Efficient Control of Selective Simulations

Brian Sheppard

Sheppard Company, Inc,
Concord, MA, USA
sheppardco@aol.com

Abstract. Selective simulation is a search technique that estimates the value of a move in a state space by averaging the results of a selected sample of continuations. The value of selective sampling has been demonstrated in domains such as Backgammon, Scrabble, poker, bridge, and even Go. This article describes efficient methods for controlling selective simulations.

1 Introduction

The domains dealt with are characterized by three issues: (1) non-determinism, (2) imperfect information, and (3) high branching factors. In such domains, exhaustive algorithms achieve a shallow search depth, which is frequently insufficient. The ineffectiveness of exhaustive search seemingly leaves static analysis as the only option for such domains. Searching for an alternative, the question arises: can a selective search improve upon static analysis?

In some domains it is possible to create effective selective-search policies. This is not true in every domain, but it is true in many domains that are of current research interest, such as Backgammon [9], Scrabble [11], poker [3], bridge [10], and even Go [5,7]. The selective-search methods described in this paper employ *selective sampling* of possible continuations. The distinction between selective sampling and Monte Carlo is subtle and maybe not even well defined, but here is an attempt at the distinction: in selective sampling we try to *choose* continuations that make the distinctions between alternatives clearer, whereas in Monte Carlo we select samples in the same distribution as the branching structure of the domain. This difference should not obscure the goal of selective simulation, which is the same goal as that of a Monte-Carlo simulation: to find the best move We just want to find the best move more quickly.

For example, assume that the player to move is playing a bear-off in Backgammon. The player considers two options. After either option the opponent can roll a doublet to bear off all his men, so the two moves work out equally in that case. In a selective simulation framework, the player to move can cease considering doublets, since any difference between the plays must be revealed in the other rolls. By avoiding doublets, the player reduces CPU usage to 5/6 of a Monte-Carlo simulation, but still arrives at the best move. In this example, the payoff is tiny. In other situations, the payoff of selectivity can be arbitrarily large.

The course of the article is as follows. Section 2 provides the basic simulation framework. In Section 3 three illustrative domains are introduced: Hold'em poker, Scrabble, and Backgammon. The emphasis is on Hold’em poker, since for that game the data used are specifically collected for this article. The data for the other games are from the literature. Section 4 focuses on generating plausible moves. The essence of this article is in Section 5: Selecting a sample of continuations. Section 6 briefly discusses time control. In Section 7 the issue of selecting a sample move (in relation to a continuation) is considered. Section 8 provides methods on how to evaluate the end result of a simulation. In Section 9 a summary is given.

2 Basic Simulation Framework

The following pseudo-code outlines the process of simulation.

1. Generate plausible moves.
2. Select a sample of continuations (an arbitrary process that can generate state changes).
3. While time remains for further simulation:
   - select a continuation,
   - select a move to simulate,
   - follow up that move with the selected continuation,
   - evaluate the end result,
   - average with all previous results for this plausible move.
4. The move with the highest average is the best.

The choices that a player makes throughout this framework determine the effectiveness of the search policy. The article discusses some options for each step in the framework. Among the options that fit the framework is a pure Monte-Carlo approach.

3 Illustrative Domains

The article contains several examples, most from three domains: Hold’em poker, Scrabble, and Backgammon. The Scrabble [12] and Backgammon [13] examples mostly summarize data from existing literature. The Hold’em data was collected specifically for this article, and does not appear elsewhere.

The game Hold’em poker is described by Billings, Papp, Schaeffer, and Szafron [2]. For this purpose, it suffices to note that each player has 2 hole cards, and there are 5 community cards. The winner is the player that makes the best 5-card hand by combining his hole cards and the community cards.

The Hold’em engines used in these experiments were rudimentary. They are nothing special in themselves, and this article is not about them. It suffices to note that they play at an intermediate level, and exhibit the variability of the domain. They are typical of the programs that a developer might struggle to improve.

Hold’em can be played with up to 10 players, but in this article we will use the 2-player game. Each experiment measures the difference between two players by playing 1600 games of Hold’em. The difference between the players is reported in the units of "small bets per hand," a standard measure of skill. We will perform each experiment 400 times, and measure the standard deviation of the mean measured by each experiment. The goal of the article is to quantify the advantage of selecting a sample in a systematic way, rather than simply using Monte Carlo.

The baseline Monte-Carlo simulation used no selectivity at all to compare the players. That is, a sample of 1600 random hands was played out, and the results were summarized. The standard deviation of the measurement was 0.251 small bets. The true value of the difference between the players in the experiment was known to be about 0.025 small bets per hand.

This poses a huge obstacle to progress. To quantify the problem, consider that an advantage of 0.025 small bets per hand is considered significant in Hold'em. Yet, the standard deviation of a 1600-game Monte-Carlo simulation is ten times as large. To reduce the standard deviation of a Monte-Carlo sample to only 0.025 would require about 160,000 trials. This is feasible, but time consuming. In addition to the time, there is a 16 per-cent chance that the weaker program would have better simulation results. Drawing the wrong conclusion because of statistical noise can set back a research agenda for months.

Now consider a more typical case, where the players differ by a small amount because of some tweak. Assume that the difference is 0.005 small bets per hand. To detect such a difference with 84 per-cent confidence would require 4,000,000 trials! Such differences are very expensive to detect, yet a difference of 0.005 is a worthwhile improvement. If a player wishes to produce a program that is aware of the subtleties of the game, then the player would have to detect such small differences.

4 Generating Plausible Moves

The selective simulation framework starts with generating a list of moves to simulate (i.e., at the route of the search). Below we briefly discuss four related topics concerning the list of plausible moves: the need for selectivity (in 4.1), the flexibility (in 4.2), a breadth-versus-depth decision (in 4.3), and metrics (in 4.4).

4.1 Need for Selectivity

In the ideal case a player can consider all legal moves. For example, in 9×9 Go, there are never too many legal moves, and a program can consider all of them [7]. Another example is poker, where the options are either Bet, Call, or Fold. The number of legal moves in Bridge is also quite manageable, so a program can consider all of them.

Other domains are not so simple. In Backgammon, for instance, it is possible to have hundreds of legal moves, particularly if one rolls small doublets such as 1-1 or 2-2. In Scrabble, the number of legal moves averages over 700. Moreover, one can imagine war games where the number of legal moves is so large that the complete list cannot even be generated. In such cases a program needs to generate plausible moves by a static analysis.

4.2 Flexibility

One valuable feature of a simulation framework is that the plausible-move selection process does not need to be particularly discriminating. A program is not relying on the plausible-move generator to determine the best play. The program simply wants the best move sorted sufficiently high that it makes a cut. This can be helpful. For example, if a program wants to select good Scrabble moves by static analysis then it should include some evaluation of access to triple word squares. But that is not necessary if the moves will be simulated. Triple word squares are neither significant enough to deny a move its place on the list of moves, nor to assure a move its place.

Plausible-move generation need not be an "all or nothing" gamble. It is possible to simulate a few moves, then look at the outcomes and decide to add a few more moves. On the basis of simulations it is possible to discover that a move that was initially discarded should be considered. The Go proverb to "move where your opponent wants to move is an example of a similar heuristic." In Scrabble, for example, a simulation may tell you where the opponent obtains big scores, and then you can generate moves that block those locations. MAVEN [12] does not actually do this, as we have not yet seen a case where it is necessary, but the possibility is attractive.

4.3 A Breadth-Versus-Depth Decision

Simulating more moves means that there is less chance of overlooking the best play. But there is a downside: more moves means that the program either uses more time, or can simulate fewer continuations. Having fewer continuations means that there is a greater chance of missing the best play because of statistical noise.

Each program faces the breadth-versus-depth decision after a time limit for the simulation has been fixed. From the time limit, the program can estimate how many trials it will be able to do in the available time, and from that it can figure how many moves it wants to spread those trials over. There is no hard and fast rule for how to do this. In general there is a tradeoff between the chance that the Nth move is best versus the chance that the best move will not be selected because of less precision. In judging this, keep in mind that the number of trials has to be quadrupled in order to halve the standard deviation.

4.4 Metrics

Each programmer should use a metric to evaluate the effectiveness of a plausible-move generator. This subsection briefly discusses two metrics. One is well known and has been described in the literature. The other was created for the MAVEN project, but has not yet been described in the literature.

A common metric is to count how often the generated moves include the moves actually selected by expert human players. This metric is frequently used in Go, and has also been used in Chess [1]. The metric has the advantage of being quick to evaluate. However, it has the disadvantage of providing no information about the quality of second-best moves.

A more robust metric is the average number of points lost. This metric is more complicated to calculate, and may be impossible to calculate, but it is more useful when it can be achieved. For example, assume the time control for a Scrabble program is set such that the program has time to simulate only 10 moves. Assume further that we fix a policy P that chooses 10 moves. During development, we can simulate 50 moves instead of 10. When the simulation selects a move that P ranks in the top 10, then the loss due to P is 0. When the simulation selects a move that P does not rank in the top 10, then the loss due to P equals the evaluation of the selected move minus the evaluation of the best move that P selected. This metric is useful when you do not have a large supply of expert games, or when the program plays better than humans.

5 Selecting a Sample of Continuations

A peculiar feature of the basic simulation framework described in Section 2 is seen in the lines 1 and 2: the selection of continuations is listed as a separate step from the generation of moves. Hence the question arises: Does the framework require that a domain's continuations are independent of the moves?

If the continuations of a domain are intricately related to the moves, then selective simulation might not be a good search engine. For example, I would speculate that simulation could not play a game such as Reversi, where the legality of continuations depends heavily on every legal move.[1] But when the domain has less dependence on the initial position, then simulation can be a better match. For example, the legality of continuations in a Go position has very little dependence on the first move. Brügmann [7] and Bouzy and Helmstetter [6] showed that simulations are surprisingly effective in Go.

Simulation is at its best when there is hidden information, because then the first move of the continuation is to select hidden information for the opponent. Such information is necessarily independent of the move choices, so the framework described above is completely correct in that case. If the domain additionally has randomization, then simulation is an almost perfect match.

When selecting continuations, the goal is to achieve convergence of the simulated results faster than simple random selection. Ideally, this should be done without biasing the results. The techniques described below fall into two broad categories. The first category attempts to reduce the variance of sampling. The techniques used in this category are: explore parallel continuations (see 5.1), adjust for non-uniformity (see 5.2), and enforce uniformity (see 5.3). The second category attempts to amplify differences by focusing on continuations in which outcomes differ (see 5.4).

5.1 Explore Parallel Continuations

The simplest approach to achieve convergence and to reduce the variance of sampling is to use parallel continuations. That is, apply each continuation after each plausible move. This is a very simple technique for controlling variance, since it means that any biases that exist in the sample are the same for all moves. For example, in Hold'em

[1] Of course, there are good deterministic algorithms for Reversi.

simulations, you can evaluate the Bet and Call options after each possible pair of opponent's hole cards. This way, it is impossible for the opponent's hole cards to be better for either option. Below we elaborate on reducing the variance by duplicating trials (in 5.1.1) and by synchronizing random generators (in 5.1.2).

5.1.1 Duplicate Trials in Hold'em

We can reduce variance by playing each deal twice, once with each player moving first. This simple change uses only half as many deals, but the luck of the draw is lower because the cards exactly balance. Using duplicate deals is a trick taken from bridge tournaments, and first applied to poker by Billings et al. [2].

Duplicate trials appear to be interdependent, since a duplicate experiment is actually 800 trials each of which measures the *difference* between the players. However, this does not change the statistical formulas. The observed standard deviation of the mean in duplicate trials is 0.075. Because the number of trials required to achieve a specific level of accuracy is a quadratic function, the reduction in variance from 0.251 to 0.075 is equivalent to an increase in speed of a factor of 11.

Not every such experiment will produce such drastic gains. The structural similarity of the poker players used in this experiment makes the gains more drastic, since there is a strong tendency for play to cancel across pairs of games. The degree of improvement that a program may experience in its own simulations may vary. The same comment applies to all of the experiments in this article.

5.1.2 Synchronizing Random Generators

The fundamental decision unit of poker AI is the "strategy triple". It is the triple (F, C, R) of positive real numbers such that F + C + R = 1. A poker AI program generates such a triple at every decision point, to represent the decision to Fold with probability F, Call (or Check) with probability C, and Raise (or Bet) with probability R.

It is natural that the probability triples of well-tuned players will have a large amount of overlap, which means that the programs will often play the same strategy. In the event that the programs play the same strategy to the end of a game then the difference between the programs is zero. When using the duplicate sampling technique (cf. 5.1.1), we can synchronize the random state so that the same sequence of numbers is generated on the duplicate plays of a deal. This reduces the variance of a duplicate sample from 0.075 to 0.048. (Synchronizing random generators makes no difference without also duplicating trials.) This reduction in variance is equivalent to a speedup of 2.4 times.

5.2 Adjust for Non-uniformity

Sometimes it is easier to remove the impact of non-uniformity after it has happened. For example, in a poker simulation a player must have some notion about the distribution of the opponent's hole cards. Rather than generating the hole cards in the assumed distribution, the player can generate a uniform distribution, and then weight the outcomes according to the presumed distribution. This works well in poker when the opponent's hole cards are easily enumerated, as they are in Hold'em. It would not work so well in 5-card draw, where the opponent has one of C(47, 5) possible hands.[2]

One downside to adjusting after the fact is a loss of information. We take a simplified example. Assume that the domain has two semantic categories that should occur in ratio 9 to 1. If the program makes a sample containing 1000 of each and scales them down to 9 to 1 ratios, then the effective sample size is only 1000 + 1000/ 10 = 1100, whereas the program has taken 2000 samples. In effect, each sample that the program has taken from the minority option counts as only 1/10 of a sample. This loss can be viewed as inefficiency, or as increase in variance, or as a loss of information, but it is a bad thing in any way the program looks at it.

For this reason, such manipulations are a poor design choice unless the programmer has no alternative. But there is one situation where the manipulations make a lot of sense: namely, if a program has simulated a sample under one distribution, and then the program wishes a simulation under a different distribution. For example, in Scrabble the program can do simulations assuming that the opponent's tiles are uniformly random. But then the program might conclude on the basis of the opponent's last turn that the opponent was unlikely to have kept the Q. It would cost a lot to do a new simulation under this hypothesis. But it is inexpensive to reweigh the outcomes under the new distribution. If necessary, after reweighing the program can add trials to categories that are very important in the new distribution and that did not occur often under the original. Below we give an example of class weighting in Hold'em.

5.2.1 Class Weighting in Hold'em

The goal of class weighting is to eliminate the variance that arises from unequal distributions of hole cards. Though there are C(52,2) = 1326 different pairs of hole cards, there are only 169 essentially different pairs after symmetries are accounted for. The 169 equivalence classes are as follows:

1. 13 hands consist of a pair of cards of identical rank ("Pocket pairs").
2. C(13,2) = 78 hands have different rank and the same suit ("Suited").
3. C(13,2) = 78 hands have different rank and different suits ("Off-suit").

The number of symmetries of each hand is as follows:

1. Pocket pairs occur with weight C(4,2) = 6.
2. Suited hands occur with weight C(4,1) = 4.
3. Off-suit hands occur with weight P(4,3) = 12.

The implementation of class weighting is to recombine the samples according to the true frequencies of each class, as follows:

```
Sum = Count = 0;
for (each equivalence class) {
   if (there are trials in this class) {
```

[2] Categorize hole cards at a higher level of aggregation (e.g., ace-high, two-pairs, ...) to apply this method to 5-card draw.

```
            v = total in this class / number of trials in this class;
            Sum += v * equivalence class weight;
            Count += equivalence class weight;
        }
    }
    Mean = Sum / Count;
```

In Hold'em, the two hole cards account for the entire difference between the hands of the players, so attacking the variance of hole card distribution is important. An interesting aspect of Hold'em is that a program can reweigh the classes based on the hole cards of either player. In my actual implementation, I apply class weighting to both positions, and then average the means.

In the Hold'em experiment, class weighting had essentially the same result as Monte Carlo. We might have expected a reduction, but this is actually not a bad result. One of the problems with class weighting is that scaling increases the variance of classes that have a relatively small sample size. Accordingly, class weighting "gives back" some of its variance reduction.

Despite the outcome of this experiment, class weighting is a valuable technique because it allows us to estimate an unbiased mean even when trials have an imbalanced distribution. That capability is important to other selective sampling techniques.

5.3 Enforce Uniformity

When a program has a strong model of the distribution of continuations, then it can sample in a way that strictly matches the known distribution. We call this enforcing uniformity. Below we provide two examples of enforcing uniformity, viz. in Scrabble (5.3.1) and in Hold'em (5.3.2).

5.3.1 Tile Distribution in Scrabble

In Scrabble, the opponent's rack can be assumed to come from a uniform distribution.[3] Maven draws the first rack randomly, and for subsequent racks it begins each rack by placing the one tile that has been most underrepresented in the sampling this far. This policy very strongly pushes the sample towards the desired distribution, but does not bias the outcome.

Table 1 provides some data that illustrates the benefits. It is taken from a MAVEN simulation that contained 2718 continuations. The table shows that blanks were expected to occur in 7.53% of all racks, and actually occurred in 7.47% of all racks, a difference of only 0.06%. For comparison purposes, if the same experiment were conducted using a pure Monte-Carlo approach with no controls on sampling, then we would have a standard deviation of 0.51%. If you look down the column labeled Difference, it is clear that the sample has much greater conformance to the expected distribution than a Monte-Carlo sample would.

[3] Alternatively, you can draw inferences from the opponent's recent moves and bias the distribution of tiles accordingly. This makes implementation more difficult, but does not change the essence of the idea.

The benefit of this policy is to eliminate biases that relate moves to continuations. In the case of Scrabble, an illustrative example is the following: if the sample included more Qs than expected then it would reduce the evaluation of any move that pays off to an opponent's Q.

Table 1. Data on tile distribution

Tile	Expected	Actual	Difference
?	7.53%	7.47%	0.06%
A	67.74%	67.92%	0.18%
B	7.53%	7.65%	0.13%
C	15.05%	14.94%	0.12%
D	30.11%	30.10%	0.01%
E	67.74%	67.73%	0.01%
F	15.05%	14.97%	0.08%
G	22.58%	22.59%	0.01%
H	15.05%	14.97%	0.08%
I	67.74%	67.59%	0.16%
J	7.53%	7.54%	0.02%
K	7.53%	7.51%	0.02%
L	30.11%	30.06%	0.05%
M	15.05%	14.97%	0.08%
N	45.16%	45.03%	0.13%
O	52.69%	52.61%	0.08%
P	15.05%	15.38%	0.33%
Q	7.53%	7.62%	0.09%
R	37.63%	37.93%	0.30%
S	30.11%	30.02%	0.09%
T	45.16%	45.14%	0.02%
U	30.11%	29.99%	0.12%
V	15.05%	14.94%	0.12%
W	15.05%	15.27%	0.21%
X	7.53%	7.58%	0.05%
Y	15.05%	14.97%	0.08%
Z	7.53%	7.51%	0.02%

5.3.2 Hole Cards in Hold'em

When a program has control over the distribution of trials, it is natural to ensure that every equivalence class occurs with the proper frequency. An experiment that samples every equivalence class at the predicted frequency is called a *stratified sample*. The technique is used in public-opinion polling, for example. This goes beyond class weighting, which merely scales results. Stratified sampling makes certain that every class is appropriately represented, which avoids the drawback of class weighting while achieving the same result.

The application to Hold'em has a few technical fine points that illustrate how to adapt a technique to a domain. It is natural to arrange the 1326 distinct hole cards in a sequence that will repeat as long as there are trials, but we can do slightly better. Note that the weights of the equivalence classes are 6, 4, and 12, which are all even numbers. It follows that a cycle of length 1326 / 2 = 663 suffices.

A good ordering for the elements should satisfy certain natural goals. First, the initial 169 elements should be distinct equivalence classes, so that every equivalence class is sampled at least once as soon as possible. Second, we should distribute samples such there is a balance of strong and weak hands. It would not do to process the hands in decreasing order of hand strength.

We constructed our sample by first sorting the 169 classes in order of hand strength, and then performing a single shuffle that interleaved strong and weak hands. The same sequence is used at the end of the 663 hands. The middle of the sequence was constructed by taking 1 instance of each pocket pair, and 4 instances of each non-suited non-pair, then sorting and interleaving as before. The result is that any short segment contains a balance of strong and weak hands, including pairs, suited and non-suited hands. Additionally, each equivalence class occurs within the first 169 trials.

Any time we run through the complete sequence we have an unbiased sample of the full distribution. If an experiment stops partway through a cycle, then some equivalence classes will have more than their share of trials, but in large runs we can ignore such biases. Or more simply, make the number of trials a multiple of 663.

In our implementation, only the hands of the first position are systematically sampled in this way. The hands of the second position are randomly sampled. It is possible to sample both hands systematically, but this would require cycling through a much larger and more complicated space of equivalence classes.

The standard deviation of this procedure is 0.047 small bets, versus Monte Carlo's 0.251. This is a spectacularly good result, the equivalent of a 28-fold increase in speed.

5.4 Focus Attention on Differences

Thus far we have focused on methods of controlling the variance. Now we shift attention to methods of emphasizing differences. The basic idea is to notice that two moves do not differ after certain continuations, but do differ after others. If you can characterize the continuations then you can emphasize those that matter.

In principle, you can make such discriminations experimentally. Taking the case of Hold'em poker again, we might notice that when the opponent holds Ace-Ace then he always wins. In such a case the difference between the Bet and Call options might be constant, or might have at any rate a low variance. In an analogous position, a suited 10-9 might have a huge variance because of straight and flush draws. Rather than continuing to sample the Ace-Ace continuation, it is better to sample the high-variance continuation.

In practice, you must sufficiently sample for patterns to emerge, so that the sampling does not become confused by a chance low-variance outcome. An epsilon-greedy policy may be able to avoid such problems. Normally, if a program emphasizes continuations then it risks biasing the simulation. To do this properly the program may have to adjust for non-uniformity in the sample. Below we provide some considerations why targeting high-variance samples in poker is to be advised.

5.4.1 Targeting High-Variance Samples in Poker

A different approach to reducing variance is to select the hole cards so as to target equivalence classes that contribute most to the variance of the measurement. The method is to simulate the equivalence class that leads to the greatest expected reduction in variance from one additional trial. A program can use various heuristics for deciding which equivalence class that is. The author suspects that a good heuristic is to differentiate the formula for variance with respect to the number of data points, and simulate the equivalence class that maximized the derivative.

In practice, this technique suffers from a "bootstrapping" problem: variance cannot be measured reliably using a small number of trials. In general, it is an easy matter to overcome this problem. For example, one can require that every equivalence class have at least $C \times N$ trials, where C is a small positive number.

In poker, the bootstrapping problem has a natural solution. One can systematically sample the hole cards for player 1 on even trials, and systematically sample the hole cards for player 2 on odd trials. On each trial, the hole cards for one player are systematically sampled, and the hole cards for the other player are randomly sampled. Random sampling of the hole cards for the other player guarantees a minimum level of coverage.

This technique has not been implemented and verified, so there are no results to report.

6 Time Control

At the start of a turn, a program can budget a number of iterations that it expects to complete. While the simulation is underway, the program will collect additional information, which may suggest altering the initial allocation. In this section we will consider three reasons for reducing search effort, viz. by obvious moves (in 6.1), by minimal differences (in 6.2), and by a search bound (in 6.3).

6.1 Obvious Moves

If one move stands out, then you can terminate the search. A 'difference of means' test will reveal when a move is forced. MAVEN employs a "six sigma" standard for permanently pruning moves, and if all moves are pruned then MAVEN declares that the decision is "obvious." The six-sigma standard was selected because numerical precision limitations prevent calculating one-tailed standard normal tests when sigma exceeds 6.

In practice, this terminates about 25 per cent of the simulations in a Scrabble game. As computers become faster, this fraction will rise. Each quadrupling of CPU speed is sufficient to halve the sigma of a simulation, so in a few years only genuinely close decisions will simulate for the full time limit.

6.2 Minimal Difference

It may also happen that two moves are so close together that it is impossible to distinguish them. This often happens in Scrabble. For example, if you can play BOARD or BROAD in the same spot and for the same score, then there might not be a hair's difference between them. Actually, in this case the variance of the difference will also be very small, so the search may terminate because of a difference of means test.

There may be other reasons for having minimal differences. It may be that the position transposes after two plies, so the future is the same, or it may be that both moves pay off to the same continuations.

6.3 Search Bound

Near the end of a simulation, the leading move may be uncatchable by any other move within the available number of iterations. That move may have a small advantage, but if the program has only a small number of continuations remaining to simulate, then the advantage may be insurmountable within the time limit.

7 Selecting a Sample Move

The inner loop of a simulation plays out a continuation after a move (see line 3c in Section 2). This gives us the opportunity to select both a move and a continuation. In general, the sequence of continuations that the simulation should pursue is fixed in advance. In this section we discuss a dynamic continuation selection (in 7.1), a dynamic move selection (in 7.2) and provide an example of dynamic move selection (in 7.3).

7.1 Dynamic Continuation Selection

We mention as a possibility that the continuations can be selected dynamically instead of following a fixed sequence. We do not know of any program that actually operates this way, and it is not immediately clear how to make this work. Many of our other ideas depend on maintaining comparability across continuations, which is difficult when the order of continuations varies across moves.

7.2 Dynamic Move Selection

Think of a simulation as having a fixed set of continuations 1, 2, 3, … C. We also have a fixed list of moves 1, 2, 3, … M. A simulation that runs to exhaustion needs M $\times$ C trials. But if we observe what happens in the early trials, then we can shift attention to moves that show the greatest chance of being better than the move that is currently ranked best. This heuristic focuses on moves that may have high potential of being best. In Scrabble, there are usually only a handful of good moves, and therefore the simulation runs in time that is linear in C + M.

Between each trial, the program should sort the moves according to the probability of being better than the move that is currently considered best. Let us introduce some notation to make this precise. Let the outcome of move M after continuation C be

E(M, C). Let the move ranked best be M = 0. Then one estimate of the difference between move M and move 0 is the mean of E(0, C) – E(M, C) over all continuations C for which we have outcomes for both move M and move 0. We can also compute the standard deviations of E(0, C) – E(M, C), and then rank the moves in increasing order of mean / sigma. Moves preferred by this metric are more likely to surpass the top-ranked move, so it makes sense to spend extra effort on them. MAVEN's simulations show that the vast majority of effort goes to only 2 to 3 moves, even when 100 moves are simulated.

Note that this trick works best when every continuation is feasible after every move. This is generally the case in games with hidden information, because the first move in such a game is to select hidden information for the opponent, and that selection cannot depend on our move.

Of course, we need some policies that guarantee that we make progress. Progress is guaranteed if search would ultimately force every move to be searched. For the tail-end moves, it is sufficient to use an epsilon-greedy strategy. The best move poses a different problem: the selection heuristic above never proposes that we spend any effort on the best move itself! Accordingly, MAVEN imposes a practical rule: the best move must always have at least as many trials as any move.

Thus, we should begin by simulating all of the moves for at least a few iterations apiece, so that the standard deviations of the differences E(0, C) – E(M, C) are defined.

Moreover, we need a special case to cover the possibility of two essentially identical moves, which could result in a standard deviation of zero, and then a division by zero. More generally, if two moves are sufficiently close, in the sense of having a very small standard deviation of E(0, C) – E(M, C) then it is possible for those two moves to freeze out the rest of the list (except for moves chosen by epsilon-greedy) if the two happen to be the first-ranked and second-ranked moves. MAVEN's approach is to regard such a move as "co-leaders." This means that (a) whenever MAVEN simulates one of these moves then it simulates the other, and (b) such moves are never considered to be second best.

7.3 Example of Dynamic Move Selection

Dynamic move selection is the most powerful method in this article, so an example is in order (see Table 2). The data was collected from a Scrabble game. MAVEN simulated 20 moves, using an evaluation function that estimated winning percentage.

At the end of the simulation, MAVEN had given the moves widely varying numbers of trials, as shown by the column labeled "Trials". In this position, three moves merited almost equal attention: GOOMBAH, HOMAGE, and GOMBO. Had all 20 moves been simulated for the maximum 2353 trials, there would have been 47060 trials. The actual simulation needed less than 1/5 as many: 9381 altogether. Of course, had all 20 moves been simulated for 2353 iterations then the simulation would have had a lower error rate. But only barely, imperceptibly lower, since all of the best moves received a full quota of trials.

The column labeled "Error" represents the amount of equity that was lost by terminating search. This is estimated from the standard normal scores at the point when simulation stopped. Note that the amounts are very small, totaling less than 1 game in

2000. That is, this simulation estimates that playing GOOMBAH results in a loss of equity of about 0.0005 compared to perfect play of this position. This loss arises because of statistical uncertainty concerning simulation results.

Moreover, we note that the values in the Error column are almost all approximately equal. The algorithm that distributes trials causes this. By selecting to simulate the move with the highest Error value, the algorithm drives down the highest member of this column. It keeps the simulation on a nearly optimal path to reduction in error.

There is one notable exception to the pattern. HOMAGE has an estimated Error that is 200 times greater than the other moves. How did that happen? At an earlier point in the simulation, HOMAGE appeared to be significantly worse than it does now, with an Error value in the same range as the other moves. At that point, HOMAGE went on a streak of good results that distinguished it from the other plays.[4] The large value of Error reflects those recent good results.

The simulation stopped with HOMAGE having 2248 trials compared to GOOMBAH's 2353. Given the good results that HOMAGE has achieved, it is probably the second-best move, if not the best. If simulation had continued, then HOMAGE would soon receive 2353 trials, at which point the simulation would continue by alternating trials to GOOMBAH and HOMAGE until the Error value of the lesser of the two dropped below 0.0000020, whereupon attention would go back to GOMBO and the other moves.

We note that every move has received at least 29 iterations. This simulation ran using 8 as the minimum number of trials. The fact that every move has received more than 8 trials suggests that the simulation should have admitted more moves. One possible dynamic strategy is to add new moves to the simula-

Table 2. Data for dynamic move selection

Word	P(Win)	Error	Trials
GOOMBAH	0.616	-	2353
HOMAGE	0.615	0.0004573	2248
GOMBO	0.611	0.0000020	2353
GABOON	0.609	0.0000020	713
BEHOOF	0.607	0.0000017	640
GENOA	0.594	0.0000018	185
BOGAN	0.592	0.0000019	29
BAH	0.590	0.0000012	131
BEGORAH	0.587	0.0000013	29
HOAGY	0.584	0.0000018	114
GOBAN	0.583	0.0000018	52
BOON	0.582	0.0000012	38
OHMAGE	0.581	0.0000012	38
GOBO	0.581	0.0000012	33
BEANO	0.581	0.0000019	33
GABOON	0.580	0.0000020	97
ABMHO	0.579	0.0000020	90
ABOON	0.577	0.0000013	58
BOOGY	0.577	0.0000019	71
OOH	0.57	0.0000018	76

[4] You can see the advantage of not pruning any moves. What if HOMAGE had been pruned before its hot streak? Dynamic move selection never permanently prunes a move. The six-sigma time control algorithm does prune moves, but only when the evidence is incontrovertible.

tion wherever every move in the simulation has, say, 20 iterations. Such a rule would dynamically scale the move list so as to use additional time to reduce the error rate at near optimal trajectories.

Finally, we note that the results of infrequently simulated moves are very uncertain. For example, OOH received only 76 trials. The standard deviation of the difference GOOMBAH – OOH is about 3.5 points. This is really poor accuracy. However, the mean of GOOMBAH – OOH is estimated as 12.1 points, which is 3.5 standard deviations. Accordingly, while we do not really know the value of OOH, we can be quite sure that GOOMBAH is a better play.

8 Evaluating the End Result

Simulated continuations eventually end, and then the end result should be evaluated. There are several approaches. Below we discuss the three most important ones, i.e., play to the end (in 8.1), heuristic evaluation (in 8.2), and compensation (in 8.3).

8.1 Play to the End

In the ideal case, a program can simulate continuations until it hits a terminal node. Then the evaluation is easy. For instance, in Scrabble, the program simply has to compare the points scored by both sides. But even here the program can have trouble. For example, in Go it can be difficult to know when the game is over. You can have seki, for example, in which the player that moves first will lose a group, so the proper play is to pass. Unfortunately, detecting seki is computationally expensive, and a better policy maybe to continue games until statically safe eyes are found. Playing to the end is the indicated method if (1) it is computationally feasible, and (2) if the game is guaranteed to end, and (3) a heuristic evaluation of middle-game positions is slower or inaccurate.

8.2 Heuristic Evaluation

Heuristic evaluation is often the best alternative, since playing to the end of the game is in most cases impractical. There is just one cautionary note. It is very important for heuristic evaluations to be comparable across all continuations considered by the program. This can be awkward in cases of early termination. For example, assume that a program has a heuristic evaluation function that measures progress towards a goal. The evaluation from one variation may be “need three more moves.” Then, another branch may terminate early, resulting in the evaluation “the program wins.” How do you average these outcomes? Below we provide an example of a heuristic evaluation for a non-deterministic strategy in Hold’em (see 8.2.1).

8.2.1 Accounting for Non-deterministic Strategy in Hold’em

The representation of the Hold’em strategy in terms of probability triples (Fold, Call, Bet) creates an attractive opportunity for the heuristic evaluation function. When a player decides to fold, the experimental trial is treated as though he folds 100 per cent of the time, whereas he actually folds only with some probability. In truth, the probabilities selected are quite often 1.0, that is, the strategies are pure. Still, there is a percentage of mixed strategy, and it would be a shame not to exploit that.

When a mixed strategy includes a non-zero probability of folding, we can credit the opponent with the pot with that probability, and then carry on the rest of the game (which may include further non-determinism). This trick applies only to two-player games, and thus it appears that it cannot be used in a normal 10-player Hold'em game. However, generalizations of the same idea are applicable. For instance, the trick can be used when all but two players have folded. Many pots in Hold'em eventually come down to two players, so the trick would apply to some degree even in a 10-player game. Another example involves simulations where you only wish to measure the equity of a single player. Such a situation arises when simulations are used to select betting strategies, as only the equity of the bettor matters [2].

Another refinement of the same idea is to sample non-deterministically a final call. When a strategy gives both Call and Raise as an option, and the Call strategy would end the game, then we can settle the game under the Call strategy, and continue the game under the Raise strategy.

The author's experimental poker environment requires significant reorganization before this complex experiment can be carried out. It seems particularly interesting in conjunction with other methods. Consider two engines that differ only in some rare aspect of play. Simulations would not normally be able to detect such differences, because they do not occur often enough to stand out above statistical noise. But assume that the simulation used duplicate trials, with duplicate random number generators. In that case, two engines would always agree except in those cases where they differ, and the simulation would correctly find no differences in those cases. If the simulations also sampled cases that have a higher variance, then the focus would go to regions of the search space where differences were found. Finally, if the simulation accounted for non-determinism, then in case the two engines differed we would see the real effect of those differences, even if the only change were a small shift in probabilities.

It may be that two very similar programs can be tested more directly. For instance, you could create a test bed of situations where they disagree on the strategy, and then measure the disagreements. But that only works if you can characterize the differences. In many situations, particularly where automatic tuning is involved, all we know is that the engines are similar. Having a combination of techniques that *automatically* identified differences would be a huge boost to the productivity of research.

8.3 Compensation

If you can determine whether a continuation is better or worse than average, then a remarkable opportunity may be available to the program. Backgammon researchers, who have the benefit of excellent evaluation functions, developed this method. The idea is to correct for the "luck of the draw" by adjusting the outcomes of continuations for the effects of fortune. Below I will elaborate on this issue.

Compensation is important enough that I will give four examples, viz. Backgammon rollouts (in 8.3.1), Backgammon races (in 8.3.2), Scrabble simulations (in 8.3.3), and reducing variance of uncontrolled sequences (in 8.3.4). The first example is the original idea, which I believe is due to Fredrik Dahl (JELLYFISH), but does not appear to be described in the literature. Other examples are from the author's research.

8.3.1 Backgammon Rollouts

Backgammon engines use the term 'variance reduction' for the concept called compensation. In a Backgammon rollout "with variance reduction" the engine's evaluation function (which is presumed to be unbiased) is used to compensate for the effect of luck. This is a very effective compensation factor. Compensation is most beneficial in situations where the evaluation function is well tuned, but it is highly beneficial in every case. Variance-reduced rollouts are frequently two to four times as fast as Monte-Carlo rollouts.

When a Backgammon engine chooses moves in rollouts using at least a 2-ply search, then they can use compensation. Backgammon engines often count plies of search in peculiar ways, so I should describe what I mean. A 2-ply search consists of (1) generating the legal moves for the roll that a player has to play, and (2) evaluating each alternative by generating all 21 different rolls for the opponent and by averaging a static evaluation of the best opponent's reply for each roll. This accounting of search depth is consistent with conventions used in other games, though I have seen it called "1-ply" in Backgammon writing. Notice that when a Backgammon engine uses a 2-ply search, it generates an evaluation of each possible opponent's roll. This is very useful, because the next thing that happens in the rollout is that the opponent will randomly select a roll. It follows that a 2-ply (or deeper) search generates all of the information needed to estimate the extent to which the opponent's roll exceeded expectations. The sequence of amounts by which rolls exceed expectations can be summed and used to offset the outcome of the game.

8.3.2 Backgammon Races

Backgammon evaluation functions are very accurate, which is enormously helpful when trying to correct for the effects of luck. But it is not necessary to be tremendously accurate.

The basic idea is to offset the luck of the dice. That is, instead of computing the sum of the trials, we compute the sum of the trials and subtract a value that represents the quality of the rolls. We will call this term a *compensator*. A compensator will reduce variance if it satisfies the following three conditions.

1. It has mean zero when averaged over all possible continuations. This is required so that subtracting the compensator does not bias the mean.
2. It correlates well to the quality of the draw. To ensure that it can compensate for the luck of the draw.
3. It has roughly the same magnitude as wins and losses. This requirement ensures that variance will be cancelled, rather than be magnified.

A typical compensator will evaluate all possible outcomes at the point of a random choice, and determine a distribution of outcomes at that point. The actual outcome is offset by its evaluation within the list. The author believes that this is the first publication that states the sufficiency of these conditions. In this section I will present an example that shows that a program does not need to have a particularly good evaluation function in order to reduce variance. The domain is racing positions in Backgammon.

The occurrence of 6-6 or 5-5 at any time in a race is a huge swing. A program can control the frequency of such rolls early in the rollout, but beyond the first few ply there are not enough trials to fill a random distribution. Nevertheless, the program can compensate for the impact of the dice on races.

Here is how it works: a program has an evaluation function that roughly measures the effect of any roll on the race. The evaluation does not have to be sophisticated, as long as it is unbiased. That is, the program simply needs the expected sum of the evaluation to be zero and the evaluation to be positively correlated with the effect of the roll. The program might take the evaluation to be C $\times$ (pip-count – 8.166666), where C is a small positive parameter, and the magic number 8.166666 is chosen to make the evaluation unbiased. Every time a roll occurs, *subtract* this evaluation from the outcome of the simulated continuation. Big rolls will help a player to win the race, but they will also be offsets against the outcome.

The program can experimentally choose the parameter C to maximize the stability of rollouts. Alternatively, based on the expert rule of thumb that having a lead equal to 10 per cent of your pip count is equivalent to a 75 per-cent chance of victory, it is possible to choose C as a function of the lead in the game, which should achieve nearly the same degree of variance reduction as you would achieve using a neural-network evaluation function, but with much better speed.

8.3.3 Scrabble Simulations

In Scrabble, the only tiles that a program can force into a known distribution are the opponent's tiles. The tiles that the side-to-move will draw after its turn are a function of the tiles that the opponent receives, so they cannot be controlled independently. Even though those tiles cannot be forced to conform to a fixed distribution, they can be controlled by compensation.

MAVEN contains a table that represents the first-order impact of a tile on scoring. It contains facts such as "Holding an S increases the score by 7.75 points." Such first-order values were estimated by simulations, and the values are widely applicable. We can use these estimates to adjust the scores of variations, by offsetting the point differentials of continuations by subtracting the first-order value of the tiles drawn by both sides.

The effect of this adjustment is fairly minor for the opponent's tiles, since the tiles are controlled through selective sampling and duplicate continuations. But for the future racks of the side to move, or if simulations extend beyond two ply of lookahead, then such adjustments can reduce the variance by a substantial amount. My data is somewhat speculative, since I have not implemented this feature in MAVEN and I therefore have no direct measurements. I have indirect data from counting the tiles in a small sample of human expert games. That data suggests that the first-order quality of one's tiles explains at least 70 per cent of the variance in scoring. Accordingly, for long-range simulations that encompass many turns, I would expect a substantial reduction in variance by this adjustment. For the short-range, 2-ply simulations that MAVEN actually uses to select moves, I expect the effect to be small.

8.3.4 Reducing Variance of Uncontrolled Sequences

All of the techniques described thus far reduce variance by controlling the cards or randomness of the simulation. Such techniques are not always available. One important example is when measuring the quality of a program's play in games against Internet players. In such cases, the server distributes cards, and other players will not allow a person to control how it is done! Billings, Davidson, Schaeffer, and Szafron [4] remarked: "Since no variance reduction methods are available for online games, we generally test new algorithms for a minimum of 20,000 hands before interpreting the results." However, while it is true that a person or program cannot control the cards, there are still techniques for reducing variance.

First, one can apply class weighting. This simple technique attacks the variance caused by unequal distributions of hole cards. The drawback of class weighting (loss of information) is lessened when a lot of data is available.

But the real opportunity is compensation. In poker games, a suitable compensator is all-in equity. All-in equity is calculated by assuming that all current players will call for the rest of the game. It is easy to verify that all-in equity satisfies the conditions given above. Any expert computer program typically computes all-in equity, so there is no run-time implication of this technique.

In my implementation, each trial is compensated by the all-in equity of the hole cards only. Note that this technique is redundant when using duplicate trials, because the compensator would cancel across trials. No compensation is made for the community cards. Results should be better if compensation applies throughout the deal.

Applying this technique to a Monte-Carlo simulation produces a reduction in variance from 0.251 to 0.153. When this compensator is used in conjunction with duplicate continuations and an enforced uniform distribution of hole cards, the variance is reduced to 0.030, equivalent to a 70-fold speedup.

9 Summary

Selective search control is an effective tool in many domains of current interest. In suitable domains, selective simulation uses a large amount of CPU power to leverage a small amount of skill into a large amount of skill. The trick is to make the CPU burden manageable.

The methods described in this paper can go a long way towards reducing the CPU cost. The author suspects that some of these ideas are so obvious that they have been discovered many times, yet they do not appear to be described in the literature. It is our hope that by gathering these ideas into one source, we may eliminate the need to rediscover them in the future.

References

1. Anantharaman, T. (1997). Evaluation Tuning for Computer Chess: Linear Discriminant Methods. *ICCA Journal*, Vol. 20, No. 4, pp. 224-242.
2. Billings, D., Papp, D., Schaeffer, J., and Szafron D. (1998). Poker as an Experimental Testbed for Artificial Intelligence Research. *Proceedings of AI '98, (Canadian Society for Computational Studies in Intelligence).*
3. Billings, D., Pena, L., Schaeffer, J., and Szafron, D. (1999). Using Probabilistic Knowledge and Simulation to Play Poker. *AAAI National Conference*, pp 697-703.

4. Billings, D., Davidson, A., Schaeffer, J., and Szafron, D. (2002). The Challenge of Poker. *Artificial Intelligence*, Vol. 134, Nos. 1-2, pp. 201-240. ISSN 0004-3702.
5. Bouzy, B. (2003). Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. *Joint Conference on Information Sciences*, pp. 505-508. ISBN 0-9707890-2-5.
6. Bouzy, B. and Helmstetter, B. (2003). Monte-Carlo Go Developments. *Advances in Computer Games. Many Games, Many Challenges* (*ACG-10 Proceedings*) (eds. H.J. van den Herik, H. Iida, and E.A. Heinz), pp. 159-174. Kluwer Academic Publishers, Boston, MA. ISBN 1-4020-7709-2.
7. Brügmann, B. (1993). Monte Carlo Go. Available by FTP from ftp://ftp.cgl.ucsf.edu/pub/pett/go/ladder/mcgo.ps
8. Frank, I., Basin, D., and Matsubara, H., (1998). *Monte-Carlo Sampling in Games with Imperfect Information: Empirical Investigation and Analysis*. Research report of the Complex Games Lab, Ibaraki, Japan.
9. Galperin, G., and Viola, P. (1998). *Machine Learning for Prediction and Control*. Research Report of the Learning & Vision Group, Artificial Intelligence Laboratory, MIT, Cambridge, MA.
10. Ginsberg, M. (1999). GIB: Steps toward an Expert-Level Bridge-Playing Program. *Proc. IJCAI-99*, Stockholm, Sweden, pp. 584-589.
11. Gordon, S. (1994). A Comparison Between Probabilistic Search and Weighted Heuristics in a Game with Incomplete Information. AAAI Fall 1993 Symposium on Games: Playing and Learning, *AAAI Technical Press Report FS9302*, Menlo Park, CA.
12. Sheppard, B. (2002). *Towards Perfect Play of Scrabble*. Ph.D. Thesis, Universiteit Maastricht. Universitaire Pers Maastricht, Maastricht, The Netherlands. ISBN 90 5278 351 9.
13. Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Communications of the ACM,* March 1995, Vol. 38, No. 3. ISSN 0001-0782.