategies must be considered.

One such strategy is the use of heuristics. Many board states in a game can be said to have a certain value. In Chess, for example, a board where the player's queen is facing capture is much less desirable (and thus has a lower value) than one where the player is set to capture the opponent's queen. From properties of specific games, one can thus assign a value to a board state or class of board states, and use these to infer which moves are "better". This can bring many games into a realm where computers may feasibly play them well, but at a price: heuristics can often introduce biases into games, and are not guaranteed to be accurate. Building on the previous Chess example, sacrificing pieces that may have high value could lead to victory for the player–something which may not be immediately apparent from the value a heuristic may give. This means the use of heuristics can also introduce suboptimal play as well. However, given the time complexity of finding an optimal move, suboptimal play is often the only option.

The branching factor of many games can often be so high that visiting every possible move from a given state is prohibitively expensive. The game Go is a notable example of this: with an average branching factor of roughly 200, evaluating all possible moves will involve rather significant overhead for relatively little reward. In many games, though, the optimal move from a state often belongs to a class of near-optimal but suboptimal moves. Because of this property, it is possible to find moves that are in the general "neighborhood" of the optimal move, even if the optimal move is not found. A recently successful area of research in computer Go playing is Monte-Carlo search algorithms, which use repeated random sampling to seek out moves of acceptable quality rather than relying on testing every possibility to find a move of the absolute highest quality. When combined with heuristics, Monte-Carlo search algorithms may be able to find good moves with impressive speed [19]. It is important to note, though, that not all policies will guarantee finding the best move. For this reason the means of selecting the next move to be evaluated is a very important factor in the success of a Monte-Carlo search algorithm.

1.3 Go

Both quite ancient and quite popular, Go has attracted the fascination of mathematicians and game enthusiasts alike for centuries [13]. The game itself is played on a square board–usually 9x9 or 19x19 cells–which starts off in a blank state and is filled in through turn-based placement of black and white stones from the first (black) and second (white) players. By placing stones on the board, players form groups of stones which are connected horizontally or vertically in order to form walls and secure territory on the board. Skillful placement of stones can also lead to the capture of enemy pieces: by surrounding an opponent's group of stones such that it has no way to expand, all stones in that group are captured and removed from play.

Players may either place a stone on the board when their turn has arrived or pass, at which point a responding pass from the other player will end the game. A player may place a stone anywhere on the board, provided it does not violate two conditions. The first of these is called *suicide*, defined as any move which would cause the immediate capture of that stone or the group to which it would be attached. The second is called *ko* or *superko* [13], and is a much more subtle issue. The *ko* rule states that no player may make a move which would leave the board in the same state that it was in one *ply* ago, where a *ply* is a set of one move per player. *Superko* is an extension to the ko rule which simply states that the board should never enter the same state twice in a single game.

1.3.1 Current Techniques

As may be inferred by the easily satisfiable conditions for placing stones on the board, as well as the size of the board, Go game trees are typically very large. The branching factor of 19x19 Go game trees is roughly 200, which is far from feasible for any exhaustive search. In addition to the large branching factor, Go lacks an opening book in the same way that Chess does–the game is simply too flexible to determine rigorously what the best opening moves are. Interestingly enough, the properties of the game which make it so difficult for computer players are quite easy for human players. Humans can more effectively eliminate whole swaths of moves and discern good moves from bad ones without much conscious processing. Computers, on the other hand, are in the unfortunate position of needing to evaluate (to some degree) each move before learning its value.

The problems computers have had with Go spurred a reevaluation of how computers should play the game. Early on, many researchers looked to Chess playing programs for inspiration, but today most of the emphasis is on Monte-Carlo methods, pattern-matching, and heuristics for determining moves [3]. These strategies are based on the previously described notion of finding a move that is acceptably good instead of relentlessly searching for the optimal move, which brings computer Go into the realm of feasibility once more. Current programs make use of many or all of these techniques, such as MoGo [19], which incorporates prior knowledge and handcrafted heuristics into its decision-making process. The best known programs, though, attempt to exploit the large branching factor of Go by assuming there will be several moves near the optimal move which are acceptably good choices. This assumption opens up searching for a good move to a class of solutions to what are known as multi-armed bandit problems.

1.4 Multi-Armed Bandit Problems

In 2006, computer Go playing was revolutionized by the application of Monte-Carlo based search methods based on solutions to the Multi-Armed Bandit problem [15]. A multi-armed bandit problem is defined as a series of K machines $X_1, ..., X_K$ with random pay sequences $X_{i,1}, ...X_{i,t}$ for some $i \in \{1, ..., K\}$ and time steps $\{1, ..., t\}$ [15]. Of these K machines, we will call any machine at index m *optimal* if it satisfies $m = argmax_{i \in \{1,...,K\}}\{\mathbb{E}[\sum\limits_{j=1}^{t} X_{i,j}]\}$. That is, m is considered optimal if the expected value of its total reward up until time t is the maximum of all machines' expected total payoffs. Searching for an optimal machine, though, may require trying many suboptimal machines first, which will lead in turn to suboptimal rewards. Thus, there is a sense of regret implicit in these problems which comes from suboptimal play. Regret is defined as the expected loss from not playing the optimal machine m at all time steps $1, ..., t$. In other words, $R_n = \mathbb{E}[\sum_{j=1}^{t} X_{m,j}] - \mathbb{E}[\sum_{i=1}^{K} \sum_{j=1}^{T_i(t)} X_{i,j}]$. $T_i(t)$ is defined here as the number of times machine i has been played after t plays. The goal of the multi-armed bandit problem is to minimize the regret after t plays of the set of machines.

In order to minimize regret, it is prudent to quickly find the machine X_m, or find machines that are close to it in the process. Finding X_m definitively requires extensive exploration through many machines, but minimizing R_n requires exploiting X_m early and often. This balance between searching for the best machine and exploiting it frequently is known as the *exploration-exploitation trade off [11].* A machine selection policy π is said to resolve this trade off if it can successfully keep the regret rate at $O(R_n^*)$ where R_n^* is the best possible regret rate [15].

1.4.1 Searching in MABs: UCB1 and UCT

UCB1 [17] (Upper Common Bound) is based on a fairly intuitive assumption: for a set of K machines with mean rewards $\mu_{i,n} = \mathbb{E}[\sum_{j=1}^{n} X_{i,j}]$ and optimal mean μ^*, there should be some machines $\{X_i : \mu_{i,n} \geq \mu^* - \delta\}$ for an acceptable threshold δ. Rewards are bounded in this case in $[0, 1]$ as well. If the optimal machine cannot be found in time, UCB1 should find other (suboptimal) machines whose reward distributions are close enough to optimal.

At n plays, UCB1 selects the machine X_m which satisfies the following:

$$m = argmax_{i \in \{1,\dots,K\}}\{\mu_{i,n} + c_{i,n}\}, \text{ where} \tag{1}$$

$$c_{i,n} = C\sqrt{\frac{ln(n)}{T_i(n)}}. \tag{2}$$

In this case, $c_{i,n}$ is a bias sequence which decreases as a function of the number of times the machine X_i is chosen, and C is some exploration constant (usually set to $C \approx \sqrt{2}$). With this balance between the bias sequence and mean reward values, UCB1 is able to resolve the exploration-exploitation trade off with a relatively small amount of time, with guaranteed convergence to the optimal machine given enough time [15].

In order to extend UCB1 to game trees, Levente Kocsis and Csaba Szepesvári devised UCT (UCB1 for Trees) in 2006 [15]. This algorithm treats each internal node of the tree as a separate UCB1 problem where bias terms $c_{i,n}$ are multiplied by the depth of each node relative to the starting depth of the search. Their actual planning algorithm searches recursively through the tree until either a terminal state or a certain depth is reached, at which point some evaluation method is used instead of playing the game further. Like UCB1, UCT is guaranteed to converge to the optimal move given enough time, and additionally has the property of quickly discovering a move that is acceptably close to optimal.

UCT can be advantageous for many reasons, particularly in Computer Go. Because of its rapid convergence to a small set of moves with high reward values, it expands a relatively small portion of the tree. In games with large branching factors this provides a crucial performance increase, as unnecessary exploration may lead to lower quality moves and higher resource consumption. However, UCT is not without its caveats. Its parent, UCB1, requires that any machine not yet played have a bias term of $c_{i,n} = \infty$, ensuring that such a machine will be selected next. This means that in UCT all nodes must be explored at least once, which causes much slower convergence with very high branching factors (such as 9x9 and 19x19 Go). Because of this, some modifications to UCB1 have been proposed, which will be discussed later.

1.5 Artificial Game Trees

Researchers in the field of computer Go have often used artificial game tree frameworks to test new search methods before trying to apply them to Go itself [18, 15, 7]. Go is not a very useful initial metric for such testing for several reasons:

- 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.

The latter two simply make evaluation slow, but the first makes useful comparisons between algorithms nearly impossible. Because there is no precomputable metric of how "good" any particular move is, the only real performance metric we can use to determine algorithm accuracy is to test the series of algorithms in question in games against one known good algorithm (or against each other).

This has several disadvantages. The first is speed. The best computer Go programs will generally take on the order of an hour to finish a game even on very powerful hardware. This makes gathering a set of test results of sufficient size for statistical analysis extremely daunting. The second disadvantage is that the results themselves are not exactly valid measurements of how good an algorithm is at playing Go in general, but rather how well it plays against the particular opponent algorithm. Since the best computer Go programs are currently not competitive with the best human players, this is a fairly serious flaw in testing methodology.

These disadvantages have in the past encouraged researchers to use artificial game trees instead of game trees from the games they were actually researching. This allows parameters such as branching factor and game depth to be changed very easily, and it also can speed up heuristic calculations by orders of magnitude. In particular, being able to modify tree parameters means that it can be made feasible to run a completely optimal search, such as alpha-beta, to determine with perfect accuracy the optimality of moves. This in turn allows for a much more objective measurement of performance, since "Does algorithm A tend to choose optimal moves?" is not dependent on any algorithm B.

A weakness of past artificial game tree generators, such as Kocsis's PGame [15], is that in order to calculate such minimax optimality values an unrealistic portion of the game tree ($O(\sqrt{b^d})$ nodes) must be checked. For smaller trees this is not a problem, but this means that trees of sizes approaching that of Go cannot use this performance metric.

In this report we present a new Artificial Game Tree framework, Gomba, which solves this problem. Gomba is a framework based around a new game tree generation algorithm we have developed which is able to determine minimax optimality values as it lazily generates nodes of the game tree, rather than needing to calculate them after the tree has been decided. This eliminates the need to run a minimax-equivalent search entirely, which allows for a very significant speed increase in moderately sized trees and makes it feasible to test algorithms against trees that were previously too large to consider at all. We have used this framework to test a number of new variations of UCT, primarily based on methods used in infinitely many armed bandit problems.

2 Artificial Game Trees

We have already mentioned several of the advantages that artificial game trees can provide over a "true" game such as Go or Chess. In particular, objectivity of comparison can be controlled much more precisely and virtually every operation involved in analyzing or mutating the tree can be made much faster than the equivalent operations over "real" game trees.

2.1 Requirements

2.1.1 Lazy State Expansion

We need to be able to simulate trees near the size of a Go game tree, which means it is completely infeasible to ever hope to expand the entire game tree. The tree needs to be definable without an attached full expansion.

2.1.2 Deterministic State Expansion

For the sake of objective comparison between multiple algorithms, however, it is necessary that the tree be persistently generated across subsequent usages. This means that child states need to be determined deterministically and independently of the order in which they are expanded.

2.1.3 Pseudorandom State Expansion

There should not be any clear statistical patterns among different parts of generated trees. The properties of individual game states need to be determined sufficiently pseudorandomly that it is difficult to predict them without actually expanding the states in question. This is important because if it were possible to detect statistical patterns during tree generation, it would be feasible for a well-designed search algorithm to simply detect the relationship between generated tree nodes, rather than needing to actually simulate full games. For the sake of comparative analysis, we need determinism, but for the sake of objective analysis we need the tree to seem non-deterministic to the algorithms in question.

It should be noted that absolute perfection here is not strictly required. It is vital that typical tree search algorithms not be able to abuse simple patterns. For example, it would be unacceptable for the first child node to always be the one to be forced by the forceWinner property of parent nodes. However, we do not necessarily need complete statistical perfection, since we in general are willing to accept the possibility that an algorithm could be constructed specifically to abuse the structure of Gomba trees. Gomba is intended as a tool for researchers, not a perfect representation of non-artificial game trees.

2.1.4 Predetermined State Optimality

In general, the ideal search algorithm for a two-player game tree is one which will always choose an "optimal" action: one which would result in it winning, even if the other player was to also play optimally. Testing whether algorithms will play optimally is therefore a reasonably accurate and totally objective measure of performance, which is excellent for the comparison of search algorithms. Previous artificial game tree frameworks computed such optimality values by running some form of complete minimax search over entire game trees. This is completely accurate, but is also slow to the point of being completely infeasible for larger game trees, such as those as large as Go game trees. One of our initial goals was therefore to be able to create a tree with predetermined knowledge of which actions were optimal for which player, rather than needing to calculate it based on the entire tree. This allows us to use a particularly useful comparative measure without the burden of requiring the tree to be small enough for a complete search.

2.1.5 Fast Action Simulation

One of the advantages of Go is that it is relatively simple to determine the subsequent board state given a particular action. Since this is a procedure which will need to be repeated billions of times over the course of our trials, it is vital that this operation be extremely fast.

2.1.6 Fast Termination Evaluation

One of the major weaknesses of Go as a testing platform is that it is difficult to determine when a game has ended and who has won. We wanted our framework to be able to determine whether a node was terminal and if so, who had won the game, very quickly.

2.1.7 Fast Heuristic Evaluation

Similarly, evaluating a state for a static reward value is exceedingly slow in Go. We would like to be able to emulate the uncertainty involved in such an evaluation without needing to spend large amounts of time considering a state.

2.1.8 Go-Like Action-Reward Distribution

Though our trees will necessarily not be exact replicas of Go game trees, our goal is to test algorithms for eventual use in Go. We would thus like to preserve a reasonably close facsimile of Go's action-reward distributions, that is, how likely moves are to change the balance of the game towards either player's favor. In particular, this notion should not be independent between the moves of a single game. The value of particular move sequences in Go is frequently independent or only slightly dependent on the order in which they are played, as the success of the RAVE algorithm has shown [9]. We would like to preserve this feature in our artificial trees.

Algorithm 1 Gomba Tree Generation

```
SimulateAction(state, action):
    if state.children[action] is not defined:
        state.children[action] := ExpandAction(state, action)
    return state.children[action]

ExpandAction(state, action):
    childState.depth := state.depth + 1
    childState.player := OtherPlayer(state.player)
    childState.prng.seed := GetNthRandom(state.childSeed, action)
    childState.childSeed := childState.prng.nextSeed()
    childState.difficulty := prng.varyDifficulty(state.difficulty)
    if state.forcedWinner = ALL or state.forcedWinner = action:
        childState.winner := state.winner
    else:
        if childState.prng.nextUniform01() < Sigmoid(childState.difficulty):
            childState.winner := PLAYER_MAX
        else:
            childState.winner := PLAYER_MIN
    if childState.winner != childState.player:
        childState.forcedWinner = ALL
    else:
        childState.forcedWinner = childState.prng.nextAction()
    return childState
```

2.2 Realization

Our final tree design was created as a balance of these requirements. We attempted to keep the algorithm as simple as possible for the sake of keeping the time required for node generation low. The final result can be seen in Algorithm 1.

2.3 Properties

The above state structure offers very fast initialization for an appropriately chosen Pseudorandom Number Generator and encapsulates most of the necessary requirements.

2.3.1 Lazy Deterministic Expansion

The tree is expanded only when actually necessary. Until the search algorithm in question attempts to actually simulate an action, the resulting state is not generated. However, when it is generated, the state is determined completely deterministically from its parent's seed value and the index of the action leading to it. All of the properties of a newly generated child are either directly and deterministically dependent on the parent state or dependent on the results of a query to the child state's pseudorandom number generator.

Concrete implementations of the Gomba algorithm have a few requirements which, taken together, ensure that the relevant PRNG query is also deterministically dependent on only the parent state. They are:

- GetNthRandom(startingSeed, n) is a deterministic function,
- The state of a PRNG immediately after its seed is a deterministic function of the seed it is set to,
- The sequence of numbers which a PRNG will draw prng.next...() values from (including nextSeed()) is a deterministic function of the current state of the PRNG.

Together, these guarantee that the childSeed of the new child is the result of running a deterministic function on the childSeed of the parent and the action indexing the child in question. This means that despite the presence of pseudorandom number generators (whose results are only deterministic in the context of their own internal states), the childSeed property of each of a node's children is not dependent on the order in which those children are expanded. The PRNG sequence's seed values–the action index and the parent state–are not mutated after being initially set. Because of this, and because all other properties of the PRNG are determined solely from deterministic functions of these values, generating the child state becomes a deterministic function of the parent state and action index.

2.3.2 Predetermined Minimax Values

The precomputation of the winner and forcedWinner properties of the child state let us maintain a notion of which player is minimax-optimal at any given state (the winner) prior to actually computing all of that state's children. To prove that this is the case, let us formally state the definition of a minimax value of a tree node:

- If a node's parent is minimizing, it is maximizing. If a node's parent is maximizing, it is minimizing.
- If a node is terminal, its minimax value is a measure of how good the state is for each player. Higher is better for the maximizing player, lower is better for the minimizing player. Any value is a valid minimax value in a terminal state.
- A non-terminal maximizing node's minimax value is the maximum of the minimax values of its children.
- A non-terminal minimizing node's minimax value is the minimum of the minimax values of its children.

We introduce a simple theorem based on these rules which will form the basis of our minimax tree generation routine.

Theorem 1. *Let a node m have $n \geq 1$ children $C_m = \{c_{m,1}, ..., c_{m,n}\}$ with respective minimax values $v_{c_{m,1}}, ..., v_{c_{m,n}}$. Let value v_m satisfy the following constraints:*

If m is a maximizing node,

$$\forall c \in C_m,\ v_c \leq v_m \tag{3}$$

If m is a minimizing node,

$$\forall c \in C_m,\ v_c \geq v_m \tag{4}$$

In both cases,

$$\exists c \in C_m,\ v_c = v_m \tag{5}$$

Then, $v_m = x_m$, i.e. the true minimax value of m.

The proof is extremely straightforward and follows from the definitions of minimax value, min, and max. The theorem requires that node m have children, so it falls into one of the final two categories of the above definition. Let its true minimax value be x_m. For convenience, let us also define the set $V_m = \{v_c | c \in C_m\}$.

Proof. In the first case (m is maximizing), by the definition of minimax value $x_m = \max_{v \in V_m} v$. By definition of max, this means that $x_m \in V_m$. We have assumed that $\forall c \in C_m,\ v_c \leq v_m$, which gives us $x_m \leq v_m$. We also know from $\exists c \in C_m,\ v_c = v_m$ that $v_m \in V_m$. However, by definition of max, we know that $\forall v \in V_m,\ v \leq x_m$. These give us $v_m \leq x_m$. Since we already concluded that $x_m \leq v_m$, $x_m = v_m$, that is, v_m is the minimax value of m.

The second case (m is minimizing) is similar. By definition of minimax value, $x_m = \min_{v \in V_m} v$. By definition of min, this means that $x_m \in V_m$. We have assumed that $\forall c \in C_m.\ v_c \geq v_m$, which gives us $x_m \geq v_m$. We also know from $\exists c \in C_m.\ v_c = v_m$ that $v_m \in V_m$. However, by definition of min, we know that $\forall v \in V_m.\ v \geq x_m$. These give us $v_m \geq x_m$. Since we already concluded that $x_m \geq v_m$, $x_m = v_m$, that is, v_m is the minimax value of m. □

We can now prove that our tree states' winner properties satisfy the definition of minimax value at every node in the tree by showing inductively that they satisfy the properties listed in Theorem 1. Our system is modeled on a binary system in which the only available knowledge from the terminal state of a tree is which player won the game. Thus, the only possible options for the winner property of a state, which we treat as our minimax value, are PLAYER_MAX and PLAYER_MIN. We define PLAYER_MAX to be greater than PLAYER_MIN - their exact values are unimportant (we use 1 and 0 in our implementation, but only their ordering affects the proof).

Proof. We use strong induction over trees of particular depth. For the basis step, we state that any tree with a depth of 1 trivially satisfies the requirement because it can contain only one node, and that node will be terminal. By definition, either state of the winner property is a valid minimax value in a terminal node.

For the inductive step, assume that every tree the Gomba algorithm can generate which has a depth of at most d has a valid minimax value as the winner property of every node. Any tree of depth $d + 1$ consists of a single root node which contains a number of children, each of which are trees generated by the Gomba algorithm with a depth at most d. Thus, the inductive hypothesis allows us to assume that every node in any tree of maximal depth $d + 1$ except the root node has a winner value which satisfies the definition of a minimax value. We will now show that the root node of such a tree also has this property. We will spell out the cases in which it is PLAYER_MAX's turn to play at the root node - the cases for PLAYER_MIN are similar and are omitted for the sake of space.

Consider first the case where both the root state's chosen winner and chosen player are PLAYER_MAX. The algorithm then specifies that the root state's forcedWinner property be set to specify a single random child. This means that when that particular child is generated, its winner property will be guaranteed to be the same as the root state's, that is, PLAYER_MAX. This satisfies the existential requirement of Theorem 1. The universal requirement is trivially satisfied since the only possible values for any node's winner property are PLAYER_MIN and PLAYER_MAX, both of which are less than or equal to PLAYER_MAX.

Next, consider the case where the root state's chosen winner is PLAYER_MIN and its chosen player is PLAYER_MAX. The algorithm then specifies that the root state's forcedWinner property be set to specify ALL children. This means that every child the algorithm can generate from this root will be forced to have a winner property of PLAYER_MIN, the root state's winner. Since there is at least one child, this satisfies the existential requirement of Theorem 1. Since no child can have a winner value other than PLAYER_MIN, no child can have a value greater than the root value (PLAYER_MIN), which is equivalent to the universal requirement of Theorem 1.

Thus, both the root node and all of its child nodes have winner states which are valid minimax values. This completes the inductive step. By strong mathematical induction, this implies that for any tree of arbitrary finite depth generated by the Gomba algorithm above, every node's winner property will be a valid minimax value. □

2.3.3 Speed

The Gomba generation algorithm is relatively simple, which allows it to be quite fast. The actual ExpandNode function consists of only a few if statements, a few assignments, and a single arithmetic operation. The bulk of the work is in its subroutines, each of which can also be made to execute quickly:

- OtherPlayer() is a single trivial if statement
- Sigmoid() is a simple function which can be computed very quickly on modern processor architectures where exp() is a single instruction operation
- prng.nextAction(), prng.nextUniform01(), and prng.nextSeed() are all dependent on the particular implementing pseudorandom number generator, but for a sufficiently simple one (our implementation uses a Linear Congruential Generator for the sake of speed) can require very few operations for these uniform sampling routines.

- prng.varyDifficulty() would typically not be a uniform routine, but can still be very fast. In our implementation this is a polynomial of degree six which approximates a normal distribution.

- GetNthRandom() is again a function of how fast the chosen PRNG is. We chose a Linear Congruential Generator in part because the nth random number to be generated from a particular seed in an LCG can be computed in constant time, so long as a single simple lookup table of at least length n has been precomputed for the LCG class.

- There is a final "hidden" cost for allocating memory for the new childState. However, tree generation of this sort lends itself well to memory pooling, which means that this cost can largely be eliminated simply by reserving memory in large blocks instead of on a per-state basis.

We expand on the specific implementation in the Gomba framework in the next section.

2.4 Weaknesses

2.4.1 Weak Randomness

The generated trees look relatively random to the human eye, but do have a few statistical properties which brings the actual randomness of the tree into question. The primary problem is that in order to deterministically generate children regardless of order of generation, we seed the pseudorandom number generator according to an operation based on the seed value from the parent. In theory, this can cause several problems:

- Computing a single path down the tree results in a process where the pseudorandom number generator repeatedly and consistently seeds itself with its own generated value. This is very atypical operation for most PRNGs, which is problematic because while most PRNGs are designed to have useful, random statistical properties when run in sequence, these properties do not necessarily hold when the PRNG is reseeded during this sequence.

- Seeding a random number generator of most useful complexities is very slow. Except for extremely simple PRNGs such as Linear Congruential Generators and Linear Feedback Shift Registers, both of which are largely outdated and outclassed by more modern generators, seeding a PRNG takes a considerable amount of time.
- Since the number of possible seeds is much more tightly limited than the cycle length of a modern PRNG, the number of potential children is significantly limited.

To alleviate the first two issues, we have implemented the pseudorandom number generator used by the Gomba tree generation system as a Linear Congruential Generator. This means that it can be seeded with its own generated values in the course of normal operation. It also makes pseudorandom number generation extremely fast [16]. It is important to note that this form of generator has been rendered largely irrelevant for most modern uses which require strict statistical randomness. For the purposes of our framework, we judged the benefits of its structure and speed to be more valuable than the statistical properties allowed by more advanced generators. Experiments on trees generated by the Gomba framework have shown that the tree nodes are sufficiently well distributed for our purposes - for example, our tests have shown that in repeated trials over generated trees of size 2^{14} no nodes with overlapping seed values are generated.

2.4.2 Random Action-Reward Distribution

The current Gomba system has no means of ensuring a reasonably consistent value for actions across parent states. In practice, this means that the generated game trees lack a property which is relatively important to Go; notably, the UCT-RAVE algorithm which we know is a significant improvement over standard UCT in Go provides no measurable improvement in a Gomba game tree. The consistency of the distribution of difficulty and winning values is very strongly correlated to the game one is playing, so making Gomba's game trees use a distribution system closer to that of Go would be a significant improvement to the accuracy of the system in evaluating algorithms for performance in Go.

2.5 Conclusions

For sufficiently complicated game trees, the Gomba tree generation algorithm is orders of magnitude faster than both Go itself and previous artificial game tree frameworks. Although it is not a perfect approximation of Go itself, its speed of evaluation makes it a useful tool for the evaluation of algorithms which are applicable to games with large state trees.

3 Gomba

A primary contribution of our project is the Gomba search framework, which is based on the previously explained tree generation algorithm. The Gomba framework has been designed primarily with speed and simplicity in mind. Our goal was to create a framework which we and future researchers can use to quickly implement and benchmark a variety of search algorithms on artificial game trees. The core Gomba tree generation algorithm and the simple but extensible framework for search algorithm implementations allows for speed of both implementation and evaluation of new algorithms, particularly those based on UCT-style searches.

3.1 Framework Requirements

When we set out to create this new framework, we had a specific set of requirements in mind that would make testing new algorithms efficient in both programmer and execution time. Though other frameworks exist which incorporate several or even most of these requirements, the Gomba framework is the first to attempt to combine all of them to the best of our current knowledge.

3.1.1 Modularity

One of the primary goals of the framework was to be able to quickly implement any new search algorithm that one might wish to test. To achieve this property, we strived to ensure that the Gomba framework struck an acceptable balance between being general enough to allow for any search methodology and containing enough base material that no implementation would take too much effort to write. The framework itself provides abstractions to allow for storing data in game tree nodes, gathering statistics about the iterative runtime performance of algorithms, the separation of internal model improvement and evaluation based on that model, and a variety of basic Monte-Carlo and Minimax base algorithms which can be expanded upon. The actual implementation of these properties is largely hidden from the algorithms - though they can typically access whatever attributes of these routines that they need, should they need them, in most cases the simple interfaces that the framework provides are enough. Because the framework itself takes care of nearly all work that is not algorithm-specific, it is extremely fast to implement new algorithms to test against, particularly when they are relatively minor modifications of existing ones as in the case of most variants of UCT.

3.1.2 Simplicity

The primary motivation for making Gomba from scratch rather than building the tree generation algorithm into an existing testing framework was that existing testing frameworks are almost universally too complex for our needs. Our goal was to be able to allow future researchers to move from an idea for a new search algorithm variant to a working implementation with a minimum of effort. We have striven to ensure that the framework is sufficiently simple, readable, and well-documented that a future contributor can determine how to test a new algorithm without needing to understand every detail of the framework as a whole, and that should some detail need to be changed, the framework is simple enough and sufficiently easily modifiable that this would not be a problem.

3.1.3 Rapid Node Generation and Access

Nodes must be generated quickly and effectively to make sure tests can be run in a reasonable amount of time in Gomba. Lazy tree generation means that when a new node is visited it will need to be generated and possibly expanded – a trade-off in order to circumvent the issues surrounding eager tree generation. As was stated before, game trees approaching the size of Go are infeasible to generate eagerly and store in memory. Because of this problem it is necessary to use lazy algorithms to generate only as many nodes as are needed, so that searching on large trees becomes feasible.

However, if nodes are simply generated and then erased, unnecessary amounts of time will be spent re-computing nodes as they are needed. For this reason nodes must be cached upon generation so that exactly as many nodes as are needed exist in memory.

3.1.4 Scalability to Go-Sized Trees

As discussed previously, one of the most important requirements we placed on Gomba is that it should be able to perform well on trees that approximate the size and behavior of Go game trees. Not only must generating the tree be efficient, but memory must be managed in a way that as trees get larger the performance overhead does not increase to unsustainable levels. Much of Gomba is made to scale well to very large trees–a helpful result of generating trees lazily and caching nodes in memory upon generation.

3.1.5 Simple Parametrization

Though our research is primarily focused on Go-like trees, it is important that the framework as a whole be sufficiently general to emulate search algorithms usable against any binary game tree. Parameters such as game branching factor, length, difficulty, and heuristic effectiveness must be easily modifiable at runtime and should be able to take on a wide range of potential values without adversely affecting the speed or statistical properties of the framework.

3.1.6 Reliable, Deterministic Results

No testing framework would be worthwhile without reproducible results that are actually useful to any users of the test data. In the case of Gomba, we expect that the most common use case will be the comparison of several similar search algorithms. Because of this, a significant emphasis has been placed on making sure that with the same testing conditions, algorithms will perform deterministically (though pseudorandomly) across test runs on the Gomba framework. We provide a means of manually setting all relevant random seeds for reproducibility, and we ensure that there is a simple means of running a variety of algorithms on the same set of trees. This helps ensure that the algorithms can be compared against one another fairly.

3.2 Implementation

The overall design of the platform was based on relatively standard object-oriented principles and is relatively uninteresting. There were, however, several interesting areas in which special care was needed to maintain runtime efficiency.

3.2.1 Node Structure

The previous section has already enumerated the required properties of the tree itself in some detail. However, a few implementation details remain to be specified which do not drastically affect the properties of the tree but do affect speed and memory usage. In particular, the list of child pointers maintained by each node in the tree is the largest section memory-wise and we gave considerable thought to how best to implement it.

We settled on a simple array of child pointers. Though this is a relatively poor option from the standpoint of memory usage, since for the vast majority of nodes only one child (or at most few children) will be expanded, our testing revealed that our computations were bounded much more significantly by time than memory constraints. We found that any space benefit gained by using a more intelligent pointer allocation scheme was outweighed by the cost in time, since the simulation of actions in the tree was such a frequently requested operation in virtually every search algorithm we tested.

This was also one of the considerations that led us to store search-specific data in the game tree itself, rather than requiring that search algorithms maintain their own trees of data. Though doing so would have allowed us to prune the game tree more aggressively, it would have required not only a massive amount of extra effort in the generation of the search algorithms themselves to maintain a new tree structure but a massive amount of repeated memory, since in both trees most space would be taken up by the child reference mechanism.

3.2.2 Memory Management

When generating nodes at the rates which we desire of Gomba–on the order of 10^7 to 10^8 nodes per second–it is important to manage memory in a way that minimizes allocation and deallocation overhead. Frequent system calls requesting and freeing space for nodes can lead to significant time that is ultimately wasted on memory management. For this reason, Gomba uses object pooling–allocation and storage of objects in large "pools" of memory–to store node data, requesting space for several thousand nodes at a time. Gomba's system of caching tree nodes for the lifetime of a search algorithm run means that nodes only need to be deallocated in large batches at the end of algorithms' runs, which can be done in the same efficient blocks of thousands of nodes at a time.

This strategy is not without weaknesses, however. Particularly, for algorithms which frequently generate nodes which will only be visited once, it is possible for a Gomba tree cache to take up enough memory to bring the system to a crawl or, worse, crash the simulation entirely. Gomba does not currently have any means of pruning its own game trees - though the operation would be safe from the standpoint of maintaining the tree structure, since children are always generated deterministically from parents, the framework cannot be sure whether the data a search algorithm may have stored in the tree for its own use is limited or unimportant enough to discard. A pruning operation would also require more careful allocation of nodes, since efficiency would dictate requiring that entire blocks in the memory pool were freed at a time and currently pools are filled on a first-come-first-serve basis with no concern for the structure of the tree itself.

3.2.3 Random Number Generation

Early on in the development of Gomba we found that the overwhelming majority of computation time was being taken up in the pseudorandom number generation engine. Specifically, the tree generation routine's need to seed a generator at every node expansion was taking an inordinate amount of time. There were two primary options available to us to solve this problem: we could either remove the seeding operations, or we could make seeding fast. Both have disadvantages.

In the first case, we would lose the property that the tree is deterministic regardless of node expansion order. This would mean that different search algorithms running against a tree based on the same initial parameters would not actually run on exactly the same game tree. We considered allowing this, but in the end we decided that it was unacceptable to not be able to run different search algorithms on the same tree; it would introduce an unacceptable amount of disparity into comparative analysis of the algorithms. Even though the results would in theory be comparable after a large number of trials, we felt that it would be better to avoid the issue entirely.

In the second case, which we eventually chose, the disadvantage is that pseudorandom number generators with internal state simple enough to be easily regenerated from a small seed tend to also be simple enough that they do not have some of the nice statistical properties that more modern and complex generators do. We use a Linear Congruential Generator to create trees in our framework, and there were several cases where testing revealed generation problems in which easily deducible parts of the tree were either copies of each other or in simple arithmetic sequence from each other. Though our tests have shown that we have eliminated obvious incongruities caused by this issue, the fact remains that Gomba's trees are generated with a relatively weak number generation scheme. Further research into the effects of this decision and other solutions which could alleviate any of these potential issues is a strong potential for future work.

3.2.4 Monte-Carlo Simulation

In testing several Monte-Carlo based search variants, we found that the overwhelming majority of execution time was spent evaluating nodes as part of random Monte-Carlo searches which would never need to be evaluated a second time. We were able to achieve an evaluation speed improvement of a factor approximately proportional to maximum tree depth by creating and using a method of rapidly simulating a one-off Monte-Carlo trial without needing to actually evaluate, create, or cache the intermediary nodes of such a simulation path.

The implementation of this system is made relatively straightforward by our tree structure. The basic process is to simply take the difficulty property of the node which will serve as the starting point of the simulation, and use that difficulty value to determine a winner value in a similar manner as is used for the randomly-decided children of that node. The basic idea that explains why this is acceptably accurate is based around the distribution of this difficulty value. Because the difficulty of a node's children is a normal distribution centered at that node's difficulty, its children will have on average the same difficulty. By induction over the same argument, so will all of its descendants. This means that, if the Monte-Carlo search would have ended at a child with a randomly determined winner value, the probability of that child winning for a particular player is the same as the probability of our simulation process deciding on a win for that player. The probability of a child being decided non-randomly in favor of a particular player follows approximately the same distribution as well.

We will now formalize why these properties hold in an idealized environment; though we recognize that the properties we are about to assume are not strictly true in practical tests, they are generally "close enough" that the results from the idealized theorem hold empirically. In particular, we recognize that the constraints the theorem imposes on branching factor and terminal node depth are realistically impossible and that descendant difficulty is not equal to but merely centrally distributed about root node difficulty. That said, the idealized theorem closely mirrors our actual test results, and we offer it here as a justification to our use of the formula it prescribes in Gomba's fast simulation policy.

Theorem 2. *Let there be some tree rooted at node r generated by the earlier reference Gomba tree generation algorithm. Assume that the difficulty property of r and the difficulty properties of all of its descendants are all d. Let $x = Sigmoid\,(d)$ be the probability that a randomly generated winner property for any such descendant be winning for PLAYER_MAX, as per the algorithm. Let b be the number of children any non-terminal node has. Then, assuming that terminal nodes have equal probability of being found at depths which are even or odd, the probability of a random Monte-Carlo simulation starting from node r ending in a terminal node in which PLAYER_MAX is the winner approaches*

$$\frac{x}{2}\left(\frac{x\,(b-1)+(b+1)}{x^2\,(b-1)-x\,(b-1)+b}\right) \tag{6}$$

as the minimum depth of terminal nodes approaches ∞.

Proof. Let the value p_n be defined as the probability that a randomly chosen node from all nodes at depth n in the tree rooted at r have a winner property defining PLAYER_MAX as the winner. By convention we will say that the root node has depth 1. Also by convention and without loss of generality we will assume that it is PLAYER_MIN's turn at the root node r (and every other node at an odd depth).

The root node is the only node at depth 1 and is not forced to take a value by its parent, and we have explicitly defined x as the probability that PLAYER_MAX will win for a randomly-chosen (that is, non-forced) node. Thus, by definition of x, we have $p_1 = x$.

We now define a recurrence relation for p_n for the cases where $n > 1$. Let us consider a random node N at depth $n > 1$ which has parent node P at depth $n-1$. There are two potential cases here, depending on the parity of n.

If n is even, then it it was PLAYER_MIN's turn to play at node P. This means that if PLAYER_MAX were to be P's winner value, all of P's children would be forced to a winner value (of PLAYER_MAX). This means that the conditional probability of PLAYER_MAX winning for node N where P's winner value is PLAYER_MAX is simply 1, since N would be forced to make PLAYER_MAX winning.

If PLAYER_MIN were to be P's winner value only one of P's children would be forced to a winner value (of PLAYER_MIN). This means that the conditional probability of PLAYER_MAX winning for node N where P's winner value is known to be PLAYER_MIN is

$$P\left(N.winner = PLAYER\_MAX \;|P.winner = PLAYER\_MIN\right) = 0\left(\frac{1}{b}\right) + x\left(\frac{b-1}{b}\right) \tag{7}$$

That is, there is a $\frac{1}{b}$ probability that N will be the child forced to use PLAYER_MIN as a winner and a $\frac{b-1}{b}$ probability that N's winner will be chosen randomly (thus, with probability x).

The probability that PLAYER_MAX would win for node P is by definition p_{n-1}. Thus, the probability that PLAYER_MAX would win for any of P's children (node N included) is

$$p_n = p_{n-1} + x\left(\frac{b-1}{b}\right)(1 - p_{n-1}) \tag{8}$$

In the second case, where n is odd, it was PLAYER_MAX's turn to play at node P. This means that if PLAYER_MIN were to be P's winner value, all of P's children would be forced to a winner value (of PLAYER_MIN) and that the conditional probability of PLAYER_MAX winning for node N in this case is simply 0. If PLAYER_MAX were to be P's winner value only one of P's children would be forced (to PLAYER_MAX), which gives the conditional probability

$$P\left(N.winner = PLAYER\_MAX \;|P.winner = PLAYER\_MAX\right) = 1\left(\frac{1}{b}\right) + x\left(\frac{b-1}{b}\right) \tag{9}$$

Thus, the final probability that PLAYER_MAX will win for any of P's children (node N included) in the case where n is odd is

$$p_n = p_{n-1}\left(\frac{1}{b} + x\left(\frac{b-1}{b}\right)\right) \tag{10}$$

This gives us the final recurrence relation

$$p_n = \begin{cases} p_{n-1} + x\left(\frac{b-1}{b}\right)\left(1 - p_{n-1}\right) & n \text{ is even} \\ p_{n-1}\left(\frac{1}{b} + x\left(\frac{b-1}{b}\right)\right) & n \text{ is odd} \end{cases} \tag{11}$$

For simplicity, we define $a = x\left(\frac{b-1}{b}\right)$ and rearrange the above to form

$$p_n = \begin{cases} a + p_{n-1}\left(1 - a\right) & n \text{ is even} \\ p_{n-1}\left(\frac{1}{b} + a\right) & n \text{ is odd} \end{cases} \tag{12}$$

From this, we can derive a more useful two-step recurrence relation. First consider the case where n is even. Then we have

$$p_n = a + p_{n-1}\left(1 - a\right)$$

$$p_{n-1} = p_{n-2}\left(\frac{1}{b} + a\right)$$

$$p_n = a + p_{n-2}\left(1-a\right)\left(\frac{1}{b}+a\right) \tag{13}$$

When n is odd, we similarly can derive

$$p_n = p_{n-1}\left(\frac{1}{b}+a\right)$$

$$p_{n-1} = a + p_{n-2}\left(1-a\right)$$

$$p_n = \left(\frac{1}{b}+a\right)\left(a + p_{n-2}\left(1-a\right)\right)$$

$$p_n = \left(\frac{a}{b}+a^2\right) + p_{n-2}\left(1-a\right)\left(\frac{1}{b}+a\right) \tag{14}$$

Giving a final relation of

$$p_n = \begin{cases} a + p_{n-2}\left(1-a\right)\left(\frac{1}{b}+a\right) & n \text{ is even} \\ \left(\frac{a}{b}+a^2\right) + p_{n-2}\left(1-a\right)\left(\frac{1}{b}+a\right) & n \text{ is odd} \end{cases} \tag{15}$$

Both of these form geometric sequences (they conveniently have the same multiplicative factor). This factor, $(1-a)\left(\frac{1}{b}+a\right)$, has zeroes at $x = \frac{-1}{b-1}$ and $x = \frac{b}{b-1}$ and its maximum is $\left(\frac{b+1}{2b}\right)^2$ at $x = \frac{1}{2}$. b's range is $[2, \infty)$, which means this maximum value's range is $(\frac{1}{4}, \frac{9}{16}]$. Thus, for any potential value of b, the widest possible range of values for the factor in the domain $[\frac{-1}{b-1}, \frac{b}{b-1}]$ is $[0, \frac{9}{16})$. Recall that x is a probability, which means its domain $[0,1]$ is a subset of $[\frac{-1}{b-1}, \frac{b}{b-1}]$. Thus, the factor's potential range is a subset of $[0, \frac{9}{16})$, which means it is also a subset of $(-1,1)$, which means that these geometric sequences will both converge.

In the first case, the convergence value is

$$\begin{aligned}
p_n &= a + p_n (1-a)\left(\frac{1}{b} + a\right) \\
&= \frac{a}{1-(1-a)\left(\frac{1}{b}+a\right)} \\
&= \frac{a}{1-\left(\frac{1}{b}+\left(1-\frac{1}{b}\right)a - a^2\right)} \\
&= \frac{a}{a^2 - \left(\frac{b-1}{b}\right)a + \left(\frac{b-1}{b}\right)} \\
&= \frac{\left(\frac{b-1}{b}\right)x}{\left(\frac{b-1}{b}\right)^2 x^2 - \left(\frac{b-1}{b}\right)^2 x + \left(\frac{b-1}{b}\right)} \\
&= \frac{x}{ax - a + 1}
\end{aligned}$$

In the second case, the value is:

$$\begin{aligned}
p_n &= \left(\frac{a}{b} + a^2\right) + p_n (1-a)\left(\frac{1}{b} + a\right) \\
&= \frac{\frac{a}{b}+a^2}{1-(1-a)\left(\frac{1}{b}+a\right)} \\
&= \frac{\frac{a}{b}+a^2}{1-\left(\frac{1}{b}+\left(1-\frac{1}{b}\right)a - a^2\right)} \\
&= \frac{a\left(\frac{1}{b}+a\right)}{a^2 - \left(\frac{b-1}{b}\right)a + \left(\frac{b-1}{b}\right)} \\
&= \frac{\left(\frac{b-1}{b}\right)\left(\frac{1}{b}+a\right)x}{\left(\frac{b-1}{b}\right)^2 x^2 - \left(\frac{b-1}{b}\right)^2 x + \left(\frac{b-1}{b}\right)} \\
&= \frac{x\left(\frac{1}{b}+a\right)}{ax - a + 1}
\end{aligned}$$

Our initial assumptions that terminal nodes only occur as depth approaches ∞ and that they are equally likely to occur in even or odd nodes together imply that the final probability that PLAYER_MAX will win at a terminal node is the average of these convergence values:

$$\begin{aligned} p_t &= \frac{1}{2}\left(\frac{x}{ax-a+1}\right)+\frac{1}{2}\left(\frac{x\left(\frac{1}{b}+a\right)}{ax-a+1}\right) \\ &= \frac{x\left(a+\frac{b+1}{b}\right)}{2\left(ax-a+1\right)} \\ &= \frac{x}{2}\left(\frac{\left(x\left(\frac{b-1}{b}\right)+\left(\frac{b+1}{b}\right)\right)}{\left(x^2\left(\frac{b-1}{b}\right)-x\left(\frac{b-1}{b}\right)+1\right)}\right) \\ &= \frac{x}{2}\left(\frac{x\left(b-1\right)+\left(b+1\right)}{x^2\left(b-1\right)-x\left(b-1\right)+b}\right) \end{aligned}$$

□

3.3 Conclusions

As implemented, Gomba provides a fast, robust, and highly extensible means of both lazily generating game trees and designing search algorithms for testing against these artificial trees. The tree generation framework is sufficiently well-optimized that in our profiling tests against actual algorithms, less than 20% of computation time is spent in tree access and generation (as opposed to in the execution of the search algorithms themselves). The speed of execution is thus bounded primarily by the search algorithms themselves, and not by any measurement over the game tree as is the case in the game of Go. This is an extremely significant speed improvement over testing algorithms against Go itself and an extremely significant memory improvement over previous similar artificial game tree frameworks.

The testing framework still leaves room for improvement, though. Although it is bounded primarily by search speed, we still feel that the framework could provide better tools for helping search algorithms manage their resource usage. At present Gomba's tree generation algorithm is able to produce on the order of 10^7 nodes per second on a typical modern desktop machine; this is acceptable in comparison to true computer Go playing, but approximately an order of magnitude below our initial hopes for the framework.

There are also a question of whether the random number generation algorithms we use bring the statistical validity of the generated trees into question. Our analysis of the generated trees does not show any obvious signs of correlation between nearby nodes, and our analysis of smaller generated trees do not seem to show obvious statistical correlation for any related sets of nodes. We do not want to suggest that our searches have been exhaustive or that there are no problems with the generation method, since we recognize in particular the weaknesses associated with our choice of pseudorandom number generator. We do not, however, believe that these potential statistical pitfalls invalidate the results the framework generates, particularly in the face of the empirical evidence of experiments based on algorithms with known performance metrics in Go.

4 Experiments

The value of the Gomba framework is that it allows for the efficient testing and comparison of a variety of game tree search algorithms. In this section we first present the results of searching Gomba-generated game trees with a number of search algorithms whose performance has already been tested in Go and on other artificial trees by previous researchers. Once satisfied that the Gomba trees allow these control algorithms to perform as expected, we tested several variants on these algorithms which have not been so thoroughly tested as a measure of how efficiently new methods could be benchmarked with our framework.

Our experiments are primarily concerned with two metrics. The first is the algorithms' performance in maximizing optimal win rate, that is, how quickly it can consistently choose moves which are minimax-optimal. This is a useful metric in smaller trees, but in practice, no algorithm can really hope to consistently choose completely optimal moves in larger trees. The second metric is the average difficulty of the moves that the algorithm chooses. This number coincides with the difficulty property of Gomba tree nodes and represents a measure of how likely each player is to win when a randomly chosen path is chosen starting from the node in question. We would like this number to be low, sometimes even at the cost of a worse optimal win rate, since it is often the case in adversarial search that making it harder for your opponent to find optimal moves is as valuable as finding optimal moves yourself.

4.1 Previously Tested Algorithms

These are algorithms which have already been extensively tested, either in computer Go or in other games. We first present a short description of each, noting in particular any properties which made them interesting to our tests. We then provide a comparative analysis of their performance on trees of varying parameters and compare this analysis to the results we expected based on previously researched performance metrics.

4.1.1 Random Search

As in many other experiments regarding search, random search is used in this case to provide a lower bound of performance for all other algorithms being tested. Its policy is simple: at any time, pick a random node and use it. Random search is not designed to be effective, but rather to set an acceptability threshold for each algorithm. After all, if an algorithm should have worse performance and require more resources than simply choosing random nodes, using it would simply be a waste.

4.1.2 Minimax Search

The minimax algorithm consists of simply traversing a tree, alternately determining a node's value to be either the minimum or maximum of the values of its children. Minimax search is both sound and complete: its results are guaranteed to be optimal, and it is guaranteed to be able to generate an optimal result. However, the time required to run an exhaustive minimax search is on the order of $O\left(b^d\right)$, which renders it useless for most commonly sized tree searches. A variety of optimizations exist which can improve this bound (for example, Alpha-Beta pruning [14] and Negascout [18]), but in general only down to the order of $O\left(b^{\frac{d}{2}}\right)$. This is still prohibitively large for most practical applications.

However, despite exhaustive search's inadequacy in practical trees, it remains useful as a testing metric. The Gomba framework contains a simple minimax implementation which we have used in testing to ensure that tree nodes' winner properties accurately reflect minimax values.

4.1.3 Random Monte-Carlo Search

At its core, the Rollout-Based Monte-Carlo family of search strategies consist of searches which work by repeatedly sampling directed random move sequences to build knowledge of which moves are likely to perform well. Any such algorithm whose selection policy guarantees that all moves will always have some positive chance of being chosen (that is, any which never entirely stops exploring) will eventually converge to an optimal solution, since the decision policy is equivalent to minimax once every possible path has been sampled. However, most game trees for practical applications are far too large for such an exhaustive search to be feasible. Instead, most Monte-Carlo methods currently in use are designed to find a move which is "good enough", rather than strictly searching for one which is optimal.

Random Monte-Carlo search is an extremely simple variant of Rollout-Based Monte-Carlo search which simply specifies an arm selection policy during the search of "choose a completely random arm at all times during simulated playout". The arm that is chosen to be played is the arm with the highest average reward. As explained above, this policy does eventually converge to a correct answer, but it will typically do so much more slowly than is useful. It is included in our tests primarily as a baseline test against which more clever Monte-Carlo algorithms can be compared.

4.1.4 UCT

UCT (Upper Confidence bound for Trees) is a rollout-based Monte-Carlo search method based on the UCB1 [2] multi-armed bandit strategy [15]. It is a very standard variant in that the only difference between it and random Monte-Carlo search is the arm selection policy during episodic playout. The selection policy works by treating each episodic sampling decision as a separate multi-armed bandit problem and applying the UCB1 algorithm to each of them.

UCB1 is a simple method of balancing exploration of new arms and exploitation of previously successful arms which guarantees a worst-case logarithmic regret growth rate under the assumptions that that the rewards for each arm are independent and bounded. Specifically, at each iteration it chooses which arm to play next according to the formula,

$$argmax_j\{\overline{x}_j + C\sqrt{\frac{log(N)}{n_j}}\}, \tag{16}$$

where $\overline{x}_j$ is the empirical average reward of arm j, N is the total number of plays so far, n_j is the number of plays of arm j so far, and C is some exploration constant (in most practical Go engines, $C \approx 0.7$).

In computer Go, UCT is well-tested and extremely effective as a baseline strategy. It is the foundation of most current competitive Go-playing programs, including MoGo [19], Fuego [7], and CrazyStone [6]. Our experimental results on UCT in the Gomba framework (Figure 1) closely mirror the results from earlier artificial tree frameworks [15] for smaller trees. In larger trees, we found that UCT was unable to converge to optimal moves within a reasonable amount of time but that it was effective at minimizing the difficulty of chosen moves (Figure 2), especially at larger branching factors. This is consistent with performance in computer Go, which lends strength to the feasibility of using artificial game trees to estimate the performance of search algorithms in computer Go.

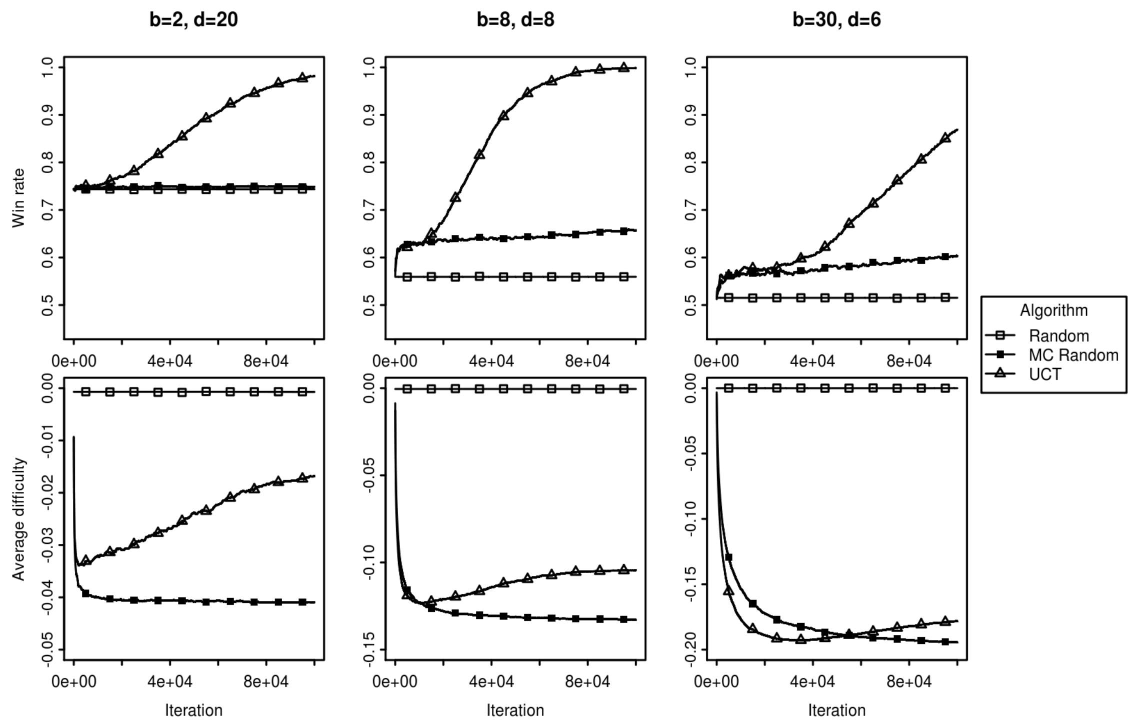

Figure 1: UCT Performance in Small Gomba Trees

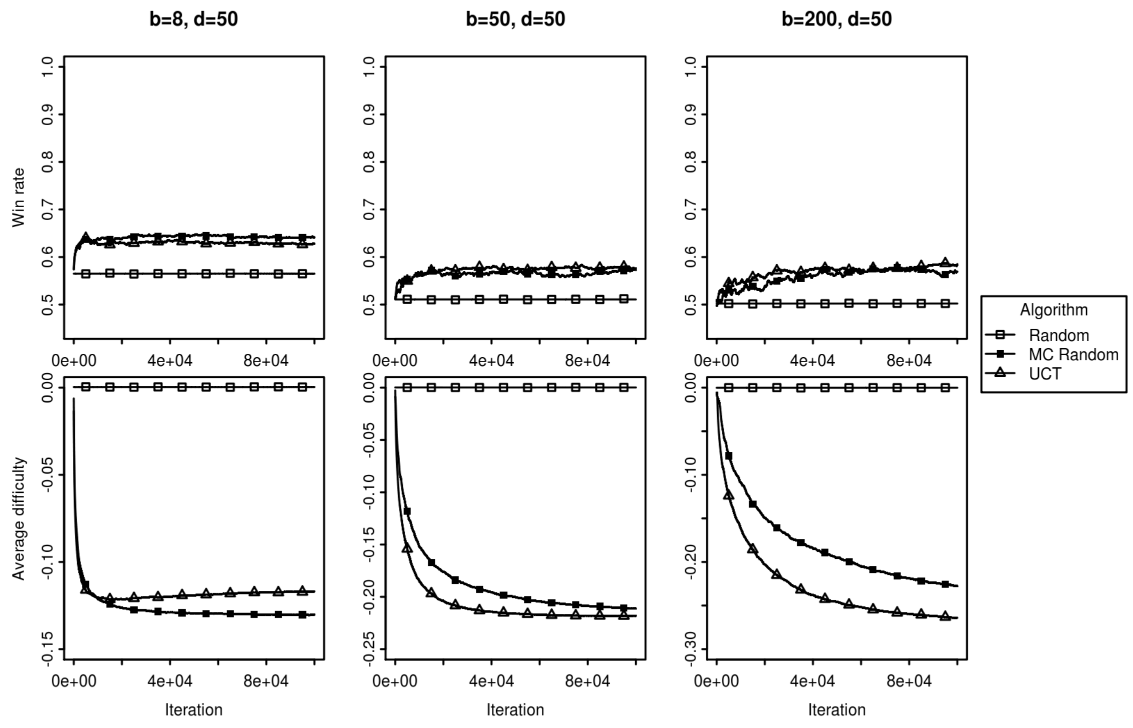

Figure 2: UCT Performance in Large Gomba Trees

4.1.5 UCT-Tuned and UCT-V

UCT-Tuned and UCT-V (UCT with Variance) are slight modifications of UCT which incorporate the notion of variance. They both attempt to favor exploitation over exploration more heavily when the observed variance of an arm is low, and exploration over exploitation when the observed variance is high. The implementation is exactly the same as in UCT, but with slightly modified arm selection formulas (UCB-Tuned [2] and UCB-V [22], respectively).

UCB-Tuned first computes an upper confidence bound for the variance of arm j:

$$V_j'(s) = (\frac{1}{s}\sum_{\tau=1}^{s} X_{j,\tau}^2) - \overline{X}_{j,s}^2 + \sqrt{\frac{2\ln t}{s}} \tag{17}$$

It then applies that variance bound to the overall upper confidence bound calculation:

$$argmax_j\{\overline{x}_j + C\sqrt{\frac{\ln N}{n_j}\min\{\frac{1}{4}, V_j'(n_j)\}}\} \tag{18}$$

The $\frac{1}{4}$ factor exists to ensure a minimum level of exploration even in cases of very low apparent variance.

The UCB-V formula works similarly, but uses a measure of actual empirical variance $V_k(s)$ rather than the upper bound for variance above $(V_k'(s))$ in the valuation formula,

$$argmax_j\{\overline{x}_j + \sqrt{\frac{2V_j(n_j)\varepsilon_N}{n_j}} + \frac{3\varepsilon_N}{n_j}\} \tag{19}$$

Here ε_t represents some non-decreasing sequence in t, such as $\ln t$. Both methods empirically give similar results, and both are fairly widely used. Though performance bounds significantly better than standard UCT have not been proved, they both empirically tend to outperform standard UCT in most computer Go simulations.

Gomba simulations of these variants tended to be slightly less effective than we had anticipated based on previous works (see Figure 3). We believe that this may be caused by a combination of two structural properties of Gomba trees. The first is that the true reward value of a Gomba tree state is a binary value. This effectively means that the winner properties of a node's children take on a Bernoulli distribution, which in effect also means that the variance is completely decided by the difficulty of the parent node in question. Since the vast majority of nodes will have difficulties approximately the same as the root node's, this means that the variances of the children of most nodes in a tree will be very close to one another, which would reduces the beneficial impact of variance-detecting UCT variants.

The second potential pitfall is that the difficulty values of various nodes' children are all determined by the same normal distribution – that is, the variance in the distribution of difficulty values (which a Monte-Carlo search can potentially work to minimize in addition to win rate) is constant. This again reduces the beneficial effects of variance-conscious searches.

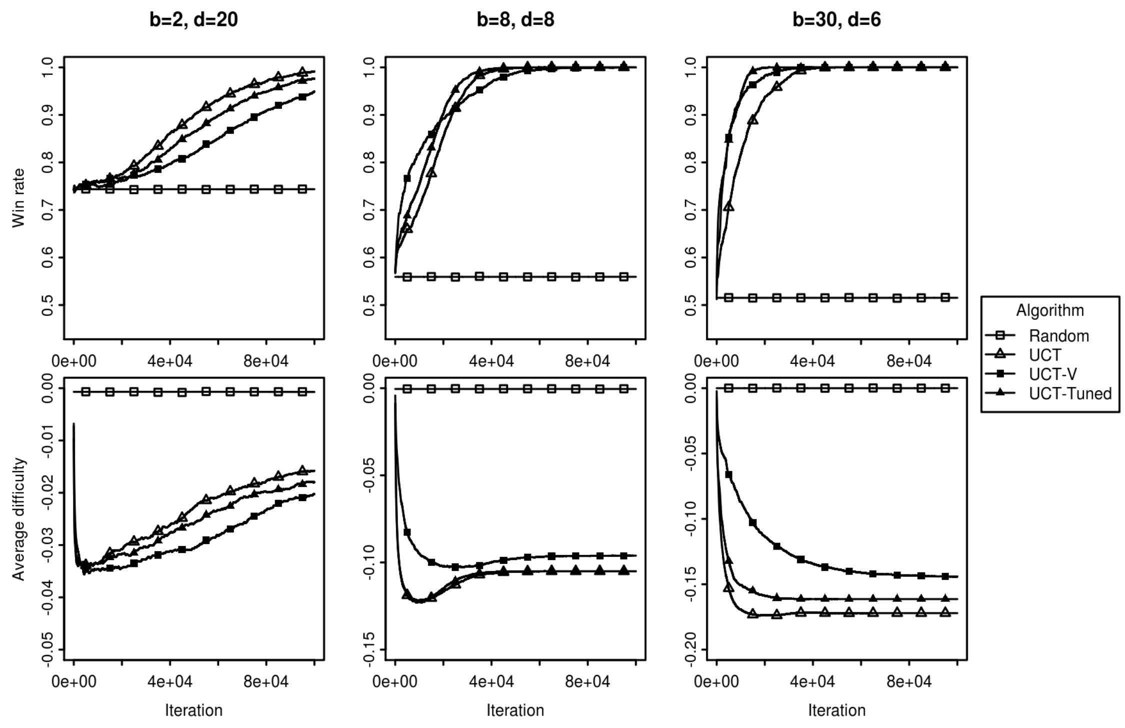

Figure 3: Variance-Based UCT Performance in Small Gomba Trees

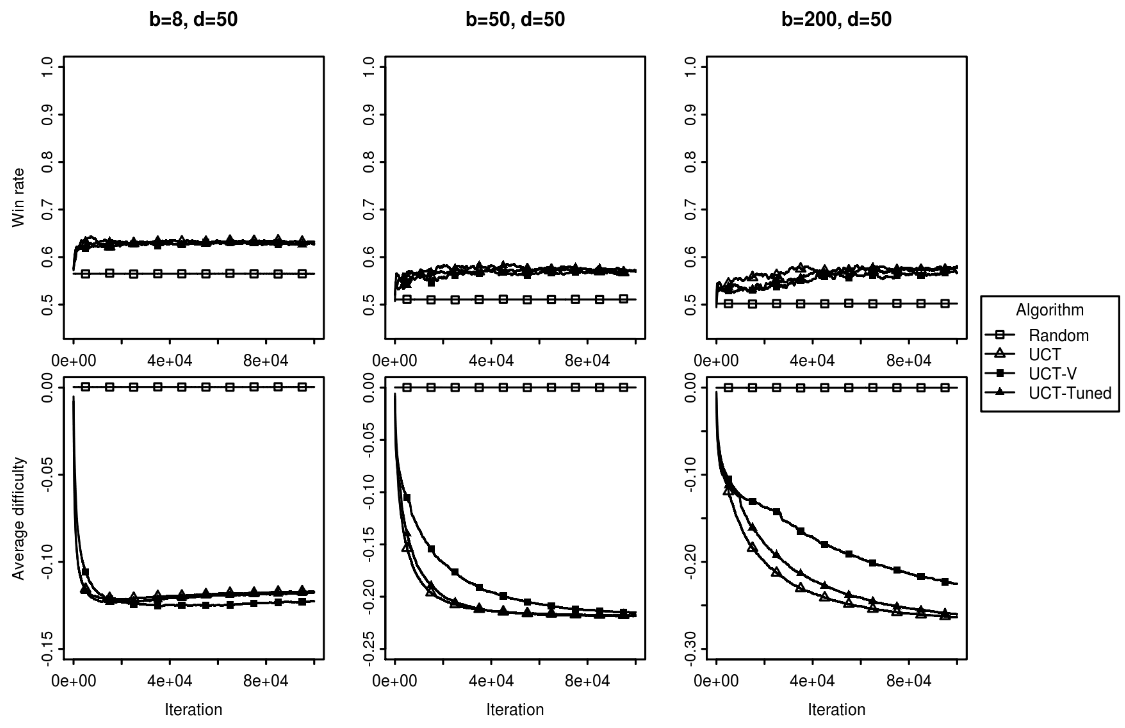

Figure 4: Variance-Based UCT Performance in Large Gomba Trees

4.1.6 UCT-FPU

UCT-FPU (UCT with First Play Urgency) [12] is a minor modification to the UCT algorithm which simply causes any unvisited node to be treated as having a fixed finite upper bound, rather than being treated as having an infinite upper bound as in standard UCT. This upper bound, expressed as a constant, can be used to shift the balance between exploration and exploitation in the tree. In particular, by setting the FPU constant lower at deeper levels of the search tree, the UCT algorithm can begin trying to exploit known acceptable paths without any exploration whatsoever of many other paths. Near the root of the search tree this would run the risk of missing potentially valuable nodes, but as the search tree becomes deeper it becomes not only less valuable but computationally infeasible to consider every node at a particular level before moving on. This method or something similar to it is thus particularly essential to UCT implementations in computer Go because of the sheer size of the search tree; UCT's tendency to heavily favor exploration over exploitation early on works well near the root of a game tree, but it becomes too slow to be useful as the algorithm progresses deeper into the tree.

Each of the UCT-based search strategies included with the Gomba framework supports First Play Urgency, and it is one of the more interesting parameters to tweak. Experimental results, particularly at high branching factors, (Figures 5 and 6) leave little doubt that including the parameter can be an effective decision for speeding win rate convergence, but the exact choice of value is slightly more difficult. Extremely low values tend to result in faster convergence to a minimax-optimal solution, but at the cost of poorer choices of node difficulty. Our results with a variety of tree sizes (Figures 7 – 12) seem to imply that the best balance of difficulty and optimality convergence values and rates occur in the range of $(0.9, 1.1)$ or so, which is consistent with what is known to work well in computer Go (typically about 1.1 [12]).

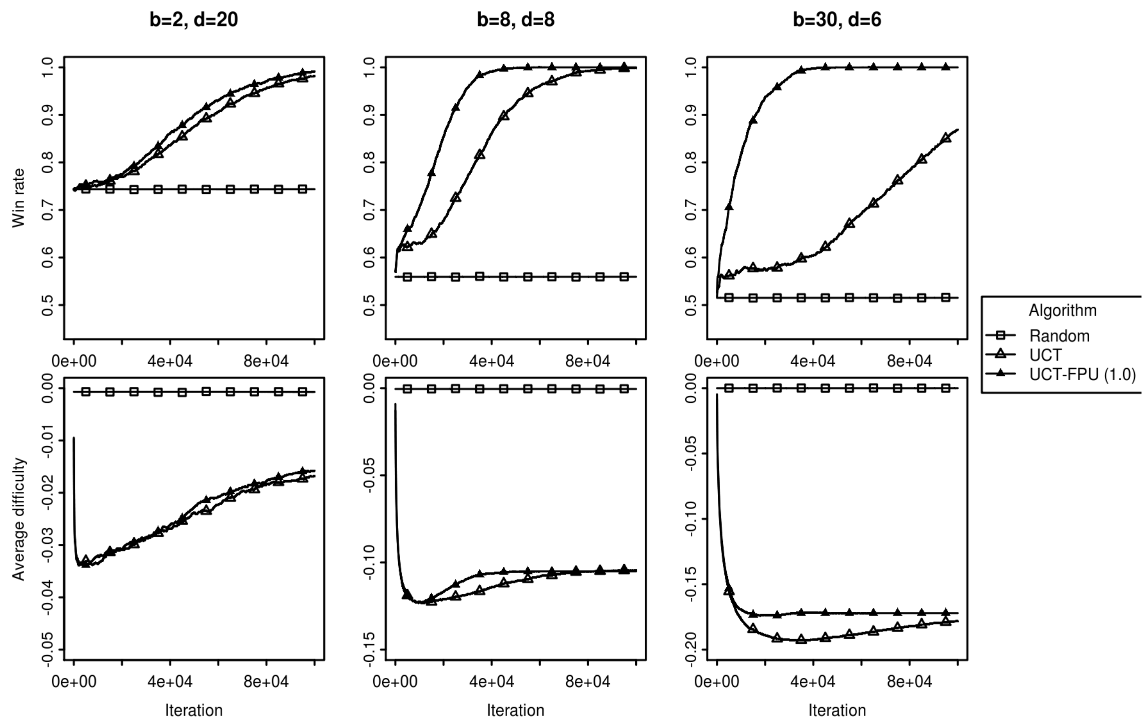

Figure 5: UCT-FPU Performance in Small Gomba Trees

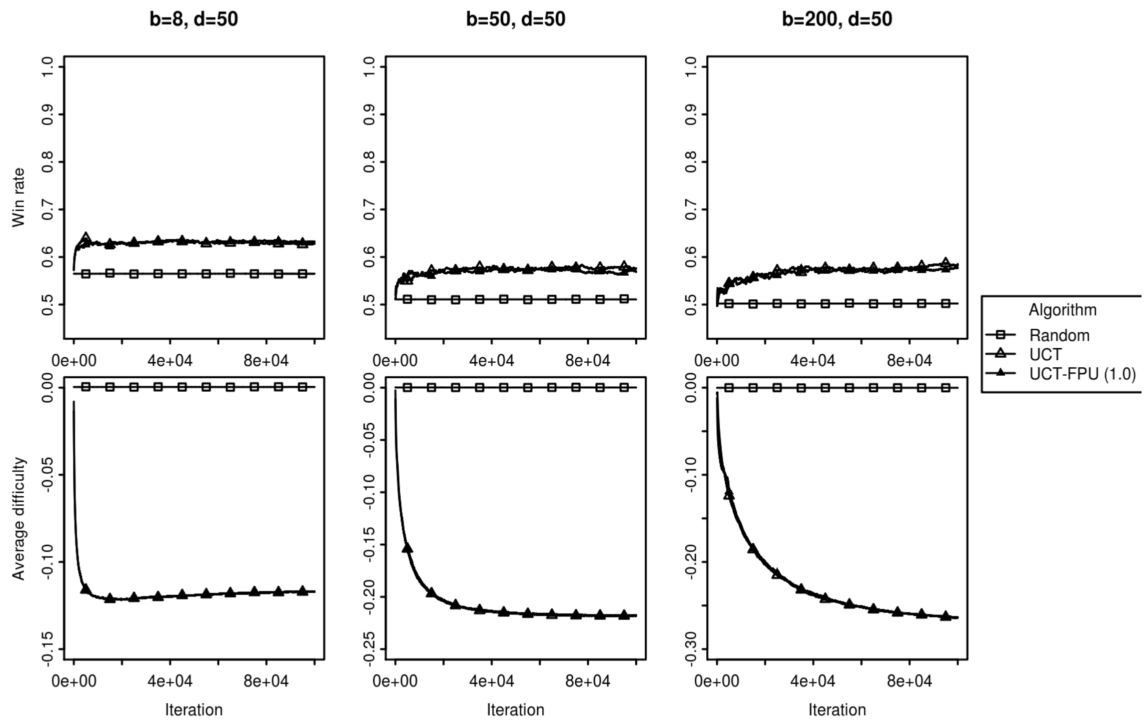

Figure 6: UCT-FPU Performance in Large Gomba Trees

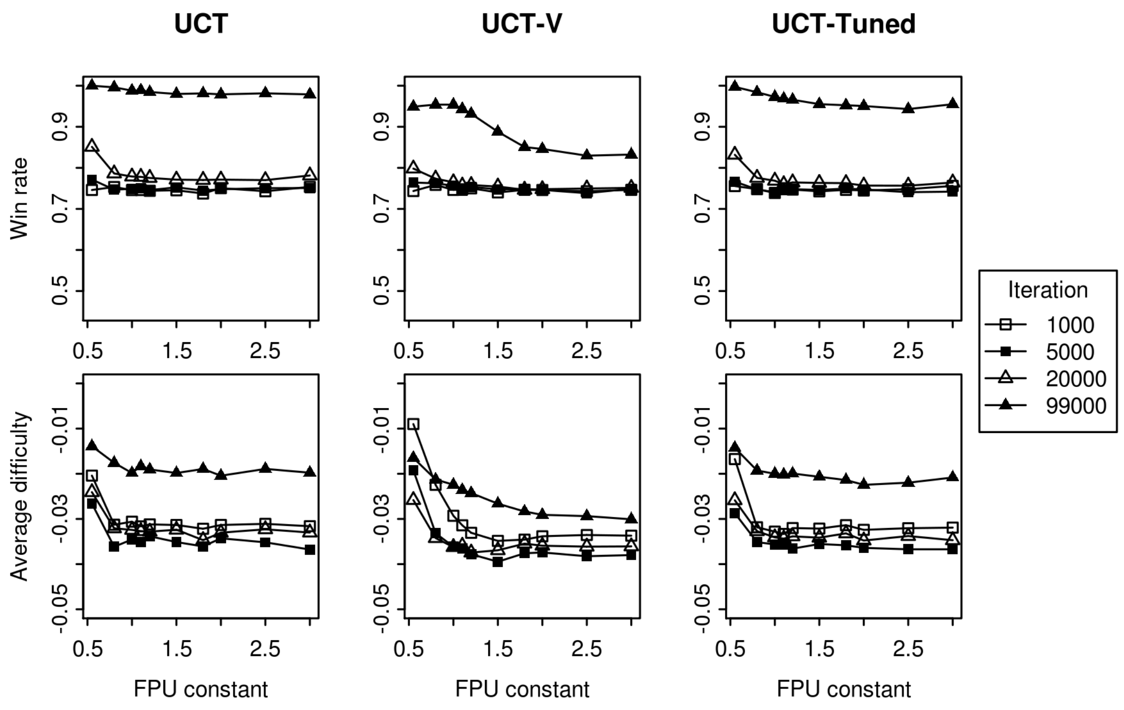

Figure 7: FPU Performance (b=2, d=20)

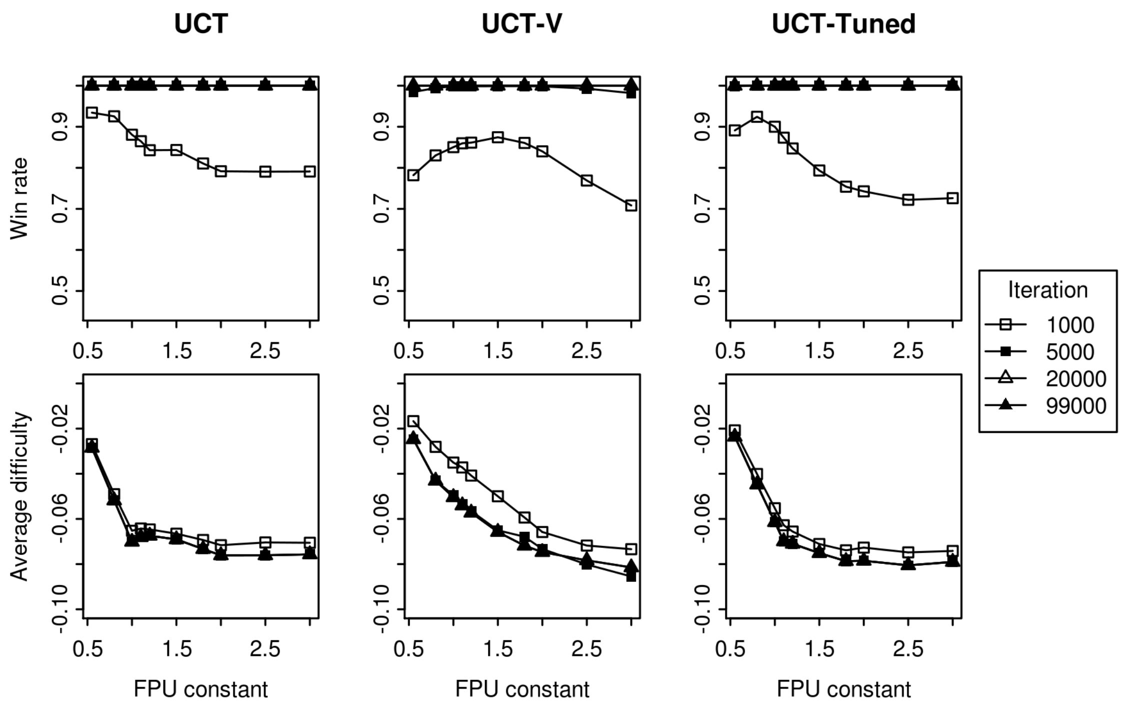

Figure 8: FPU Performance (b=6, d=6)

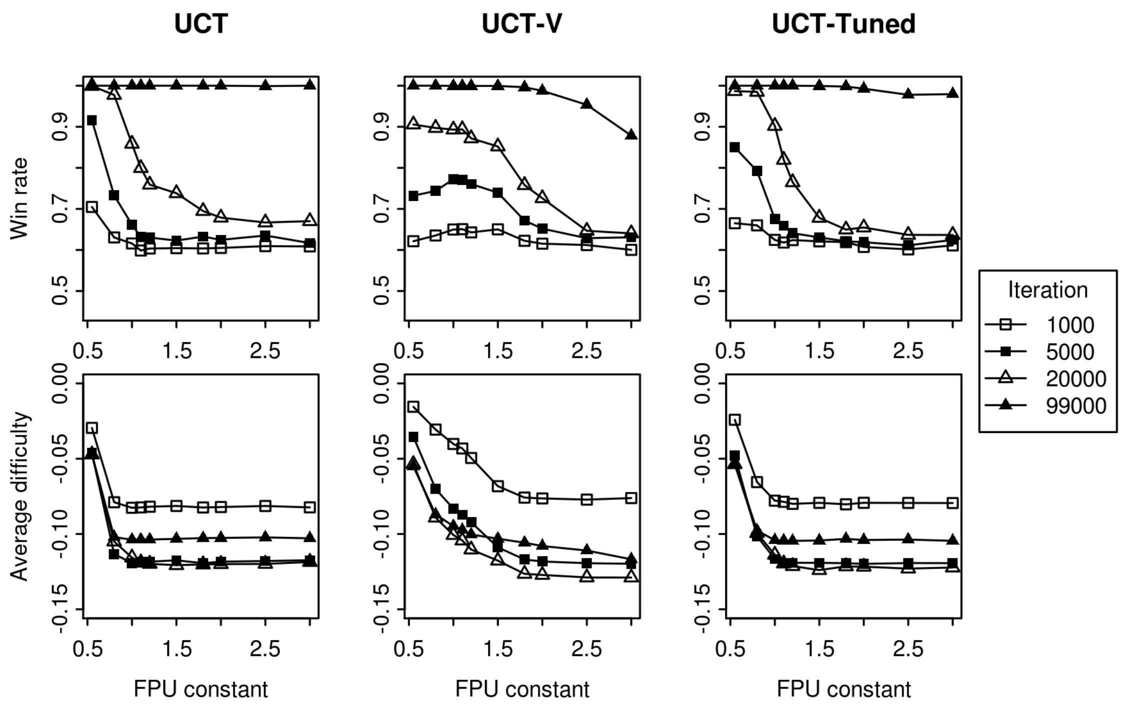

Figure 9: FPU Performance (b=8, d=8)

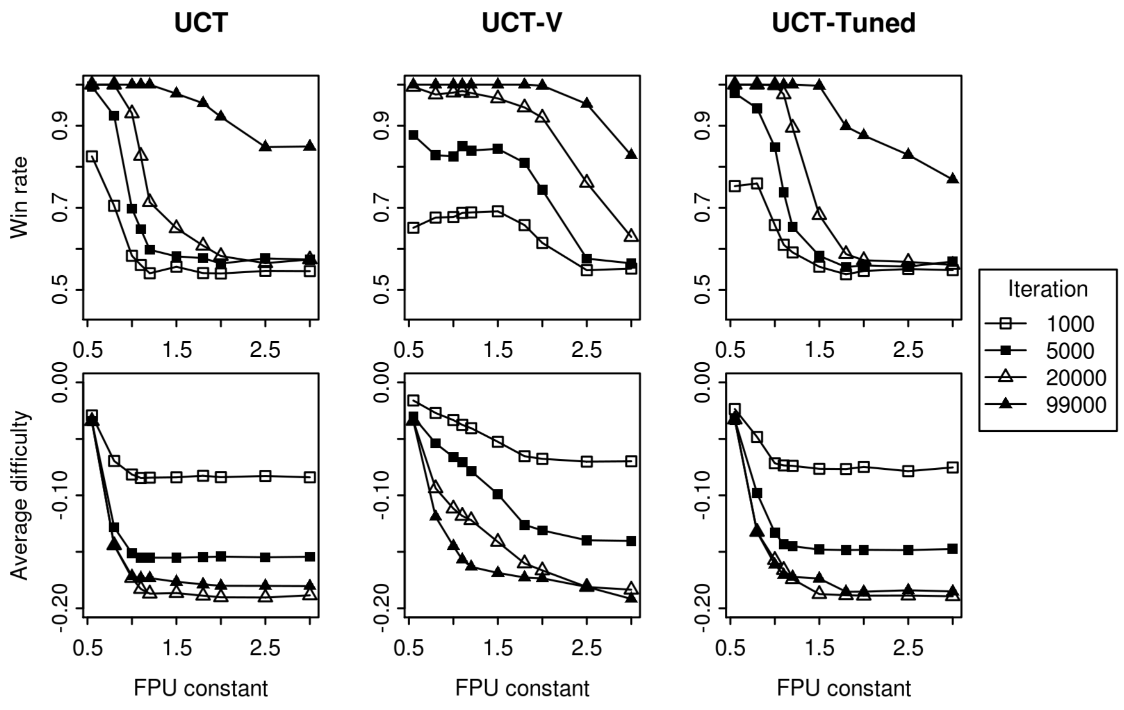

Figure 10: FPU Performance (b=30, d=6)

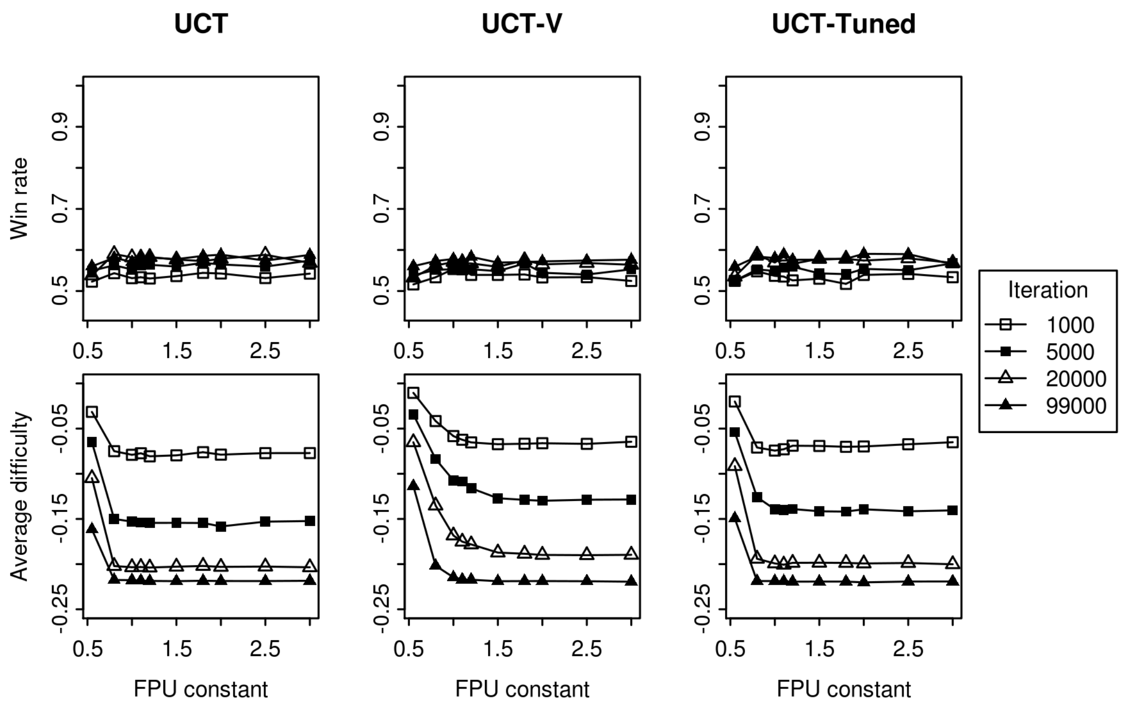

Figure 11: FPU Performance (b=50, d=50)

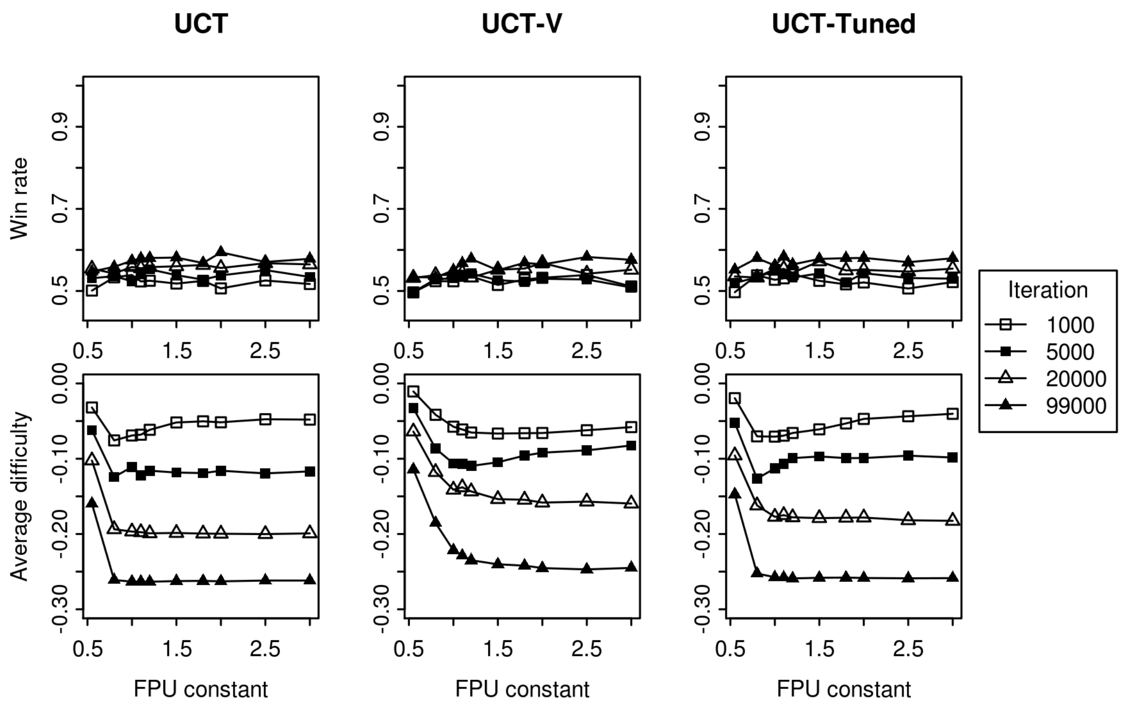

Figure 12: FPU Performance (b=200, d=50)

4.1.7 Heuristic UCT

Heuristic UCT is a modification of UCT which incorporates prior knowledge about the specific problem domain as an initializer to the UCT algorithm [19]. That is, whenever a new node is expanded by the UCT algorithm, its value is initialized according to the heuristic function rather than starting Monte-Carlo search with no knowledge. The heuristics incorporate both a notion of the value of a node and a notion of confidence in that value, which can be used to represent approximately how many Monte-Carlo searches the heuristics are "worth" as an estimator of a particular state. It should be noted that as with any method involving significant domain-specific knowledge, tests involving this method in artificial game trees may not provide accurate information about how effective the method could be in another domain, such as Go. Despite this warning, it is also worth noting that this method has been implemented with significant success in the Go-playing program MoGo [19].

The majority of Monte-Carlo based search types we have tested on Gomba are capable of taking a parameter for simulation effectiveness. When set to a nonzero value, this parameter acts to bias the random playouts used at the fringes of the generated search tree slightly towards the true minimax value. This biased playout policy is meant to emulate the behavior of the heuristic-based playout policies favored by most competitive computer Go programs. The results of applying the policy are presented for completeness but are relatively uninteresting. Tests on smaller trees (Figure 13) showed relatively little change for reasonable heuristic effectivenesses, which we attribute to UCT itself having an effectiveness which dominates the relatively low values we considered reasonable in these tests. In larger trees (Figure 14), where the algorithms are much slower to converge to winning choices, adding the effectiveness measure showed a significant improvement in win rate.

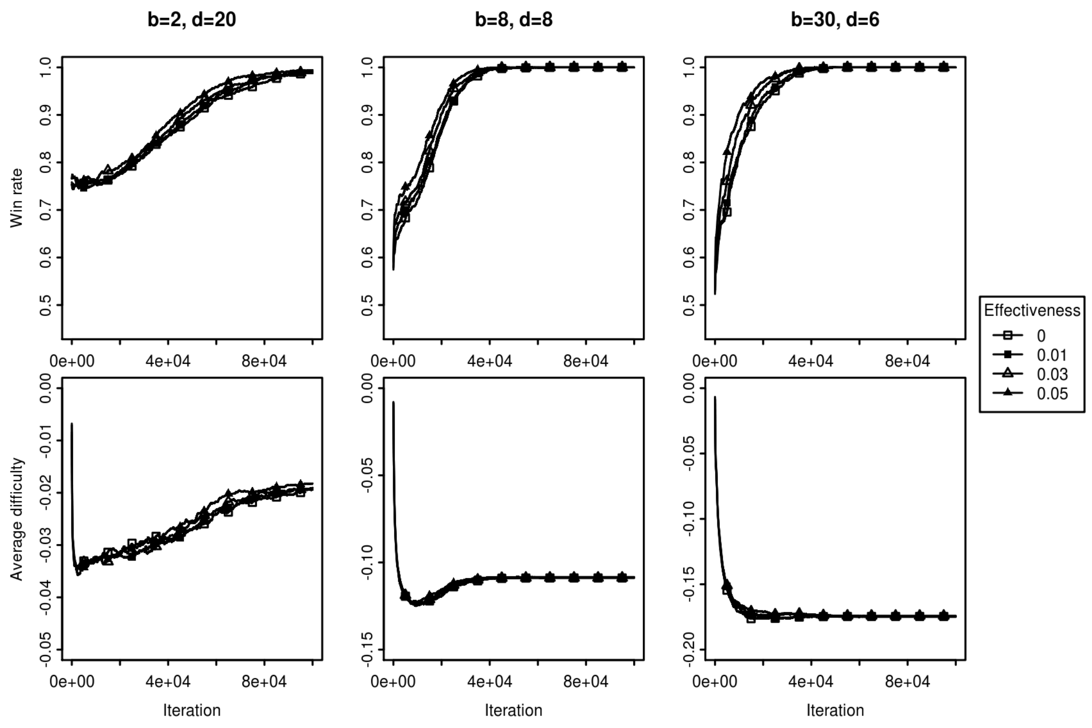

Figure 13: Simulation Policy Effectiveness in Small Gomba Trees

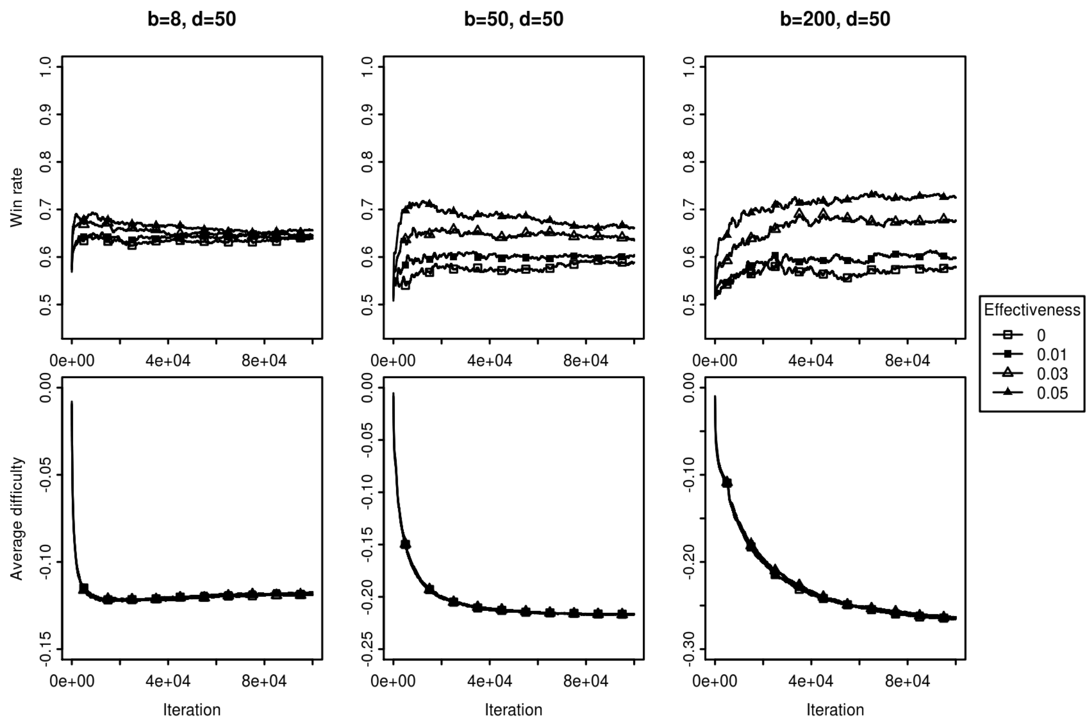

Figure 14: Simulation Policy Effectiveness in Large Gomba Trees

4.1.8 UCT-RAVE

UCT-RAVE (UCT with Rapid Action Value Estimation) is an extension of UCT which provides a means of generalizing the value of taking an action across multiple states, rather than only considering the results of actions on a strictly per-state basis [10]. Specifically, the estimated value of an action at a particular state s is allowed to take into consideration the estimated value of that action at all substates of s. This method has proved to be extremely effective in Go, where the value of moves can often be at least partially independent of the order in which they are played. Since RAVE values tend to have much lower variance than standard UCT valuation would, it tends to be more useful at the beginning of the evaluation of a node when information is limited. However, RAVE values also tend to converge to a less accurate result than standard UCT given enough time, so the typical implementation of UCT-RAVE evaluates nodes in a manner which tends to value RAVE estimates initially fairly highly (Gelly and Silver [9] suggest approximately 1000 times the weight of a normal simulation) but constantly, so that once a large number of Monte Carlo simulations have been performed they will outweigh the initial estimates.

We have not tested an implementation of UCT-RAVE in Gomba because Gomba game trees as they are currently implemented do not exhibit the action transposition property that the algorithm relies upon. That is, unlike in Go, the performance of the same action choice at different depths in a search path are independent of each other in Gomba game trees. UCT-RAVE's strong performance in Go suggests that this is a major weakness of the framework, and modifying the tree generation algorithm to include this property is a suggested avenue of future research.

4.2 Tree Properties

We have shown how the previous experiments' results vary with a variety of representative tree branching factors and depths, but these are not the only factors which can influence algorithm performance. In the interest of completeness, we have also analyzed the results of varying the other major parameter in tree generation, the difficulty bias of the generated trees.

4.2.1 Difficulty Bias

The previously mentioned results have all been based on unbiased games, that is, where the root state has a difficulty value of 0 (meaning that randomly chosen children are equally likely to be minimax optimal for PLAYER_MIN and PLAYER_MAX). This value does vary between different states in a game tree, but it is difficult to tell exactly how it affects search strategies since Gomba does not currently generate output statistics based on anything but the root node of a tree search. To compensate for this, we simply ran several experiments with a variety of difficulties assigned to the root nodes of Gomba trees.

The results (Figures 15, 17, 16) are essentially as expected, with the exception of the curious tendency of the FPU-based searches (in these graphs, UCT-Tuned uses an FPU cutoff) to converge to winning values very quickly at high difficulties. Our hypothesis is that this is because a relatively low FPU value is, in a high difficulty scenario, a fairly accurate guess of the average state's value. The premise of FPU-based search, that the first decent-looking node is likely to be good enough, is especially valid in a high difficulty Gomba tree. This is very a direct result of the forced minimax tree consistency that Gomba tree generation enforces. Even at high difficulties, the root node of the generated trees are being coerced into having at least one winning child. In a game tree of average difficulty and sufficient branching factor, Gomba trees look reasonably randomly generated despite this consistency forcing. However, when the difficulty gets high enough that it becomes unlikely for random moves to win, this forced consistency makes a noticeable change in the structure of the tree. Those nodes which are forced to change values to the non-favored player (that is, the only ones likely to be optimal) will also be among the only nodes which will force their own children to win for the non-favored player. The proportion of terminal nodes which will be winning because of this propagated forcing grows with higher difficulty, which effectively means that in high-difficulty states (particularly those with low depth), the forced nodes' improved minimax values correlate much more strongly to their improved random playout win rates than would be the case in less biased situations. Thus, the nodes which FPU-based search would likely choose not to ignore are very likely to be optimal in high-difficulty searches, which would explains the somewhat bizarre-looking disparity.

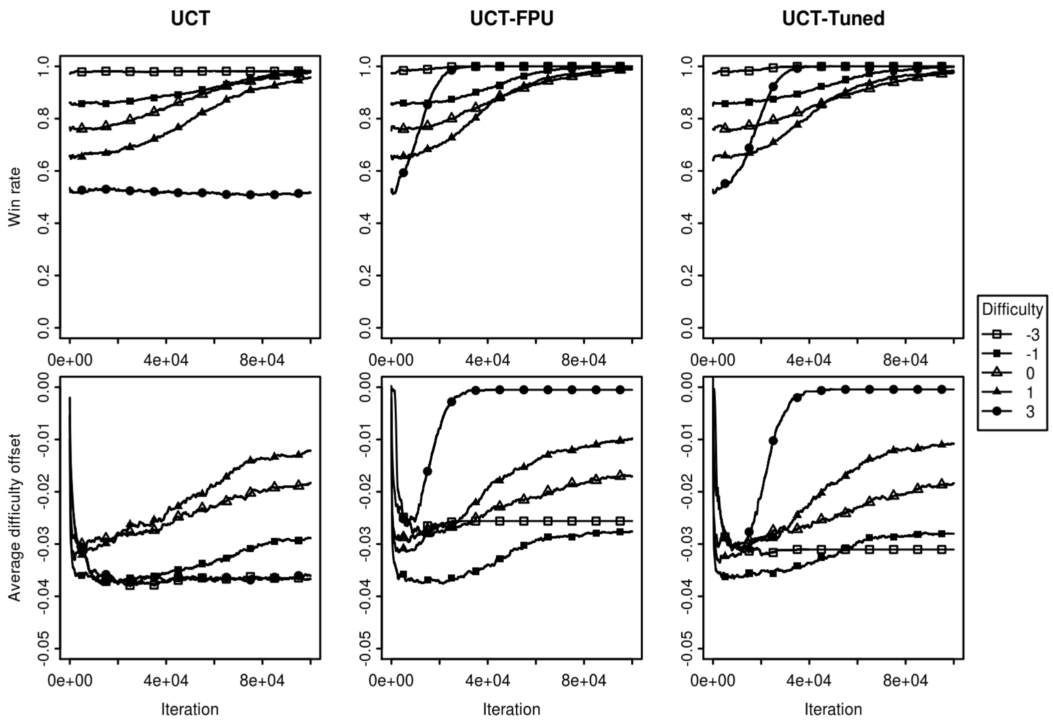

Figure 15: Effects of Difficulty Bias in Gomba Trees (b=2, d=20)

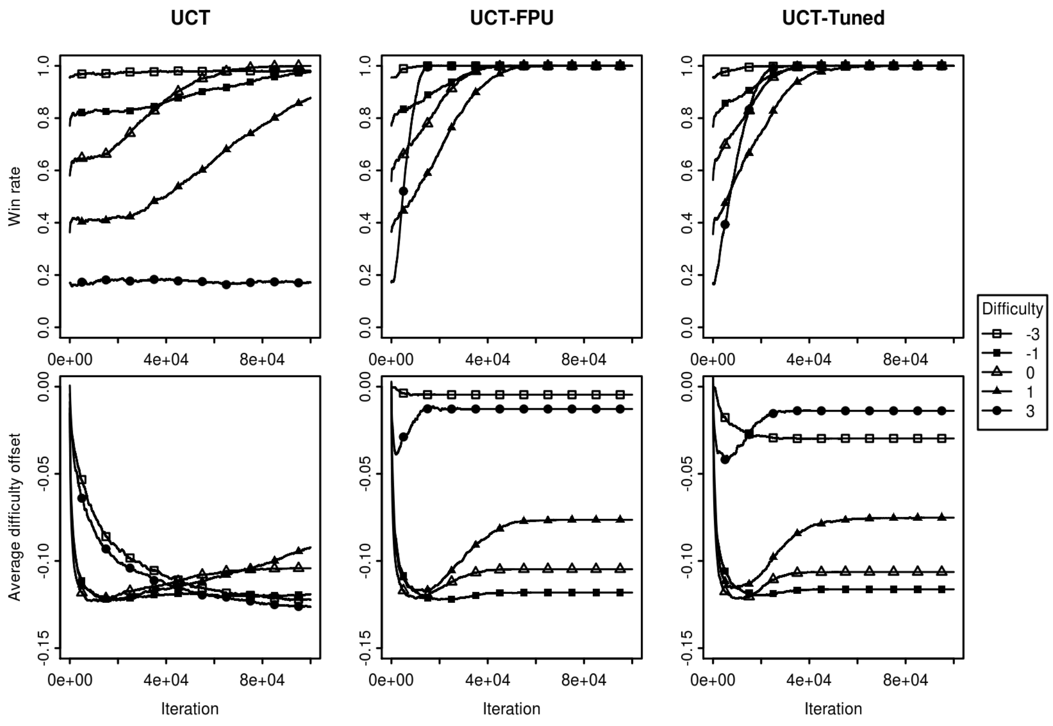

Figure 16: Effects of Difficulty Bias in Gomba Trees (b=8, d=8)

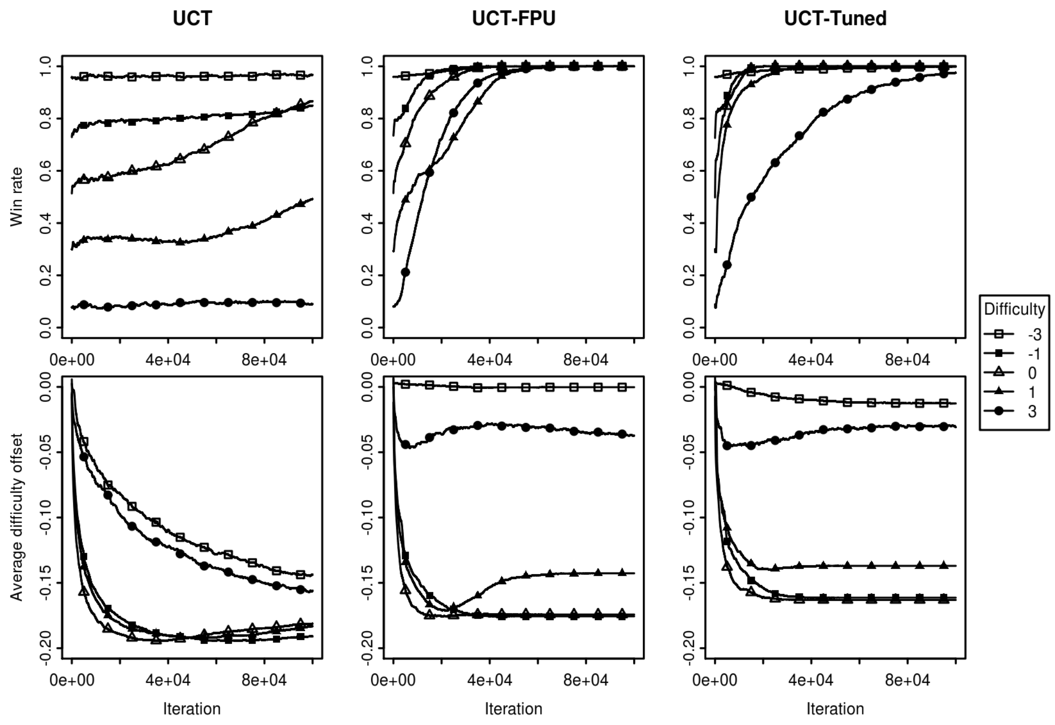

Figure 17: Effects of Difficulty Bias in Gomba Trees (b=30, d=6)

4.3 Untested Algorithms

These are algorithms which have not been extensively (or in some cases at all) tested in computer Go. We consider them here primarily as candidates for later integration with true Go libraries, such as Fuego. We primarily consider algorithms based on solutions to the infinitely-many armed bandit problem, much like UCT was taken from a solution to the standard many-armed bandit problem.

4.3.1 k-Failure

k-Failure is a simple arm selection method for infinitely-many armed bandit problems. It is based on the assumption that the problem can be expressed as a Bernoulli distribution with a success probability parameter p, representing a uniform distribution over the interval $[0, \mu*]$ for an ideal mean reward $\mu*$. The strategy is as follows: choose an arm, and play it until the kth failure occurs, and then move on to the next arm [22]. We have adapted this selection strategy to game trees of finite branch factor by using it as the selection policy in a normal rollout-based Monte-Carlo algorithm, and returning to the first arm again after the failure criteria has been satisfied for the final child of a particular state. The results in small trees, while not quite as good as modern UCT-based methods, do perform better than base UCT initially when compared against minimax optimality (Figure 18). Unfortunately, they suffer from hitting a ceiling on minimax optimality rate which is not 1, and they tend to universally perform more poorly than UCT-based methods at minimizing chosen difficulty, particularly in larger trees (Figure 19).

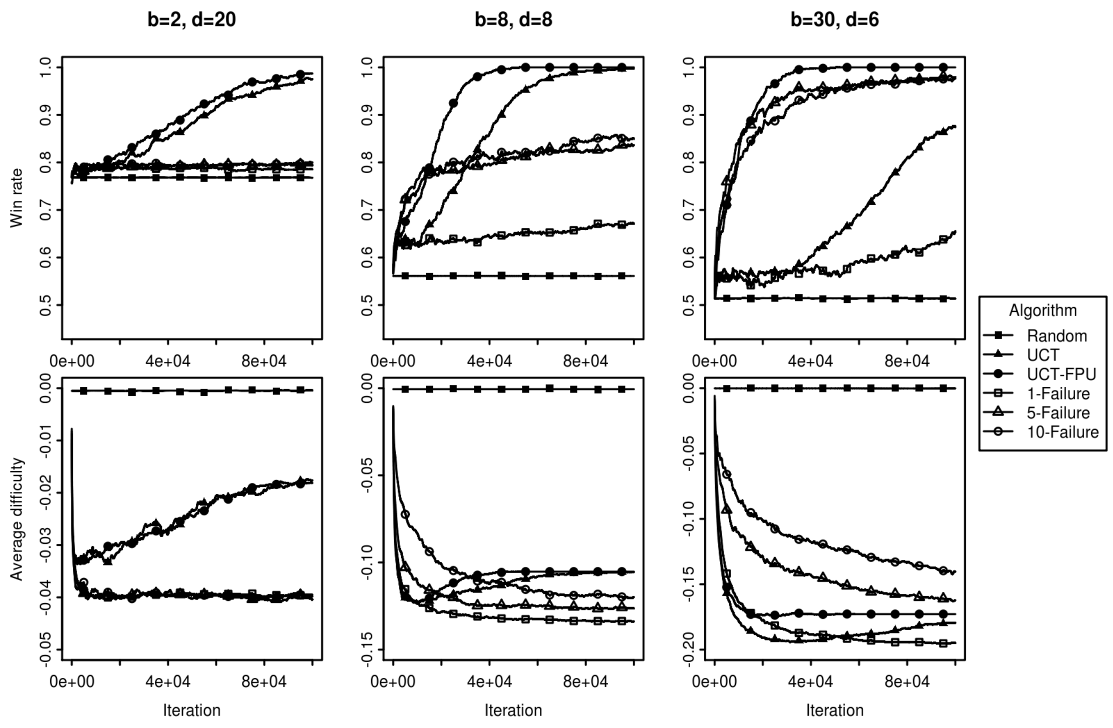

Figure 18: K-Failure Performance in Small Gomba Trees

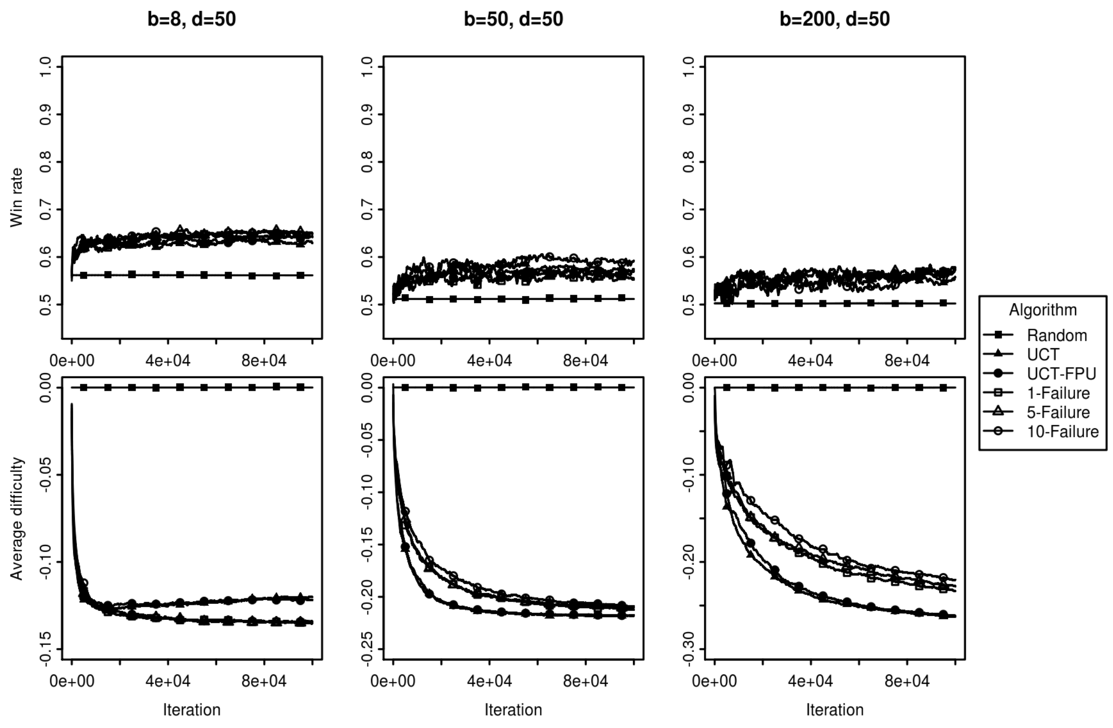

Figure 19: K-Failure Performance in Large Gomba Trees

4.3.2 k-Failure Variants

There are several other simple strategies based on the same assumptions and general structure as k-Failure. We will describe several of them here in the interest of completeness, but we have not tested them significantly due to a combination of the poor performance of k-Failure and their tendency to eventually stop switching arms, which violates the continuous exploration principle often required for convergence to minimax-optimal choices in game trees.

The m-run strategy makes the same assumptions regarding distribution and mean as the k-failure strategy. However, it differs in how it chooses moves. In this policy, an arm is played until m consecutive successes occur (a new arm is chosen on failure) or m arms are played. In the latter of the two cases, the arm with the highest average reward of the set of arms played is selected forever [22]. A variation of m-run, the non-recalling m-run strategy, does not stop exploring until it sees m successes, at which point the current arm will be selected for all subsequent runs [21]. The m-learning strategy is related to m-run's other stopping condition: It entails using the 1-failure strategy not until it sees m consecutive successes but simply until m rounds have gone by, after which point the arm with the highest success rate so far is selected and played forevermore [21].

4.3.3 UCT-V(∞)

UCT-V(∞) is one of several new algorithms we are proposing as alternatives to UCT-FPU to be able to manage the massive search tree in Go without necessarily searching every node. It is a modification of UCT which incorporates the UCB-V(∞) strategy for the infinitely many-armed bandit problem [22], which states that for multi-armed bandit problems with infinite arms, it is sufficient to simply choose some arbitrary number of random arms K and choose between them based on the UCB-V policy. In particular, since there are infinitely many "arms" (which means many of them will be reasonably close to optimal), it is feasible to choose an exploration sequence ε_t for use with the UCB-V formula which is considerably smaller than usual and still maintain reasonable bounds on the expected regret. Wang et al. recommend a function on the order of $\log \log t$ rather than the typical $\log t$ [22].

In UCT, each individual node is essentially treated as its own bandit problem. No individual node will generally approach having effectively infinite children, but a tree with a sufficiently large branching factor will very quickly approach such a state as a whole as the search deepens. The upper bound on expected regret can be minimized by selecting an appropriate K based on the total number of expected plays n and a domain-specific parameter β, which measures how close to optimal an average arm is [22]. Since the expected plays n for any particular state is based on the performance of the state, it cannot be easily estimated ahead of time. For this reason, we use a version of this algorithm which is able to compensate for this problem.

UCT-AIR (Arm Increasing Rule) is a modification to UCT-V(∞) based on UCB-AIR [22]. The primary difference is that UCB-AIR does not fix a static K at the beginning of a bandit problem (i.e., node expansion) but allows for K to increase as the search progresses. Specifically, at time n, UCB-AIR tries a new arm if

$$K_{n-1} < \begin{cases} n^{\frac{\beta}{2}} & \text{if } \beta < 1 \text{ and } \mu^* < 1 \\ n^{\frac{\beta}{\beta+1}} & \text{otherwise: } \mu^* = 1 \text{ or } \beta \geq 1 \end{cases} \tag{20}$$

Otherwise, the same UCB-V policy as in UCB-V(∞) is applied to the already drawn arms.

We were able to adapt this algorithm to UCT with little modification. It already incorporates the idea that more frequently visited nodes will be more likely to expand children, which is exactly the property we had hoped to emulate by using algorithms for infinitely-many armed bandit problems.

The results from this algorithm are somewhat encouraging. In smaller trees with higher branching factor (Figure 20), we can see an improvement over UCT-FPU for well-chosen β factors (0.7 worked well in Gomba trees). Although the results in larger trees are comparatively disappointing, providing a decrease in difficulty performance, the improved win rate in the tree of branching factor 30 led us to test this algorithm in a true Go framework.

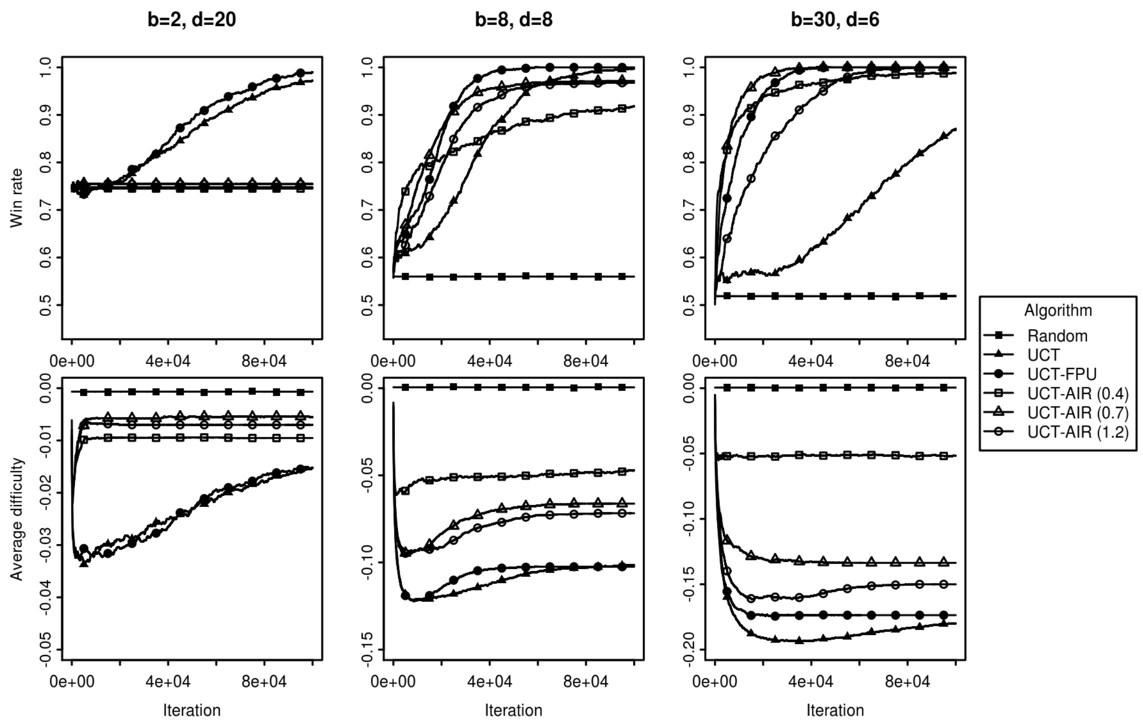

Figure 20: UCT-AIR Performance in Small Gomba Trees

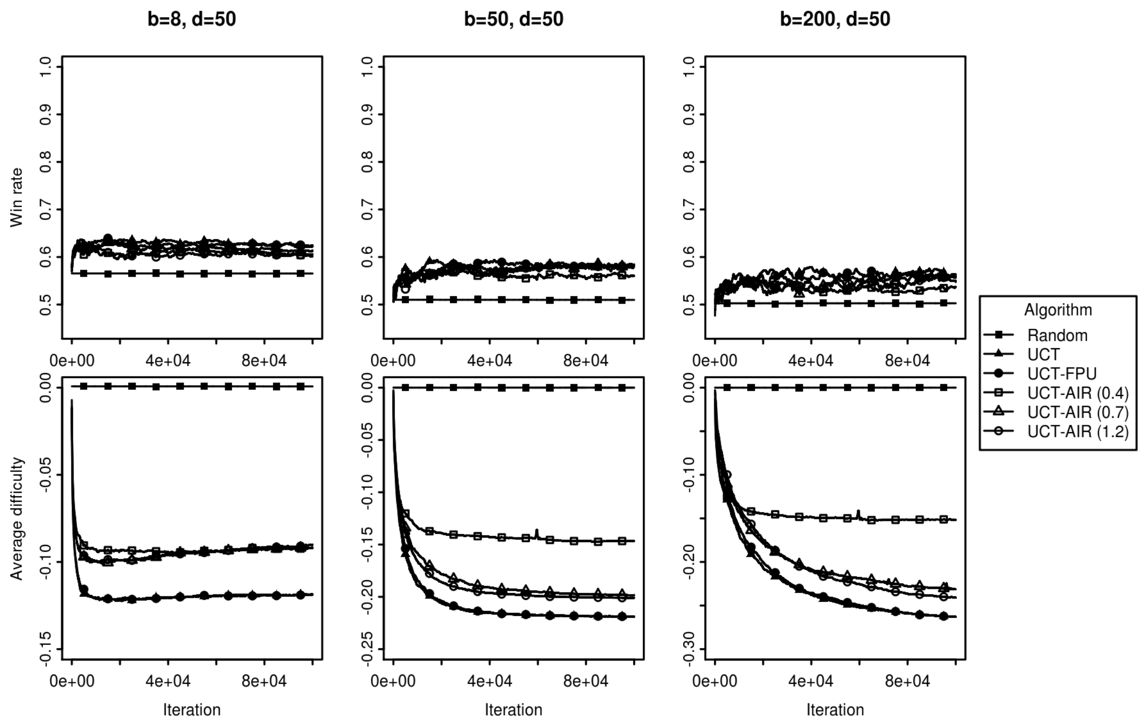

Figure 21: UCT-AIR Performance in Large Gomba Trees

4.3.4 Meta-EvE

In some games, the environment is likely to change very suddenly and drastically, rendering as invalid some previous assumptions about the game state. A meta-analysis of the game state can help in choosing between exploration and exploitation in such a volatile situation. This is what Meta-EvE attempts to do [12].

This algorithm assumes that the best current decision $i*$and its mean reward $\mu*$ are known. Meta-Eve keeps track of the rewards $x_1, ..., x_T$ received from time steps 1 to T, as well as m_T, the cumulative difference in reward values from the mean at time T. Additionally, Meta-EvE keeps track of the maximum m_i value, called M_T. From these values, one can use Page-Hinkley statistics [12] to determine whether or not a change in the environment has occurred at a given time. This is done by, in essence, signaling a change point whenever the inequality($m_t > M_T + \lambda$) is true at $t = T$. In Meta-EvE, λ is the false alarm detection parameter. Higher values will ensure fewer false detections of change, as is evident from the inequality. However, it also runs the risk of missing genuine change points. Lower values will have the opposite effect [12].

When such a change point has been detected, the algorithm handles it using discounts. Assume the following: a given decision i at time t has been visited $n_{i,t}$ times with average reward $\hat{\mu}_{i,t}$. If there is a change point at time T and a previous change point at time T_C, $n_{i,t}$ for all i and all $t = T_C, ..., T$ will be multiplied by some discount factor γ in the interval $[0, 1]$. This method, termed γ-restart, provides a means to decrease the impact of the previous environment on the current environment. For all i and t,$\hat{\mu}_{i,t}$ is not changed.

4.3.5 Probabilistic Minimax Monte-Carlo

The value update function for all of the previous Monte-Carlo variants has consistently been a rolling average of empirical results from child nodes and random playouts. We considered a modification of this method which uses a measure of the

Algorithm 2 Probabilistic Minimax Update

```
Update(state):

    if state.isTerminal:
        if state.winner = PLAYER_MAX:
            state.minimaxProbability := 1
        else:
            state.minimaxProbability := 0
        return
    default := DefaultProbability(state.numChildren)
    p := default ^ state.numUnvisitedChildren
    for each child in state.visitedChildren:
        if state.isMinimizing:

            p := p * (1 - child.minimaxProbability)
        else:
            p := p * child.minimaxProbability
    if state.isMinimizing:
        state.minimaxProbability := 1 - p
    else:
        state.minimaxProbability := p
```

probability of a move being minimax-optimal for a particular player rather than the typical empirical measure of average move outcome. The modified update function is shown as Algorithm 2.

We tested this update method with a very simple search strategy which simply blindly followed whichever path it currently thought was most likely to lead to success. We found that this search strategy was extremely effective at converging to a minimax optimal state; in smaller trees, it generally performed at least as well as UCT-FPU (Figure 22), which is exciting because its actual search policy is so trivial. However, its performance in minimizing difficulty leaves much to be desired; in smaller trees it performs much more poorly than algorithms with a more typical update policy, and in larger trees it does not do significantly better than a purely random search (Figure 23). It is possible that further research could reveal a combination of this technique and a more intelligent search policy which could prove more effective than either method alone, and it is only because of time constraints that we have not explored this avenue of research ourselves.

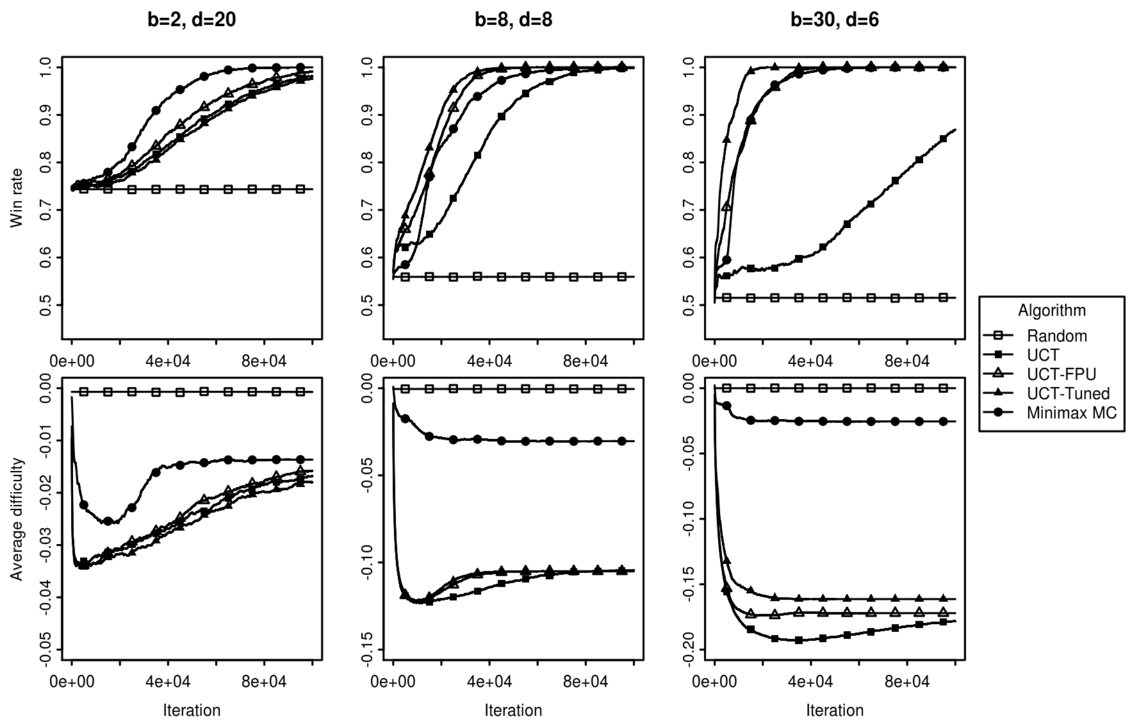

Figure 22: Probabilistic Minimax Monte-Carlo Effectiveness in Small Gomba Trees

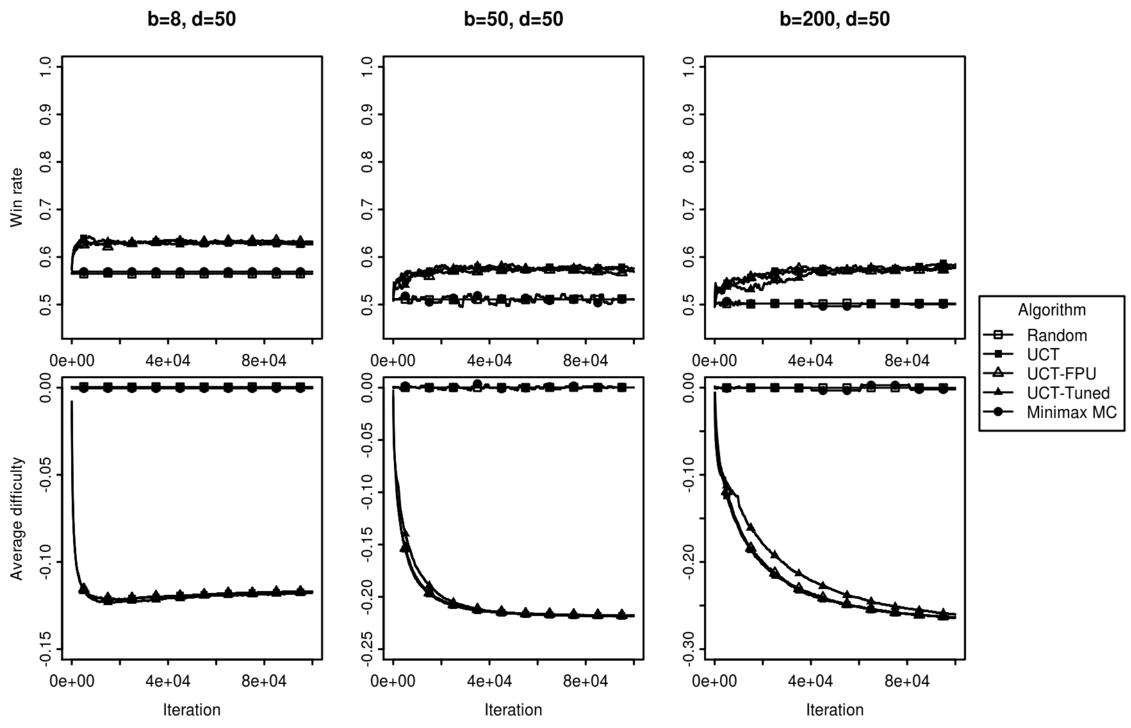

Figure 23: Probabilistic Minimax Monte-Carlo Effectiveness in Large Gomba Trees

4.4 Conclusions

Our experiments with previously tested algorithms largely coincided with the performance we expected based on previous works in other artificial frameworks and in Go itself. The results suggest that Gomba trees can be an effective tool for the rapid prototyping and testing of search algorithms, and although the tool is certainly not equivalent to testing performance in Go itself, its results can be an effective predictor of search algorithm performance in computer Go. In particular, the performance of the newly attempted variants based on arm pool limiting in infinite-armed bandit problems was promising enough to warrant further testing in a true Go framework.

5 Fuego

Despite the effectiveness of Gomba in testing search algorithms, one thing is certain: Gomba game trees are not Go game trees. While this is intentional and advantageous, it does mean that the algorithms we have selected for further testing in Go must be re-implemented within another framework specifically designed to play Go. For this reason it was necessary to find a separate framework for testing. In this section we elaborate upon Fuego [7], an open-source collection of libraries for Computer Go, and its use in testing our selected algorithms in actual Go games.

Fuego is a robust software framework for playing Go which uses UCT as its chief move selection policy, along with optional extensions to the base UCT algorithm. This makes the framework ideal for our testing purposes, as modifications to UCT are made easier with the algorithm itself already present and implemented. Additionally, Fuego is able to interface easily with other computer players via GTP (Go Text Protocol), which will be expanded further in the following section.

By implementing the selected algorithms in Go as opposed to other, artificial game trees, we are able to evaluate their performance in situations where many of the factors Gomba sought to alleviate are brought back into view. Slower evaluation time for moves and time limits (as opposed to iteration limits) are factors which a game tree framework optimized for speed can avoid fairly readily. In Go, however, this is not the case.

Changing between board states requires checking to make sure moves are legal– a process which can involve examining significant portions of the board at each iteration. Additionally, move-values are not often as clear-cut as one might like, making the problem of finding the "best" move an even harder one. These are issues that must be accounted for, though, and any robust algorithm is expected to handle them well. As such, testing on Go itself provides a means to not just test the performance of search algorithms on a specific game, but to also determine how robust these algorithms are.

For testing purposes we chose to implement UCT-V and UCT-V(∞) with a fixed K arm selection policy. In addition to these algorithms, which are new to Fuego, we chose to test the current UCT implementation with different first-player urgency values for unvisited nodes.

5.1 The Existing Codebase

Of Fuego's codebase, there are two main components that required modification to support extensions to UCT: the actual Go player itself and the GTP engine. The version of Fuego modified was Fuego 0.4.1.

5.1.1 UCT Player

Like many of the best current Computer Go players, Fuego's player component uses UCT as the core of its search for the best move [7]. As implemented, there are several common extensions and enhancements available to users when using Fuego's UCT player, but by default Fuego does not enable any particular UCT variants.

These may be configured through GTP commands, providing a convenient way to specify Fuego's behavior at runtime, which is in turn useful for testing purposes. Two features were notably missing from Fuego, though: UCT-V and the ability to use algorithms designed for infinite-armed bandit problems. For this reason we successfully extended Fuego to incorporate variance (and, by extension, UCT-V) and fixed K arm selection for UCT-V(∞).

This player's search functionality is found in the smartgame component of Fuego. In particular, UCT is implemented and modified in the SgUctSearch class, along with several other helper classes which were not modified.

5.1.2 GTP Engine

The Go Text Protocol [1], or GTP, is a text-based means of communication between Computer Go programs. Several Computer Go comply with GTP to varying degrees, which makes them much more amenable to automated plays against one another. Fuego implements GTP within an engine class, GtpEngine, through which developers can register commands and callbacks for those commands. This provides a clean interface for adding, removing, and changing GTP commands and their corresponding behavior in the Fuego backend. Such convenience was quite helpful when implementing UCT-V and infinite-armed bandit algorithms. The implementation itself, though, is not particularly notable and thus will not be discussed here.

5.2 Implementation Details

For UCT-V(∞) with a fixed K arm policy, it was necessary to modify Fuego such that it would choose only K moves from the set of legal available moves. Adding the capability to randomly select a set of K arms requierd some changes to the method of move generation Fuego uses. Generation occurs in two steps. In the first step, Fuego produces all possible board states which could come from the set of possible moves from a given board state. Obviously, many of these could be illegal moves. For this reason, the second step takes the set of all possible subsequent board states and filters it so that only states derived from legal moves remain. The challenge, of course, was to take this set of board states derived from legal moves and somehow filter *it* so that only K random states remained from that set.

This is currently implemented by, for a set of states with initial size n, removing a random state from the set for $n - K$ iterations. This behavior–removing $n - K$ random states from a set of n states–is equivalent to choosing K states from the beginning. In the case where $n \leq K$, no removals occur.

UCT-V, fortunately, was a rather trivial modification to the `GetBound()` method in smartgame's SgUctSearch class. The typical behavior for `GetBound()` relies upon statistical data gathered by the node over time, which already included mean reward information and simply needed the addition of variance. `GetBound()` was extended to use the UCT-V upper bound as specified by Wang et al. [22] when UCTV was enabled. For a given node n with s visits to its parent node (a measure of how much time has elapsed where n could be selected), t visits to n, and a reward variance of σ^2, the bias term is implemented as:

$$C\sqrt{\frac{3\sigma^2 log(log\, s)}{t+1}} \tag{21}$$

C is an exploration constant as defined by Fuego. By default, this is 0.7, but this may be changed through GTP.

Finally, AIR was implemented in Fuego by generating all possible moves upon the first addition of an arm, then caching the moves in order to avoid the need for fairly expensive computation. A single random move is then picked once it is necessary to do so according to the AIR formula, from which a new child node is generated.

5.3 Experiments

In order to test the performance of modifications we made to Fuego's player program, it was necessary to find a player suitable for the task. Due to the relatively low sample size one could gather from playing against human players, we chose to use a computer player instead. In this section, we describe the means by which we ran tests to evaluate the performance of modifications to Fuego against another computer player, GNU Go.

5.3.1 GNU Go

Released in 1999 by the Free Software Foundation in an effort to produce an open-source Computer Go player, GNU Go [8] remains one of the strongest non-commercial Go playing programs to date. Unlike Fuego, it does *not* use Monte Carlo algorithms by default when choosing a move. Rather, GNU Go combines many different move generators and evaluates all of these moves after they are generated. Individual generators are specialized to search for moves based on certain features, such as patterns on the board or opening moves based on knowledge from databases.

Once all moves are generated, each move is coupled with what are known in GNU Go as “reasons”–features of the move that make it notable–which can then be analyzed to see what is the best possible move of the set of potential best moves. Following analysis, moves are given values and the move with the highest value is chosen as the one which will be played [8]. While this behavior is very different from Fuego’s Monte Carlo techniques, it has still shown itself to be a strong contender in Computer Go competitions. We chose GNU Go because of its strength and compliance with GTP, which makes testing against it much easier than it would be with other Computer Go platforms. The version used was GNU Go 3.8.

5.3.2 Testing Parameters

Testing against the same instance of GNU Go for all variations of UCT in Fuego would not be very useful, for obvious reasons. As a result, testing was implemented with the goal of using parameters that would be able to test both the absolute robustness of our algorithms and their speed in finding useful moves. Ultimately, due to similarities between UCT-V and UCT-V(∞) when testing (both use the same formula to calculate bias), the primary parameters that were varied for UCT-V(∞) were time and number of arms. For other algorithms, time and exploration constants were used. The parameters we used for testing Fuego are available in Appendix C: Fuego Experiment Parameters.

5.3.3 Results

All of the win rates against GNU Go with the specified test parameters (see Appendix C) are in this section. Win rates are also accompanied by a 95% confidence interval.

UCT, UCT-V Results

	(5,0.7)	(20,0.7)	(40,0.7)
UCT	9.4% (±0.9)	3.9% (±0.6)	59.4% (±1.6)
UCT-V	6.5% (±0.8)	3.8% (±0.6)	55.3% (±1.6)

	(5,0.1)	(20,0.1)	(40,0.1)
UCT	3.9% (±0.6)	2.7% (±0.5)	70% (±1.4)
UCT-V	5.4% (±0.7)	3% (±0.5)	67.6% (±1.5)

UCT-V(∞) Results

(K,Max. Time Per Move)	(20,5)	(20,20)	(20,40)
UCT-V(∞)	0.8% (±0.3)	0.5% (±0.2)	3% (±0.5)

(K,Max. Time Per Move)	(30,5)	(30,20)	(30,40)
UCT-V(∞)	2.3% (±0.5)	2.1% (±0.5)	17.1% (±1.2)

(K,Max. Time Per Move)	(40,5)	(40,20)	(40,40)
UCT-V(∞)	2.9% (±0.5)	2.7% (±0.5)	33.9% (±1.5)

(K,Max. Time Per Move)	(50,5)	(50,20)	(50,40)
UCT-V(∞)	4.7% (±0.7)	3.3% (±0.6)	51.8% (±1.6)

UCT-V(∞) with AIR Results

AIR Parameter β	0.4	0.7	1.0
UCT-V(∞) AIR	47.9% (±1.6)	50.3% (±1.6)	49.9% (±1.6)

5.4 Conclusions

While UCT-V in Fuego does appear to offer some benefits over plain UCT after significant amounts of time (above 40 seconds), it does not appear to offer any particular advantage in shorter amounts of time. This is primarily due to the variance bound in UCT-V: at earlier stages of the algorithm's run UCT-V is much more focused on exploration than UCT. While this means that it will be able to explore nodes which still give erratic data after many visits (and thus have a high reward variance), the result is that early on it does not perform any better than UCT. In fact, in some cases it may even perform worse than UCT when time limits are low due to increased exploration leading to less optimal moves being chosen.

The results of UCT-V(∞) show that it performs poorly in Go for small values of K relative to the number of possible moves. The algorithm's results are highly dependent on the number of child nodes generated by each parent. In 9x9 Go, low K values lead to extremely poor performance, and with good reason – selecting the best of only a few random moves at each turn does not bode well for Fuego when using UCT-V(∞). As K increases, when higher amounts of time per turn are allowed performance increases substantially. This is due to UCT-V(∞) being unable to find a suitable move out of its possible choices in such a short time frame, in addition to small K values precluding Fuego from including good moves in its search early on in the game when the game tree's branching factor is generally higher. Using AIR instead of a fixed K provides fairly good performance, which looks promising for future work.

6 Conclusions

6.1 The Gomba Search Framework

The first major contribution of our project is the Gomba framework itself. Our experiments have proved Gomba to be fast and effective at gauging algorithm performance, and the underlying code is simple enough and abstract enough that implementing and testing new search algorithms is a relatively simple affair. Gomba is the first open, simple framework to allow for the testing of massive game trees, and we hope and expect that it will prove to be a useful tool for future research into the performance of new search variants.

6.2 Infinite-Armed-Bandit Based Search

We were able to use the Gomba framework to evaluate several new variant search methods. In particular, two new variants based on solutions to the infinite-armed bandit problem, UCT-V(∞) and UCT-AIR, have proven to be effective against both artificial Gomba trees and actual Go game trees. They represent a potentially promising new direction of research for computer Go in future work.

6.3 Future Work

Although the Gomba framework is quite usable in its current state, there are areas which could be improved by future work. Some specific areas of potential improvement, in no particular order, are:

- Modification of the tree generation algorithm to allow for the effective move transposition, as exploited by UCT-RAVE

- Addition of iterative expansion of child pointer arrays, which would allow for significant memory savings in nodes with few expanded children

- Research into statistical inconsistencies caused by the somewhat unusual usage of Linear Congruential pseudorandom number Generators

Of course, using the framework to test other new types of search algorithms would also be a valuable avenue of future research. In particular, we think that continued research into the combination of improved value update policies and current search policy in Monte-Carlo searches could lead to further performance improvements, as could further refinement of the infinite-armed-bandit based solutions we have presented.

References

[1] Go text protocol. `http://www.lysator.liu.se/~gunnar/gtp/`, 2002.

[2] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256, 2002.

[3] J. Brodeur and B. Childs. Random search algorithms. Worcester Polytechnic Institute, 2008.

[4] F. Warren Burton and Rex L. Page. Distributed random number generation. *J. Function Programming*, 2:203–212, 1992.

[5] N. Baskiotis O. Teytaud C. Hartland, S. Gelly and M. Sebag. Multi-armed bandit, dynamic environments and meta-bandits, 2006.

[6] R. Coulom. Monte-carlo tree search in crazy stone. In *Proceedings of Game Programming Workshop 2007*, pages 74–75, 2007.

[7] M. Enzenberger and M. Müller. Fuego - an open-source framework for board games and go engine based on monte-carlo tree search. TR 09-08, University of Alberta, 2009.

[8] Free Software Foundation. Gnu go. `http://www.gnu.org/software/gnugo/`, 2009.

[9] S. Gelly and D. Silver. Achieving master level play in 9x9 computer go. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, 2008.

[10] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 273–280, New York, NY, USA, 2007. ACM.

[11] Sylvain Gelly and Yizao Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go, December 2006.

[12] C. Hartland, S. Gelly, N. Baskiotis, O. Teytaud, and M. Sebag. Multi-armed bandit, dynamic environments and meta-bandits, 2006.

[13] Nihon Kiin. *Go, the world's most fascinating game.* Tokyo, 1973.

[14] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293 – 326, 1975.

[15] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[16] H Niederreiter. Random number generation and quasi-monte carlo methods. *CBMS-NSF Reg. Conf. Series Appl. Math*, (63), 1992.

[17] N. Cesa-Bianchi P. Auer and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256, 2002.

[18] A. Reinefeld and T. Hsu. An improvement to the scout tree search algorithm. *ICCA*, 6(4):4–14, 1983.

[19] Remi Munos Sylvain Gelly, Yizao Wang and Olivier Teytaud. Modification of uct with patterns in monte-carlo go. Technical report, INRIA, 2006.

[20] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[21] Olivier Teytaud, Sylvain Gelly, and Michele Sebag. Anytime many-armed bandits.

[22] Y. Wang, J. Audibert, and R. Munos. Algorithms for infinitely many-armed bandits. In *Neural Information Processing Systems*, 2008.

Appendix A: Gomba Experiment Parameters

In general, our experiments were run using many runs of many trials each on a Condor cluster graciously provided by MTA-SZTAKI. The parameter listings that follow are divided into related sets of these experiments, and are specified as lists of parameters. In the cases where a parameter is itself a list of values, the experiments were run over every combination of the multi-valued parameters.

For details on exactly how the parameters define the resulting experiments, see Appendix B.

Set 1: Comparative Algorithm Performance

- 5000 trials
- 100000 iterations
- 0 base difficulty
- Algorithms:
  - Random
  - Random Monte-Carlo
  - UCT
  - UCT (1.0 FPU)
  - UCT-Tuned (1.0 FPU)
  - UCT-V (1.0 FPU)
  - Probabilistic Minimax Monte-Carlo
- Tree sizes:
  - branching 2, depth 20

    - branching 8, depth 8
    - branching 30, depth 6
    - branching 8, depth 50
    - branching 50, depth 50
    - branching 200, depth 50

Set 2: FPU Performance

- 2000 trials
- 100000 iterations
- 0 base difficulty
- Algorithms:
    - UCT
    - UCT-Tuned
    - UCT-V
- FPU Values:
    - 0.55
    - 0.8
    - 1.0
    - 1.1
    - 1.2
    - 1.5
    - 1.8
    - 2.0

  - 2.5
  - 3.0

- Tree sizes:

  - branching 2, depth 20
  - branching 6, depth 6
  - branching 8, depth 8
  - branching 30, depth 6
  - branching 50, depth 50
  - branching 200, depth 50

Set 3: Simulation Effectiveness Effect

- 2000 trials
- 100000 iterations
- 0 base difficulty
- Algorithms:

  - Random Monte-Carlo
  - UCT

- Simulation Effectivenesses:

  - 0.0
  - 0.001
  - 0.005
  - 0.01

    - 0.02
    - 0.03
    - 0.05
    - 0.1
    - 0.2
    - 0.5

- Tree sizes:
    - branching 2, depth 20
    - branching 8, depth 8
    - branching 30, depth 6
    - branching 8, depth 50
    - branching 50, depth 50
    - branching 200, depth 50

Set 4: Difficulty Bias Effect

- 2000 trials
- 100000 iterations
- Algorithms:
    - Random
    - Random Monte-Carlo
    - UCT
    - UCT-FPU (1.0)
    - UCT-V

  - UCT-Tuned

- Difficulty Bias

  - -3
  - -1
  - 0
  - 1
  - 3

- Tree sizes:

  - branching 2, depth 20
  - branching 8, depth 8
  - branching 30, depth 6
  - branching 8, depth 50
  - branching 50, depth 50
  - branching 200, depth 50

Set 5: k-Failure Performance

- 2000 trials
- 100000 iterations
- 0 base difficulty
- Algorithms:

  - Random
  - UCT

    - UCT (1.0 FPU)
    - k-Failure ($\forall k \in 1, 2, 3, 5, 10, 15, 20$)
- Tree sizes:
    - branching 2, depth 20
    - branching 8, depth 8
    - branching 30, depth 6
    - branching 8, depth 50
    - branching 50, depth 50
    - branching 200, depth 50

Set 1: Comparative Algorithm Performance

- 2000 trials
- 100000 iterations
- 0 base difficulty
- Algorithms:
    - Random
    - UCT
    - UCT (1.0 FPU)
    - UCT-V
    - UCT-V (1.0 FPU)
    - UCT-AIR ($k = 1, \forall \beta \in .1, 2., .3, .4, .5, .6, .7, .8, .9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5$)
- Tree sizes:

- branching 2, depth 20
- branching 8, depth 8
- branching 30, depth 6
- branching 8, depth 50
- branching 50, depth 50
- branching 200, depth 50

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 of which can be found at http://www.apache.org/licenses/LICENSE-2.0. The source code for the project can be found at its Google Code page at http://code.google.com/p/gomba-mqp/. The code itself is fairly well documented, and the framework as a whole is relatively simple - we thus recommend that for questions pertaining to specific components of the framework, you reference the comments in the source itself. This appendix is meant to serve not as a complete reference to every component, but as a general overview of how to use the framework to test algorithms against parameters of your own choosing and how to extend the framework to encompass new search strategies.

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 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 files 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 files 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 codebase 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 us-

age 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 specification of the form "string_name1,2,...,K" to a search algorithm constructed with the call "ClassName(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 defined 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

Modifications to Fuego were tested on the Condor cluster which MTA SZTAKI kindly allowed us to use for the duration of this project. The parameters for the experiments were chosen so that we would be able to measure the effects of time (for all algorithms) as well as K and β values for arm selection (for UCT-V(∞)) on the performance of Fuego against GNU Go. An algorithm that is able to find a good move on par with or faster than its rivals is highly useful for game playing, and thus we sought to determine whether or not our modifications to Fuego truly did lead to useful move selection in Go.

Table 1: Fuego Experiment Parameters for UCT and UCT-V

(Max. Time Per Move, C)	(5,0.7)	(20,0.7)	(40,0.7)
Trials	1000	1000	1000
Maximum Time Per Move (seconds)	5	20	40
GNU Go Difficulty	4	4	4
Fuego Exploration Constant C	0.7	0.7	0.7

(Max. Time Per Move, C)	(5,0.1)	(5,0.1)	(5,0.1)
Trials	1000	1000	1000
Maximum Time Per Move (seconds)	5	20	40
GNU Go Difficulty	4	4	4
Fuego Exploration Constant C	0.1	0.1	0.1

Table 2: Fuego Experiment Parameters for UCT-V(∞) with fixed K arms and AIR

(K,Max. Time Per Move)	(20,5)	(20,20)	(20,40)
Trials	1000	1000	1000
Maximum Time Per Move (seconds)	5	20	40
K (number of arms to select)	20	20	20
GNU Go Difficulty	4	4	4
Fuego Exploration Constant C	0.7	0.7	0.7

(K,Max. Time Per Move)	(30,5)	(30,20)	(30,40)
Trials	1000	1000	1000
Maximum Time Per Move (seconds)	5	20	40
K (number of arms to select)	30	30	30
GNU Go Difficulty	4	4	4
Fuego Exploration Constant C	0.7	0.7	0.7

(K,Max. Time Per Move)	(40,5)	(40,20)	(40,40)
Trials	1000	1000	1000
Maximum Time Per Move (seconds)	5	20	40
K (number of arms to select)	40	40	40
GNU Go Difficulty	4	4	4
Fuego Exploration Constant C	0.7	0.7	0.7

(K,Max. Time Per Move)	(50,5)	(50,20)	(50,40)
Trials	1000	1000	1000
Maximum Time Per Move (seconds)	5	20	40
K (number of arms to select)	50	50	50
GNU Go Difficulty	4	4	4
Fuego Exploration Constant C	0.7	0.7	0.7

Trials	1000	1000	1000
Maximum Time Per Move (seconds)	40	40	40
GNU Go Difficulty	4	4	4
AIR Parameter β	0.4	0.7	1
Fuego Exploration Constant C	0.1	0.1	0.1