SOLVING PROBABILISTIC COMBINATORIAL GAMES

Ling Zhao[1] *and Martin Müller*[2]

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada, T6E 2E8

ABSTRACT

Probabilistic combinatorial games (PCG) are a model for Go-like games recently introduced by Ken Chen. They differ from normal combinatorial games since terminal position in each subgame are evaluated by a probability distribution. The distribution expresses the uncertainty in the local evaluation. This paper focuses on the analysis and solution methods for a special case, 1-level binary PCG. Monte-Carlo analysis is used for move ordering in an exact solver, that can compute the winning probability of a PCG efficiently. Monte-Carlo interior evaluation is used in a heuristic player. Experimental results show that both types of Monte-Carlo methods work very well in this problem.

1. INTRODUCTION

Heuristic position evaluation in game playing programs should be a measure of the probability of winning. However, in point-scoring games such as Go or amazons, evaluation functions usually approximate the score of the game instead of the winning probability. This can lead to serious blunders when programs make a risky moves to gain even more territory, instead of playing conservatively to preserve a comfortable lead (Müller, 1995; Chen, 2005).

One approach to dealing with uncertainty of evaluation is to use partial ordered values such as probability distributions instead of scalar evaluation. For the case of mini-max search such approaches were developed in (Baum and Smith, 1997; Müller, 2001). (Chen, 2005) extends this approach to the case of combinatorial sums of independent games (Berlekamp, Conway, and Guy, 1982), and defines the framework of *probabilistic combinatorial games* (PCG). Figure 1 shows an idealized example of how PCG might be applied to Go. A board is split into three independent areas, and the result of local search in each area is represented by a PCG. Terminal nodes in the local search are evaluated by a probability distribution.

(Chen, 2005) defined the PCG model and gave a high-level solution algorithm. This paper presents first computational results. The main contributions are:

- Efficient exact and heuristic solvers for the special case of sums of 1-level binary PCG's, called *SPCG*.
- An analysis of the search space of SPCG.
- A Monte-Carlo move ordering technique for the exact SPCG solver.
- A Monte-Carlo based heuristic evaluation technique for SPCG with performance close to that of the optimal player.
- An extensive experimental evaluation of the two solvers.

[1] zhao@cs.ualberta.ca
[2] mmueller@cs.ualberta.ca

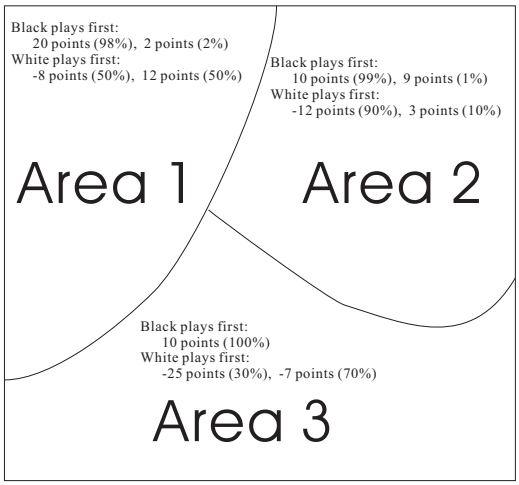

Figure 1: PCG example: sum of three subgames

The remaining part of the paper is organized as follows. In Section 2, we introduce definitions and notation of SPCG, and raise the main problems addressed in this paper. Section 3 analyzes game trees in SPCGs, including the methods to evaluate terminal nodes and interior nodes. Section 4 presents our approaches to solve the game, and to play the game strongly if solving it is not feasible. Section 5 presents detailed experimental results and analysis, and we conclude with future work in Section 6.

2. DEFINITIONS AND PROBLEM IDENTIFICATION

The following definition of probabilistic combinatorial games (PCG) is quoted from (Chen, 2005) with slight modification:

1. A terminal position, represented by a probability distribution $d = [(p_1, v_1), (p_2, v_2), \cdots, (p_n, v_n)]$, is a PCG. In d, the outcome is v_i with a probability of p_i ($1 \leq i \leq n$). $0 \leq p_i \leq 1$ and $\sum_{i=1}^{n} p_i = 1$.
2. $\{A_1, A_2, \cdots, A_n \mid B_1, B_2, \cdots, B_n\}$ is a PCG if $A_1, A_2, \cdots, A_n$ and $B_1, B_2, \cdots, B_n$ are all PCGs. $A_1, A_2, \cdots, A_n$ are called left options and $B_1, B_2, \cdots, B_n$ are called right options, the same as in the normal combinatorial games.
3. A sum of PCGs is a PCG. Summation is understood in the sense of combinatorial game theory (Berlekamp *et al.*, 1982): a move in a sum game consists of a move in exactly one subgame and leaves all other subgames unchanged.

PCG are played as follows:

- A move can be played in any nonterminal subgame.
- If all subgames are terminal, the game itself is terminal, and its probability distribution is the sum of the distributions of all subgames.
- A game is won for Left if the value is greater than 0, and won for Right otherwise. From a terminal game, the probability of winning is computing by adding the probabilities of all values greater than 0.

The PCG model contains combinatorial games as the special case where all probability distributions in terminal positions have a single point distribution of the form $d = [(1, v)]$.

This paper focuses on a different special case of PCG which models uncertainty but keeps the combinatorial games as simple as possible. Simple PCG (SPCG) are PCG which obey the following constraints:

1. A SPCG consists of $n \geq 1$ subgames.
2. Each subgame has exactly one option for Left and one option for Right.
3. Each option immediately leads to a terminal position represented by a probability distribution.
4. Each distribution d in a terminal position has only two values with associated probabilities: $d = [(p_1, v_1), (p_2, v_2)]$.

A SPCG of n subgames is of the form $\sum_{i=1}^{n}\{d_i^L \mid d_i^R\}$, where d_i^L and d_i^R are 2-valued probability distributions.

In SPCG, a move can be simply indicated by the number of the subgame. In the experiments and analysis in this paper, without loss of generality, Left always moves first. This paper focuses on two basic problems concerning SPCGs:

1. Efficient exact solution: What is the winning probability of Left in a given SPCG?
2. Approximate solution: When solving a SPCG is too slow, how to play as well as possible under time constraints?

3. GAME TREE ANALYSIS OF SPCG

The SPCG model presents many regularities in terms of its game trees. Suppose a game G has n subgames. The first move has n choices (n subgames to select from), and after that, the second move, which will be done by the other player, has $n-1$ choices (since one subgame is no longer available). So in general the kth move has $n-k$ choices. If the root of a game tree is defined as a node at depth 0, then any node at depth k has exact $n-k$ children, and there are $n!/(n-k)!$ nodes in total at depth k. Therefore, the total number of nodes in the game tree is $\sum_{k=0}^{n} n!/(n-k)! = \sum_{k=0}^{n} n!/k!$, and there are $n!$ terminal nodes.

Since the subgames are independent, the order of play of the subgames chosen by the same players does not change the evaluation. Each position can be represented by three sets: The *Left set (Right set)* contains the indices of all subgames played by Left (Right), and the *open set* contains the indices of all subgames that have not been played yet. For example, the sequence of subgames chosen (1, 2, 3) is equivalent to the sequence of (3, 2, 1). if $n = 5$, then the Left set after both these sequences would be $1, 3$, the Right set 2, and the open set $4, 5$.

For a position after k moves (a node at depth k in a game tree), k subgames have been played by the two players, $\lceil k/2 \rceil$ by Left and $\lfloor k/2 \rfloor$ by Right. The number of distinct nodes at depth k is equal to the number of possible combinations of left and right sets, $\binom{n}{k}\binom{k}{\lfloor k/2 \rfloor}$. There are $\binom{n}{\lfloor n/2 \rfloor}$ distinct terminal nodes, and the total number of distinct nodes in the game tree is: $\sum_{k=0}^{n}(\binom{n}{k}\binom{k}{\lfloor k/2 \rfloor})$.

As an example, Table 1 lists the number of nodes for SPCG with 12 to 15 subgames. The total number of distinct nodes in a game increases roughly by a factor of 3 per subgame. The number of distinct terminal nodes increases by a factor close to 2 (exactly 2 if n is odd, $2 - \frac{2}{n+2}$ if n is even). There is a very large number of transpositions in the game tree, and the number of distinct interior nodes is significantly more than that of distinct terminal nodes.

3.1 Terminal Node Evaluation

For a SPCG with n subgames, each terminal node T in its game tree is a sum of n probability distributions.

# subgames	12	13	14	15
# all nodes	1,302,061,345	16,926,797,486	236,975,164,805	3,554,627,472,076
# terminal nodes	479,001,600	6,227,020,800	87,178,291,200	1,307,674,368,000
# all distinct nodes	143,365	414,584	1,201,917	3,492,117
# distinct terminal nodes	924	1,716	3,432	6,435

Table 1: Statistics for SPCG with 12 to 15 subgames.

$$T = \sum_{i=1}^{n} [(p_{1_i}, v_{1_i}), (p_{2_i}, v_{2_i})]$$

T itself can be expressed as a single, complex probability distribution over sums of values v_{1_i} and v_{2_i}. The winning probability of T for Left is the sum of all probabilities of values greater than 0 in T.

The following formula expresses the winning probability P_w of T. Let $q(i) \in \{1, 2\}$ $(1 \leq i \leq n)$ be such that the result value $V_q(i)$ is chosen at subgame i. Then

$$P_w = \sum \{p_{q(1)_1} p_{q(2)_2} \cdots p_{q(n)_n} | \sum_{i=1}^{n} v_{q(i)} > 0\}, \forall 2^n \text{ combinations of } q(1), q(2), \cdots, q(n). \quad (1)$$

There are at least three methods to evaluate P_w.

1. Direct evaluation of all 2^n combinations above.
2. A dynamic programming algorithm, representing the distribution as a list of bins and adding one distribution at a time. This is effective when the values v are all integers within a small range. However, in the worst case (real numbers or large integers) the final distribution can have up to 2^n distinct non-zero entries, and this algorithm does not improve on the direct one.
3. Using the fact that the probability density of a sum is the convolution of their probability density functions.

In the experiments reported in this paper, for the range of n tested, methods 1. and 2. perform similarly well. Method 2. can be further improved by noting that the final result is only P_w, not the exact distribution. All partial result that ensures a win or loss can be bagged and removed from further processing. This improves the efficiency. In experiments with $n = 14$ subgames, optimized dynamic programming was about twice as fast as the direct approach for a range of values $v \in [-1000, 1000]$ but twice as slow for a larger range $v \in [-5000, 5000]$. Since dynamic programming version is efficient only when the range is limited, method 1. was used for all further experiments and discussion in this paper.

It is costly or even infeasible to compute the exact winning probability of a terminal node when n is large. In such a situation, it is desirable to have a good approximation method with controllable statistical errors. In cases when a quick estimate is sufficient, approximation can improve the performance of a SPCG solver or player.

Monte-Carlo sampling is used to approximate winning probabilities. For each distribution d, a value is generated. For example, if $d = [(p_1, v_1), (p_2, v_2)]$, then v_1 is generated with probability p_1 and v_2 with probability $p_2 = 1 - p_1$. In a SPCG terminal node T, n values are generated from the n distributions, and the sum of these n values is the result of the sample. If it is greater than 0, it counts as a win for player Left, and a loss otherwise. The fraction of wins P_w^k from k independent samples is used to approximate the winning probability in T.

Each sample has a probability of P_w to win, and $1 - P_w$ to lose. The mean of the distribution is P_w, and the standard deviation is $\sqrt{P_w(1 - P_w)}$. According to the Central Limit Theorem, the

mean of random samples drawn from a distribution tends to have a normal distribution. When k is sufficiently large, P_w^k has a normal distribution with the mean of P_w, and the standard deviation of $\sqrt{\frac{P_w(1-P_w)}{k}} \leq \frac{1}{2\sqrt{k}}$. This inequality can be used to select the minimum value of k for a given required accuracy. For example, according to the normal distribution, almost for sure (99.7%) the difference between P_w and P_w^k is no more than $\frac{3}{2\sqrt{k}}$. With 10000 samples, the difference is 99.7% likely to be within 0.015.

Experiments shown in Table 2 list the difference between P_w and P_w^k in random terminal nodes, and compare it with the theoretical bounds. For games consisting of n subgames, 100 terminal nodes are randomly chosen as test cases. For each terminal node, probabilities were uniformly generated from between 0% and 100% with granularities of 1% and 0.001%. Values were uniformly generated between -1000 to 1000 with a granularity of 1.

For each terminal node, the error estimate, which is computed by $P_w - P_w^k$, is recorded for k from 3 to 33333. The standard deviation of the estimated error from the 100 terminal nodes is compared with its theoretical bound ($\frac{1}{2\sqrt{k}}$) in Table 2 for n is 14, 17, and 20 respectively.

# samples	3	10	33	100	333	1000	3333	10000	33333
$1/2\sqrt{k}$	0.2887	0.1581	0.0870	0.0500	0.0274	0.0158	0.0087	0.0050	0.0027
14 subgames	0.1768	0.1118	0.0488	0.0320	0.0189	0.0118	0.0070	0.0050	0.0042
17 subgames	0.2294	0.1176	0.0717	0.0348	0.0203	0.0121	0.0074	0.0059	0.0052
20 subgames	0.2427	0.1020	0.0580	0.0314	0.0246	0.0121	0.0077	0.0059	0.0045

# samples	3	10	33	100	333	1000	3333	10000	33333
$1/2\sqrt{k}$	0.2887	0.1581	0.0870	0.0500	0.0274	0.0158	0.0087	0.0050	0.0027
14 subgames	0.1893	0.0977	0.0655	0.0322	0.0190	0.0107	0.0053	0.0034	0.0019
17 subgames	0.1923	0.1093	0.0619	0.0419	0.0206	0.0125	0.0064	0.0038	0.0019
20 subgames	0.2091	0.1154	0.0634	0.0329	0.0199	0.0130	0.0064	0.0034	0.0019

Table 2: Standard deviation with probability granularity of 1% and 0.001%

The experimental results match the prediction very well when there is a fine granularity for randomly generated probability, but the model is not appropriate for coarse granularity.

The term *Monte-Carlo terminal evaluation* (MCTE) is used to refer to the Monte-Carlo method discussed in this subsection. MCTE estimates the winning probability of a terminal node.

3.2 Monte-Carlo Sampling for Heuristic Interior Node Evaluation

Left's winning probability at an interior node, which is represented by its Left, Right and open sets, can be computed by a complete mini-max search, using the exact evaluation of Section 3.1 in all terminal nodes of the search. However, when full search is too slow, Monte-Carlo sampling is useful for improving heuristic evaluation as well. Such methods are popular in games with incomplete information such as Poker (Billings *et al.*, 2002), and also in games with complete information such as Go (Bouzy and Helmstetter, 2003). Abramson's expected-outcome evaluation (Abramson, 1990) evaluates a node in a search tree by averaging the values of terminal nodes reached from it through random play. Similar ideas can only be found in (Palay, 1985; Baum and Smith, 1997). This method is adapted here.

From an interior node, the sequence of alternate moves by both players to reach a terminal node is simulated by randomly choosing each move of the sequence among all its legal choices with equal probability. Such a simulation is iterated k times, and the average winning probability of the k sampled terminal nodes is an approximation of the winning probability of the interior node. The winning probability at terminal nodes can be either accurately computed by using Formula (1) in Section 3.1, or approximated by methods such as MCTE.

In SPCG, the order of moves chosen by the same players is irrelevant. For efficiency, a move se-

quence simulation can be replaced by uniformly randomly selecting k nodes out of all descendant terminal nodes below this interior node. Such sampling is only meaningful if k is (much) smaller than the total number of descendant terminal nodes of an interior node.

Experiments measure the difference between P_w and P'_w. For an interior node, its exact game value P_w and the approximate value P'_w estimated by the average winning probability of all its descendant terminal nodes are computed. The test set consists of starting positions from 100 randomly generated games with $n = 14$ subgames. The mean of $|P_w - P'_w|$ is 0.148 in the experiments. Since the difference is large, it is implausible to approximate P_w using P'_w. However, for a set of interior nodes at the same depth, their errors are highly correlated, and the relative order of their P'_w values is a very good approximation of the order of their P_w values. Section 5.4 gives detailed results. P'_w is an excellent move ordering heuristic.

In contrast to MCTE, *Monte-Carlo interior evaluation* (MCIE) is used to refer to the Monte-Carlo sampling discussed in this subsection. MCIE estimates the winning probabilities at an interior node. *Monte-Carlo move ordering* uses MCIE for move ordering at interior nodes in a search.

Monte-Carlo sampling is used in both the Monte-Carlo terminal evaluation and the interior evaluation, but they serve different purposes, and have different parameters to control their qualities. These two types of evaluation are used in both an exact SPCG solver and a heuristic SPCG player, which will be discussed in the next two sections.

4. EXACT SOLVER AND HEURISTIC PLAYER FOR SPCG

In order to compute the exact winning probability of a game, a complete solver was implemented based on alpha-beta mini-max search. Standard enhancements including transposition tables and move ordering using the history heuristic (Schaeffer, 1989) are used.

The optimized solver spends more than 90% of its time on the accurate evaluation of terminal nodes Since a SPCG game tree has far more interior than terminal nodes, most terminal nodes must be evaluated in order to solve the game. Unless a more efficient method can be found to evaluate terminal nodes, it is difficult to further improve the overall performance of the solver.

For cases when it is infeasible to solve a SPCG, or when it is desirable to play the game fast with reasonable strength, a heuristic player was designed as follows:

A fixed depth alpha-beta search is performed. Nonterminal frontier nodes of the search are evaluated using MCIE. Within MCIE, terminal nodes are evaluated by MCTE for efficiency.

A heuristic Monte-Carlo player with search depth of 1 is the same as the expected-outcome player in (Abramson, 1990). It performed well in experiments. This player chooses its move as follows: it finds all legal moves from the starting position, generates a depth 1 interior node for each move, and compares these nodes using MCIE. The move that leads to the highest-valued depth 1 node is chosen. This simple technique performs very well in SPCG. An advantage of this 1-ply Monte-Carlo sampling method is that it is friendly to time control. Each sampling process costs almost the same amount of time, and it can be performed continuously until time runs out.

Experimental results for the solver and heuristic player are given in the next section.

5. EXPERIMENTAL RESULTS AND ANALYSIS

This section summarizes experimental results for different configurations of the SPCG solver and the Monte-Carlo SPCG player, and measures how Monte-Carlo move ordering performs in games.

5.1 Experimental Setup

All experiments were run on Linux workstations with AMD 2400MHz CPUs. The compiler was gcc 3.4.2. The transposition table has 2^{20} entries, using about 34MB memory. Since the evaluation of terminal nodes is expensive, they can never be overwritten by interior nodes in the transposition table.

A set of 100 randomly generated games with $n = 14$ subgames are used for testing. Probabilities were uniformly generated between 0% and 100% with a granularity of 1% and values between -1000 to 1000 with a granularity of 1. Table 3 contains statistics for solving these instances. Each cell is of the form: mean value $\pm$ standard deviation. The first player has a big advantage in SPCG, so the average winning probability much larger than 50%.

Solving time (sec)	Total nodes visited	Terminal nodes visited	Cache hits	Winning prob.
8.2 ± 0.8	$(2.2 \pm 1.2) \times 10^5$	$(2.5 \pm 0.2) \times 10^3$	$(1.3 \pm 0.8) \times 10^4$	$70.0\% \pm 25.8\%$

Table 3: Statistics of solving the 100 random games

5.2 Performance of the Exact Solver

Monte-Carlo move ordering has two important parameters: the percentage of the total number of descendant terminal nodes to be sampled, n_t (see Section 3.2), and the number of value combinations to be sampled for approximate evaluation of each terminal node, n_c (see Section 3.1). The Monte-Carlo move ordering uses MCIE, which contains two levels of Monte-Carlo sampling: on the top level, a number of an interior node's descendant terminal nodes are randomly chosen. On the bottom level, each of these terminal nodes is evaluated by the average result of a number of value combinations sampled from the value distributions of the node. Finally, the evaluation of the interior node is the average evaluation of those terminal nodes sampled.

The solver has a depth limit for Monte-Carlo move ordering (d_m). Nodes at depth up to d_m use MCIE for move ordering, and others use the history heuristic. The total time for solving the 100 test games for different parameter combinations is given in Figure 2.

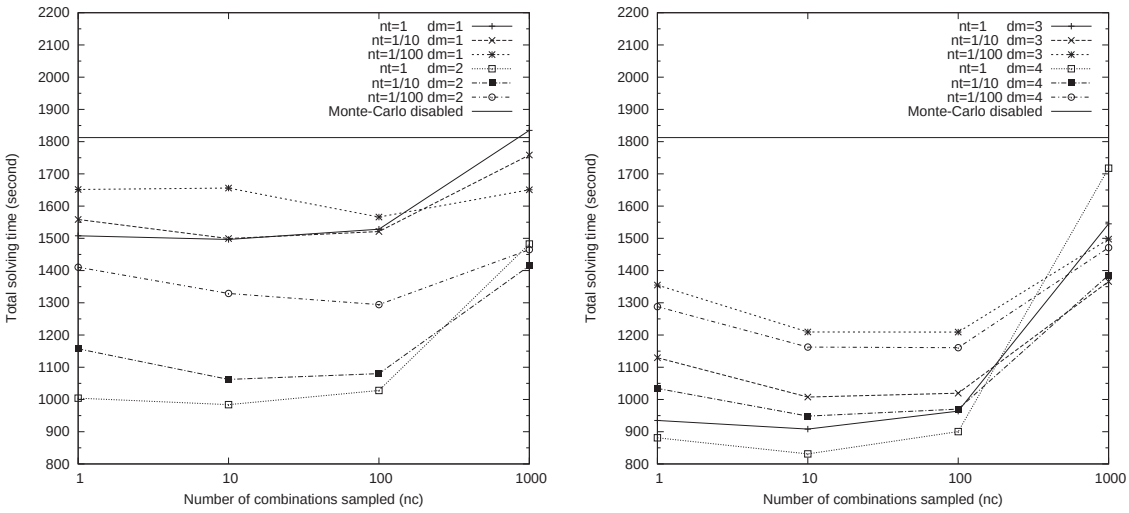

Figure 2: Solving time with respect to parameters in Monte-Carlo move ordering

In the figure, the straight lines parallel to x-axis denote the solving time when Monte-Carlo move ordering is disabled, i.e., the solver only uses the history heuristic for move ordering. The large difference between these lines and the best solving time demonstrates that Monte-Carlo move ordering is superior to the history heuristic.

Even when only a small number of value combinations are sampled (n_c = 1, 10, 100), the terminal node evaluation seems to provide a reasonably good estimate to compute the approximate value of interior nodes. But when n_c is 1000, it causes too much overhead.

If we focus on the range of n_c from 1 to 100, then for each depth limit d_m, the solving time almost always increases when n_t decreases. This indicates that the more descendant terminal nodes sampled, the better the performance will be. Thus it is best to sample all of those terminal nodes whenever possible.

The depth limit for Monte-Carlo move ordering is an intricate parameter that needs a trade-off between search efficiency and overhead computation. We set $n_t = 1$ and $n_c = 10$, and tested the solver's performance for d_m from 0 to 14. The result is shown in Figure 3. Note $d_m = 0$ implies only the history heuristic are used for move ordering, and $d_m = 14$ implies only the Monte-Carlo move ordering is used.

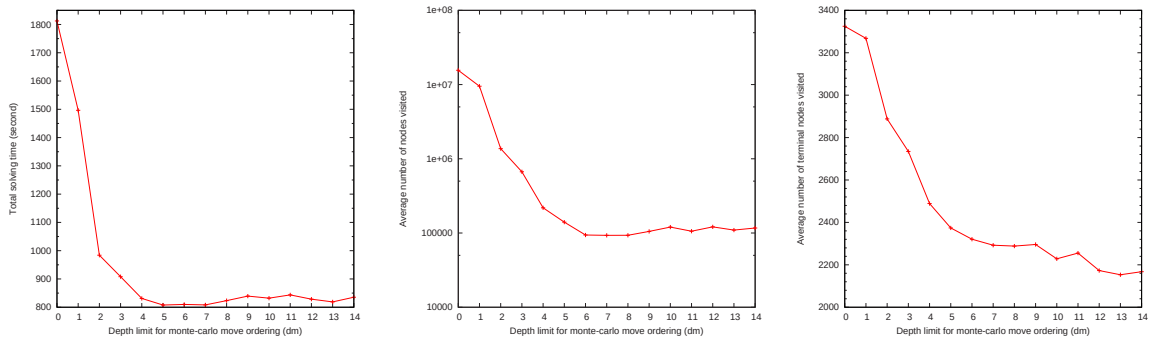

Figure 3: Statistics of the solver for different values of d_m (n_t=1, n_c=10)

Figure 3 clearly shows that when d_m is at least 4, the solver's performance is relatively stable in terms of solving time and the number of nodes visited. This fact suggests that the Monte-Carlo move ordering is more powerful for nodes close to the root of the game tree. It is also evident in the rightmost graph of Figure 3 that the number of terminal nodes gradually decreases as d_m increases, which proves the Monte-Carlo interior evaluation is more accurate than the history heuristic for move ordering in this game.

5.3 Performance of the Monte-Carlo Player

The relative strength of two players is tested by playing each of the 100 random games twice, switching colors for the second game. A player's winning probability of a game is the average of its winning probabilities in the two rounds. The perfect player always achieves 50% winning probability against itself, and at least 50% against any player.

The Monte-Carlo player has two parameters: the search depth (at least 1), and n_c for the approximate evaluation of terminal nodes. We tested the Monte-Carlo player with different configurations of these two parameters, and let it play against the perfect player. Figure 4 illustrate the winning probability of the perfect player as well as the ratio of the time the perfect player spent to that of its Monte-Carlo counterpart. The parameter d_m is set to the search depth (meaning only MCIE is used), and n_t is set to be 1, since this setting works best. In the figure, $n_c = all$ means the accurate evaluation of terminal nodes is performed.

The left graph in Figure 4 reveals an interesting property of the Monte-Carlo player: an increased search depth does not necessarily generate a better winning probability, and sometimes it even makes it worse. The Monte-Carlo player with the accurate terminal node evaluation is the best against the perfect player, but it uses almost the same amount of time, so it is not practical.

When the number of value combinations sampled, n_c, is from 100 to 1000 while the search depth is very shallow (1 or 2), the Monte-Carlo player performs very well. Especially when n_c is 1000

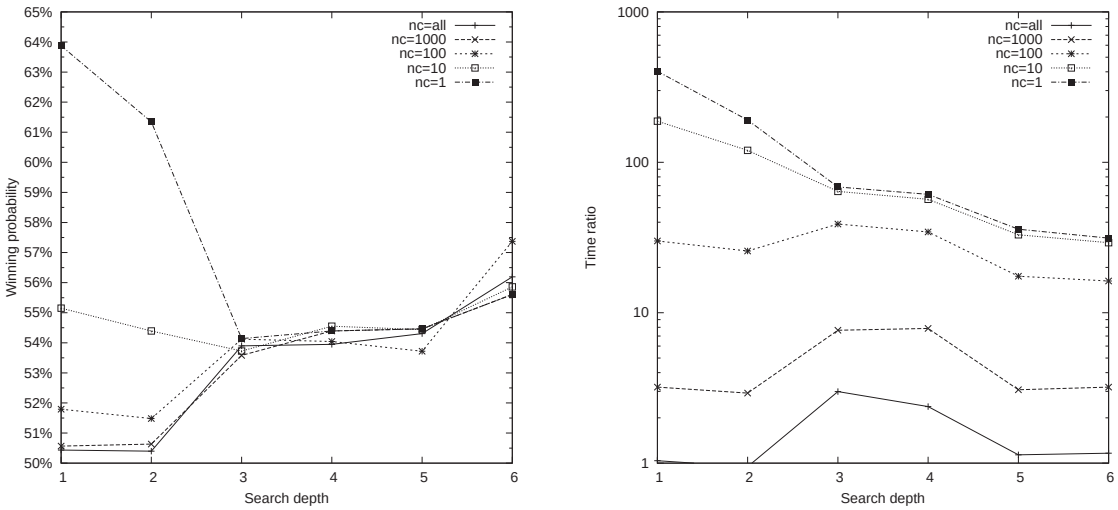

Figure 4: The perfect player against the Monte-Carlo player: winning probability and time ratio

and the search depth is 1, the perfect player only has a slightly better winning probability of 50.56%. That means the Monte-Carlo player is just about 1% weaker than perfect.

The Monte-Carlo player with n_c = 1000 needs time comparable to the perfect player (about 25% to 30%), and is much slower than the player with n_c = 100. The latter player seems to be a good compromise between time and accuracy: it plays games quickly, using 4% of the perfect player's time, and is only about 3% weaker.

It is not surprising that players with small n_c do not perform well. Insufficient sampling incurs large errors. The graph on the right side in Figure 4 suggests that a deeper search can make up for insufficient sampling to some extent.

5.4 Monte-Carlo Move Ordering

Since MCIE provides good estimates, it is important to provide a quantitative measurement to show how good it is with regard to move ordering.

As an experiment, one interior node at each depth is randomly selected from each of the 100 test games.

For each node, all pairs of legal moves are compared to test if the order of their exact values is the same as the order suggested by MCIE. If it is different, then Monte-Carlo evaluation makes a mistake, and the winning probability difference of the two moves is recorded. This value denotes the winning probability lost due to the mistake. The *average probability error*, defined as the average of this value from all move pairs, measures the quality of move ordering for this node. The influence of move ordering on choosing the best move is measured by the *worst probability error*, the winning probability difference between the best move and the move suggested by MCIE.

Figure 5 illustrates the influence of n_c on the quality of Monte-Carlo move ordering. Data points with probability error of 0 are not shown due to the logarithmic scale used. Again, n_c = all means that accurate terminal node evaluation is performed. The two graphs in Figure 5 clearly show that Monte-Carlo evaluation provides nearly perfect move ordering with marginal error if there are a substantial number of value combinations sampled. Even when only a small number of combinations sampled, such as when n_c is 10 or 100, the worst probability error is still less than 2%, though the error becomes larger when the node is closer to terminal nodes.

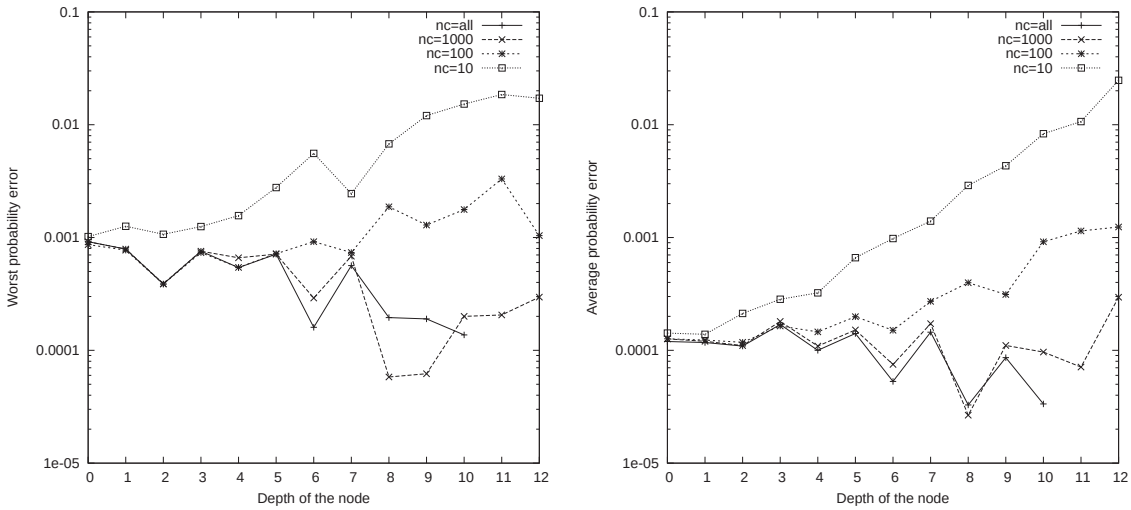

Figure 5: Probability error due to incorrect move ordering

6. CONCLUSIONS AND FUTURE WORK

This paper investigates SPCG, sums of 1-level probabilistic combinatorial games, and discusses methods to solve them, as well as strong heuristic players. Properties of this game are analyzed, and experimental results clearly show that Monte-Carlo methods are useful for evaluation of both terminal and interior nodes.

Very interestingly, comparing the average winning probability of all descendant terminal nodes of an interior node is a good indicator of relative value as measured by winning probability. Experimental results show convincingly that this heuristic performs very well in move ordering for this game, and that is why it could bring a big performance increase to the complete solver and also why the Monte-Carlo player based on it performs well against the perfect player. It is interesting to investigate when and why Abramson's simple expected-outcome evaluation provides a good heuristic in games, SPCG seems to be a good abstract model to study.

The SPCG solver and the Monte-Carlo player still have room for improvement. The bottleneck of the solver lies in the accurate evaluation of terminal nodes.

An improvement on sampling strategy would be to incorporate progress pruning (Billings *et al.*, 2002; Bouzy and Helmstetter, 2003). In the current Monte-Carlo player, each legal move is sampled at the same frequency. However, it is more efficient to use an adaptive strategy such that most of the effort is spent on those moves that have a high chance of being the best.

A SPCG solver could be incorporated in a Go program. The solver would be used on the high level to instruct the program to maximize its winning probability. As proposed in (Chen, 2005), such an approach might help to improve the playing strengths of current Go programs. It would present a significant new application of Monte-Carlo methods.

7. REFERENCES

Abramson, B. (1990). Expected-Outcome: A General Model of Static Evaluation. *IEEE transactions on PAMI*, Vol. 12, pp. 182–193.

Baum, E. and Smith, W. (1997). A bayesian approach to relevance in game-playing. *Artificial Intelligence*, Vol. 97, Nos. 1–2, pp. 195–242.

Berlekamp, E. R., Conway, J. H., and Guy, R. K. (1982). *Winning Ways for your Mathematical Plays.* Academic Press.

Billings, D., Davidson, A., Schaeffer, J., and Szafron, D. (2002). The Challenge of Poker. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 201–240.

Bouzy, B. and Helmstetter, B. (2003). Monte Carlo Go Developments. *Advances in Computer Games conference (ACG-10)* (ed. E. A. H. H. J. van den Herik, H. Iida), pp. 159–174.

Chen, K. (2005). Maximizing the chance of winning in searching Go game trees. *Information Sciences.* To Appear.

Müller, M. (1995). *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory*. Ph.D. thesis, ETH Zürich. Diss. ETH Nr. 11.006.

Müller, M. (2001). Partial Order Bounding: A new Approach to Evaluation in Game Tree Search. *Artificial Intelligence*, Vol. 129, Nos. 1–2, pp. 279–311.

Palay, A. (1985). *Search with Probabilities.* Morgan Kaufman.

Schaeffer, J. (1989). The history heuristic and the performance of Alpha-Beta enhancements. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, No. 11, pp. 1203–1212.

ACKNOWLEDGEMENTS

The authors would like to thank Markus Enzenberger and David Silver for their valuable comments on this paper. This work has been supported by the Alberta Informatics Circle of Research Excellence (iCORE) and the Alberta Ingenuity Fund.