Highly Volatile Game Tree Search in Chain Reaction

Dafyd Jenkins
School of Biosciences
University of Birmingham
UK, B15 2TT
djj134@bham.ac.uk

Colin Frayn
CERCIA, School of Computer Science
University of Birmingham
UK, B15 2TT
cmf@cercia.ac.uk

Abstract— **Chain Reaction is a simple strategic board game for two or more players. Its most interesting feature is that any static evaluation for board positions is highly volatile and can change dramatically as the result of one single move. This causes serious problems for conventional game-tree search methods. In this work, we explore an innovative approach using Monte Carlo analysis to determine advantageous moves through a stochastic exploration of possible game trees. We extend the concept of Monte Carlo analysis to include round-based progressive pruning. We also investigate the concept of volatility as a bias to the alpha-beta values within a hierarchical search tree model, in order to cope with the inherent unpredictability.**

I. Introduction

Chain Reaction is an as yet unstudied game played on a square or rectangular board of user defined size. The current work considers only two opposing players, though it is possible to play with a larger number. Due to the potential complexity of move resolution, Chain Reaction is a computer-only board game with implementations on many different platforms. It has very simple rules, yet is also very strategic and complex.

The object of the game is to remove all of the opponent's pieces from the board by causing chain reactions of 'explosions'.

A. Rules of Chain Reaction

Chain Reaction consists of several simple rules:

Game set up

The board is initially empty, and each player has a number of pieces of a specific colour.
Each square on the board is assigned a 'capacity' value depending on the number of neighbours it has (up, down, left and right):

- A corner square has 2 neighbours
- An edge square has 3 neighbours
- All other squares have 4 neighbours

Placing pieces

Each player takes turn in placing a piece of their specific colour on the board.
A piece may be played on a square if:

- The square is empty, or
- The square contains 1 or more pieces of the player's colour.

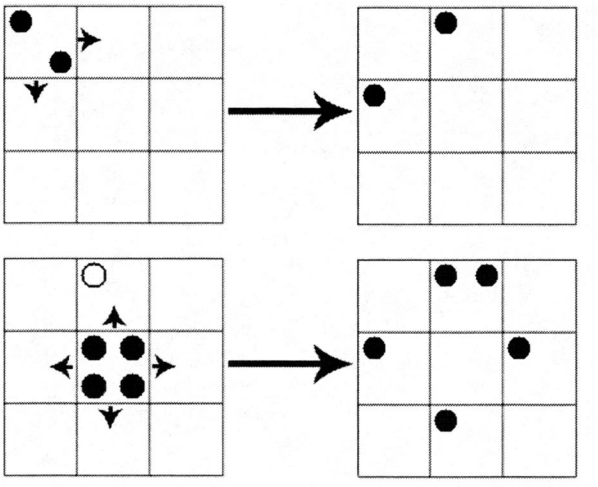

Fig. 1. (top) Square explosion (bottom) Square explosion and capture

Exploding a square

After a piece has been placed on the board, it must be checked for explosions.

A square explodes if it contains an equal or greater number of pieces to its 'capacity' value.

If the square explodes then the pieces in the square are removed and one piece is added to each of its neighbour squares. Any opposing pieces currently in the neighbour squares are removed and replaced with an equal amount of the player's pieces.

Each of the neighbours must then be checked for explosions, and so chain reactions of explosions may take place. Figure 1 (top) shows the outcome of a cell explosion, and Figure 1 (bottom) shows the outcome of a cell explosion capturing an opponents piece.

The order in which the board is updated makes a significant difference to the final outcome of the move. As such a specific order for updating the board is defined. The initial square in which the move is made is added to a queue. If the square has reached capacity, then the neighbouring squares are added to the queue in the order

1-4244-0464-9/06/$20.00 2006 IEEE.

CIG'06 (May 22-24 2006)

Left, Right, Up, Down.

B. Motivation for using Chain Reaction

Whilst Chain Reaction has very simple rules and game play, the game has an 'easy to learn; hard to master' complexity to it. It shares many similarities with Go, such as the importance of structure within a game position, and an extremely large branching factor (the exact branching factor is, at this time, currently unknown). This means that many standard game playing techniques give poor results. Chain Reaction also does not have a decreasing complexity as the game progresses, unlike chess for example, where the complexity begins to decrease as pieces are captured. The complexity only generally decreases during the extremely short end-game of Chain Reaction (usually one or two moves).

II. Standard approaches

Perhaps the most commonly used game playing technique is the Minimax algorithm, along with its many variants. Minimax has been used on a wide variety of games, most notably Chess (such as 'Deep Blue'[1] and 'ChessBrain'[2]) and Draughts/Checkers (such as 'Chinook'[3]). However, the Minimax algorithm has several drawbacks:

- High time complexity for large branching factors
- Requires large amounts of subjective domain knowledge incorporated into a heuristic function for board evaluation
- The 'horizon effect' in which good moves may be missed if the ply depth is not large enough, or bad moves may be inadvertently selected if a specific branch currently looks promising, but leads to an extremely poor move one or more ply ahead.

Evolutionary methods have also been widely used in game playing algorithms, in particular in Go. This is largely due to the inherent problems the Minimax approach encounters when applied to such a complex game. Neural networks have been applied to board evaluation to offset the problems associated with hand-crafting a sufficient evaluation heuristic function for a Minimax search[4]. Spatial reasoning techniques have also been used to influence neural network design for Go, in order to attempt a 'divide and conquer' approach to board evaluation [5], [4] and this also referred to as 'soft segmentation' [6].

Our original work consisted of attempting to use several evolutionary techniques, along with the classical Minimax algorithm to develop game players for Chain Reaction. A range of players were designed and played against each other, including an evolutionary player using neural network board evaluation functions influenced by spatial reasoning, together with genetic algorithms and data mining techniques for knowledge discovery. We also implemented a standard Minimax player with $\alpha - \beta$ pruning and a simple piece counter heuristic. Finally, we investigate the application of particle swarm optimisation to a piece-position table board evaluation heuristic. The results of these experiments can be found in our earlier work[7].

In our earlier work, we were unable to evolve significantly successful strategies using a spatial neural-network approach under a number of different training schemes. The motivation for this new work was to explore new and distinct techniques for evolving strategies within this complex game, and to investigate whether or not the new methods are capable of overcoming the difficulties of dealing with a highly volatile game tree.

III. Monte Carlo approach

The Monte Carlo method (also known as statistical sampling) is a stochastic technique designed to sample a large search space, allowing inferences to be made about the properties of that search space as a whole. This has been used to explore the behaviour of various physical and mathematical systems such as numerical integration and optimisation problems, and it relies on random (or pseudo-random) number generators to allow the non-determinism required to generate statistically valid results. The use of Monte Carlo simulations for a problem assumes that the problem can be described using probability density functions. The simulation then proceeds to randomly sample the probability density function many times. Such repeated sampling allows an average result to be generated, which is then used as the solution[8].

The Monte Carlo method was first applied to Go in 1993 by Brügmann[9]. In this paper the method was combined with a Simulated Annealing process which was used to assign probabilities to each available move. Using these probabilities many random games (samples) are played from the current board position until completion. The move which returns the highest average score from these statistical samplings (that is, the move with the highest win-fraction), will then be the move selected and played.

Whilst this is an incredibly simple algorithm, and is based purely on random sampling, it performs quite well with the authors citing a ranking of around 25 kyu (Kyu is the ranking system used within Go; a lower Kyu indicates a stronger player. A new or extremely novice player usually has a kyu of over 30) on a 9x9 board. This result is very interesting as the choice of the moves made is based solely on random numbers, and randomly sampled games, with no domain knowledge required other than the rules of the game.

Bouzy and Helmstetter also investigated the use of the Monte Carlo method in Go in their 2003 paper[10]. The authors used a simpler approach than that used by Brügmann called Expected-Outcome proposed by Abramson[11], and

introduce several modifications to the base algorithm.

A. Monte Carlo method for Chain Reaction

The Monte Carlo approach we propose is slightly different and simpler than the approach used by Bouzy and Helmstetter[10].

The Monte Carlo method we designed consists simply of sampling the required number of random games until a resolution is reached, for each legal move available to the player. Moves within the game simulations are selected at random from the current player's available legal moves. The probability density function used to sample the random moves is uniform, so all moves have an equal probability of selection. The outcome of each game is recorded. The move with the highest number of wins from the random game simulations is selected and played.

This method limits the domain knowledge required by the algorithm to an absolute minimum, and so only fundamental game rules are supplied to the player.

This simpler method was chosen due to the large computations often required to update the game board after each move. However, the 'game over' states are easily detectable. This is in contrast to Go which has a complex board update (for example locating eyes and 'dead' pieces) and also complex 'game over' states, involving difficult evaluations of territorial control.

B. Results of Monte Carlo method

Our Monte Carlo player was tested against a random player, and a Minimax player with a piece counter heuristic. 2000 games were played on a 5x5 board with each player alternating colours and therefore the first move. The Monte Carlo player generated between 1 and 100 samples per move each game. The Minimax player used a ply depth between 1 and 4 each game.

As can be seen from Figure 2 the Monte Carlo player wins over 90% of all games against the random player even at the lowest number of samples used in the Monte Carlo simulation. The results indicate a general decrease in playing strength of the Monte Carlo algorithm as the number of samples used per move is reduced.

We have investigated the playing strength of algorithms without considering the time spent per move. Our interest here is purely in the variation of playing strength with the internal parameters of the algorithms chosen. In future work, one could investigate the efficiency of the various algorithms after normalising by their speed. When comparing a Monte-Carlo player against a standard $\alpha - \beta$ search this is easy to implement by setting the latter to use a fixed-time limit instead of a fixed-depth limit.

IV. Progressive Pruning

One of the modifications used by Bouzy and Helmstetter was 'Progressive Pruning', which was applied to the algorithm in an attempt to speed-up the Monte Carlo method[10]. This simple modification involves 'pruning' candidate moves when they initially appear statistically inferior to other moves. This allows the algorithm to concentrate on a more careful analysis of the remaining, more promising possibilities.

As our Monte Carlo method did not use the same statistical framework as Bouzy and Helmstetter, a new Progressive Pruning algorithm was devised. This algorithm was based on a tournament selection strategy, in which a number of 'rounds' are played within the tournament, with only a certain fraction of candidate moves allowed to proceed to the following round. Once a certain maximum number of rounds has been played, the best remaining move is selected and played.

The number of rounds and samples per candidate move per round were arbitrary selected, as with the standard Monte Carlo algorithm. Further investigation of the most suitable values to use here is left for future work.

Each round the maximum capacity of candidate moves which are allowed to progress to the following round is reduced by $\frac{100\%}{rounds}$. For example if 5 rounds are selected, then the maximum capacity will be reduced by 20% each round. The number of wins recorded for each candidate move is carried forward each round, with only the top ranking (cumulative) players proceeding. On a 5x5 board, this means that the 5 worst players are discarded each round. However, the algorithm specifies that only the maximum number of candidates is reduced each round, so in most cases, with fewer than 20 candidate moves, no moves will

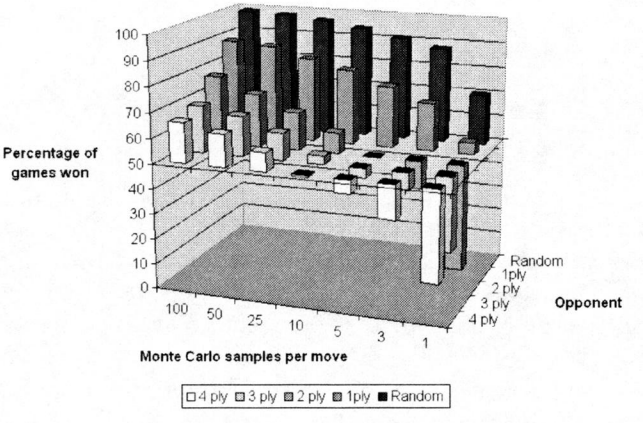

Fig. 2. Monte Carlo player against Random and Minimax players. A result of 50% indicates that the players are of equal strength.

be discarded until the 2nd or 3rd round.

Only the candidate moves which survive to the final round of the tournament will be analysed with the maximum specified number of samples.

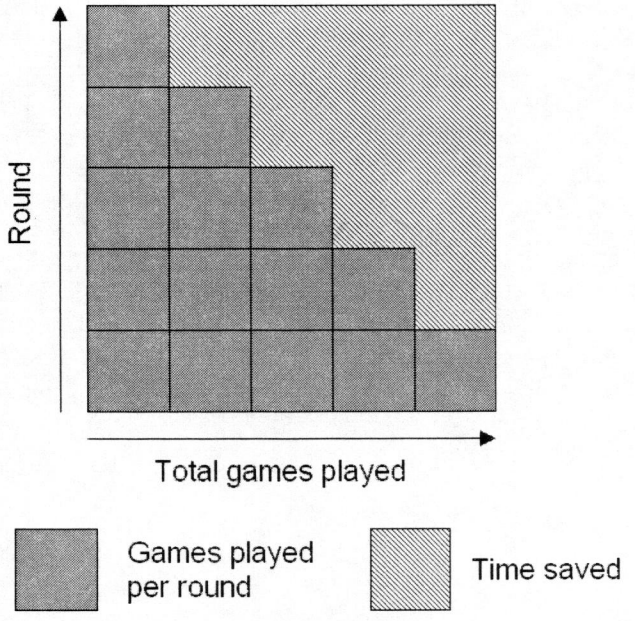

Fig. 3. Computational complexity saving of Progressive Pruning algorithm

As shown by Figure 3 the Progressive Pruning algorithm will never require more effort than the standard Monte Carlo method. The candidate moves surviving until the last round will be examined in exactly the same depth as with the standard method, however all other (supposedly inferior) moves will be analysed in decreasing levels of detail. Based on an average number of total moves available to the player at any given time of half the number of squares on the board, then the Progressive Pruning algorithm will require approximately 30% fewer samples than the standard Monte Carlo method on a 5x5 board. A varied pruning method could clearly reduce this still further, but with a trade-off against the risk of accidentally removing a promising branch.

A. Results of Progressive Pruning method

In order to test that the Progressive Pruning method did not worsen the game playing strength of our code, we played a number of games against a random player on a 5x5 board, once again with each player playing 1000 games as the first player, and 1000 as the second player. An average was taken over the first and second set of results. The total time taken was also recorded. These results were then compared to the results obtained from the standard Monte Carlo method.

	Monte Carlo samples per move				
	100	50	25	10	5
Monte Carlo	984	980.5	970	937	918.5
Progressive pruning	989.5	980.5	966.5	943	910

TABLE I
AVERAGE NUMBER OF WINS OF MONTE CARLO AND PROGRESSIVE PRUNING AGAINST RANDOM PLAYER

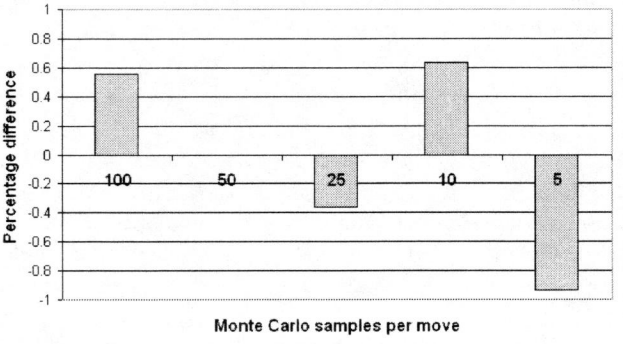

Fig. 4. Difference between results of Progressive Pruning and standard MC player against random player. Note the small scale of the y-axis

As shown by Table I and Figure 4 the performance of the Progressive Pruning algorithm is equivalent to that of the Monte Carlo algorithm against the random player, with variations less than one percent in either direction.

Figure 5 shows the reduction in computational complexity of the Progressive Pruning algorithm in comparison to the standard Monte Carlo algorithm. The results also confirm the 30% reduction estimated in section IV.

V. VOLATILITY

During our analysis of Chain Reaction it was noticed that the game has an usual characteristic. Game positions within Chain Reaction can vary extremely rapidly. This means that in many positions it is difficult to define a suitable heuristic

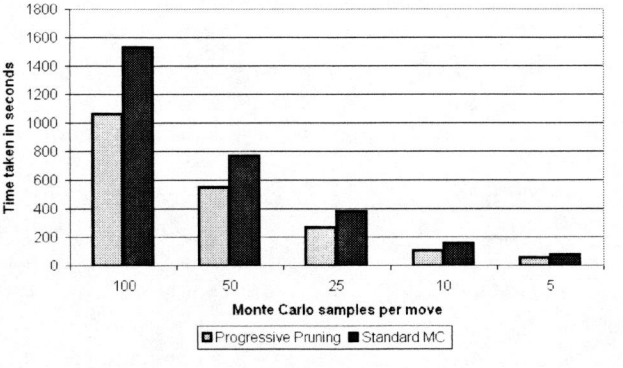

Fig. 5. Time taken for Progressive Pruning and standard MC player against random player

to evaluate a given game position. An example of this is shown in figure 6. Here we see an example game position where white seems to have lost, and indeed if black is to move next then the game should be over. However, if white is to move, then she will in fact win instantly. Although this is a purely manufactured game position, similar positions occur very frequently within the course of a game of Chain Reaction. Whilst the position may not lead to a win for the currently losing player, as in the example, the number of pieces, and territory of the players can fluctuate rapidly.

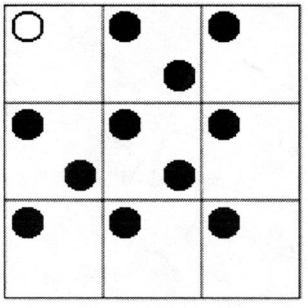

Fig. 6. Example of volatile game position

A. Theory of volatility

Our analysis of Chain Reaction and volatility leads us to believe that quantifying the volatility of the current game position may have a significant impact on the performance of a standard $\alpha - \beta$ game-tree search. We believe this for several reasons, most importantly:

- Any extra information and knowledge given to the evaluation heuristic can prove beneficial, provided the information is incorporated in an accurate way into the evaluation.
- It may search the game-tree more efficiently, by reducing cut-offs when using the $\alpha - \beta$ algorithm. This would be due to the more informed evaluation heuristic, but also the influence of the volatility score on the search window bounds. Whilst this may slow the algorithm down, as fewer cut-offs within the tree are made, the risk that a seemingly good move in a very volatile environment will be accepted is reduced.
- Volatility may also be beneficial when using unsafe pruning techniques such as 'delta cuts' by allowing an adaptive 'delta' parameter to be used, instead of the static parameter used with the technique.

B. Example of volatility

The concept of volatility was investigated as an alternative to the quiescent (or horizon) search algorithm for game-tree search. Quiescent search allows leaf nodes within the tree to be further expanded if 2 conditions are met:

1) The leaf node is non-terminal (e.g. the search has stopped due to the ply depth limit being reached)
2) The leaf node is classed as 'interesting'.

What makes a game position 'interesting' is highly subjective, but examples of 'interesting' moves in chess would be piece exchanges, promotions and check positions. If a fairly long set of moves is caused by a number of piece exchanges, then we could say that the current branch is not only interesting, but also volatile, due to the rapid fluctuations of player advantage. In situations such as these it is beneficial to expand the current branch of the tree until it is no longer volatile. For example, a series of piece captures may lead to the players Queen under threat, but if the move is a leaf due to the ply depth being reached, then the player will not take the vulnerability of the Queen into account. This may lead to very poor moves being made, as the player is unable to see 'over the horizon' to the next ply.

However, whilst quiescent search is an extremely useful algorithm for games such as chess, its application to Chain Reaction may be somewhat limited. This is due to the larger inherent volatility within the game, than for example chess. If we take the example of piece exchanges in Chain Reaction as the condition to describe 'interesting' moves and so lead to a quiescent search, then it is highly likely that a search would never terminate. In mid and end game situations in Chain Reaction, the piece turnover between players is extremely rapid. This means that most leaf nodes will need to be explored a lot further, which is clearly infeasible in terms of search time. We therefore propose that using volatility as a measure of the degree to which we can 'trust' results from each branch, may be a viable alternative to using a quiescent search.

C. Volatile game-tree search for Chain Reaction

Due to time constraints, the algorithm designed for Chain Reaction is used in a standard $\alpha - \beta$ algorithm without unsafe pruning techniques such as 'delta cuts' or 'razoring'. The algorithm used is as follows:

1) Generate required statistics about current game state. Statistics include quantifying the volatility of the potential moves, and also estimated pay-off of potential moves.
2) Generate a game-tree using the $\alpha - \beta$ algorithm.
3) Pass down estimated pay-off to appropriate leaf nodes.
4) Modify the α and β parameters using the quantified volatility to widen the $\alpha - \beta$ search window
5) Select appropriate move using Minimax algorithm

1) Generating statistics: Statistics about the current game position can be created using the standard Monte Carlo method (or progressive pruning method) described in the previous chapter. During each Monte Carlo sample, the number of pieces each player has is recorded at each move. From this the average pieces change per move can

be estimated using the following formula:

$$volatility = \frac{1}{m}\sum_{i}^{m}\frac{|p_i - p_{i-1}| + |o_i - o_{i-1}|}{2}$$

where m is the total number of moves played in the simulated game, p_i is the number of pieces on the board belonging to the current player on move i, and o_i is the number of pieces on the board belonging to the opponent player on move i.

The value generated from this formula is used as the quantified volatility for the given move. This process is then repeated for all legal moves from the current game position.

Future work may involve investigating the effect of playing only a limited number of moves when simulating the game, rather than playing until completion. It is likely that the volatility value generated for the entire game will be significantly different to specific parts of the simulated game, and so will be inaccurate at any particular time. herein lies another interesting avenue for future research.

The estimated pay-off of potential moves is simply calculated as the number of wins simulated by the Monte Carlo method of the given move. This value is taken as the normalised win percentage from 0 (no wins) to 1 (win all games). Once again this process is repeated for all legal moves from the current game position. This is used in the initial $\alpha - \beta$ move ordering.

2) Game-tree generation: The game-tree is generated using the standard $\alpha - \beta$ search algorithm, however several other parameters are also required. The volatility value and estimated pay-off is passed down the appropriate branches as the tree is created.

The generated score is then scaled using the estimated pay-off, so that moves with a smaller estimated pay-off will be scaled down more than moves with a larger estimated pay-off.

3) Modifying α and β parameters: Within the $\alpha - \beta$ algorithm, the 2 parameters α and β are used to cut branches from the game-tree which will result in a move at most no better than the best currently found. This allows the search to complete much more quickly as fewer nodes and branches have to be expanded. However, in a highly volatile game position, the scores generated for the move may not be very high (due to the simple piece counting heuristic), but the move may still be quite good. Therefore, the α and β parameters are modified using the current volatility, to widen the search window. The following 2 formulas are used:

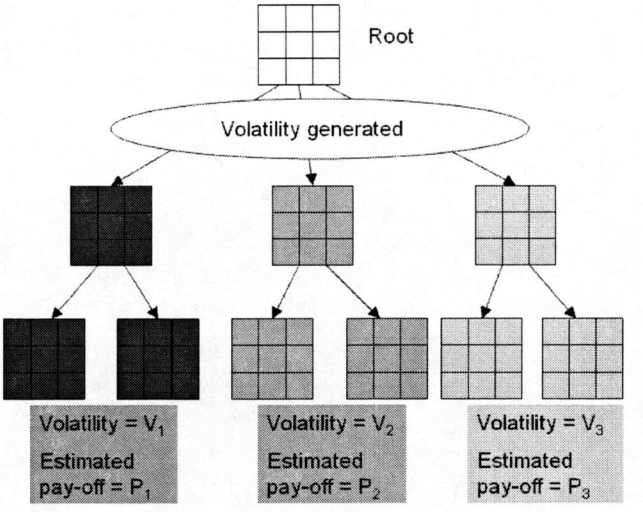

Fig. 7. Example of volatile game tree

$$alpha \leftarrow alpha - volatility$$

and

$$beta \leftarrow beta + volatility$$

This has the effect of widening the search window, meaning that it is more difficult for a seemingly good move within a highly volatile branch to cause a cut-off.

VI. Results of Volatility method

Our Volatile player was played against a random player, and a Minimax player with a piece counter heuristic separately. 1000 games were played on a 5x5 board as each of first and second player. The Volatile player generated between 1 and 100 samples per move and a ply depth between 1 and 4 each game. The Minimax player used a ply depth between 1 and 4, which was kept the same as that of the Volatile player.

Figures 8 and 9 show that the Volatile game tree algorithm has a negative effect on the strength of the player, compared with a standard game tree with the same ply depth. However, as the number of samples generated per move increases the strength of the player increases, and in the case of ply depths of 1 and 2 against the random opponent, and ply depth 1 against the equivalent Minimax opponent the Volatile game tree is a stronger player than an equivalent Minimax game tree.

Whilst the technique presented here does not improve performance of a standard $\alpha - \beta$ search, unsafe pruning techniques may benefit from using volatility for generating

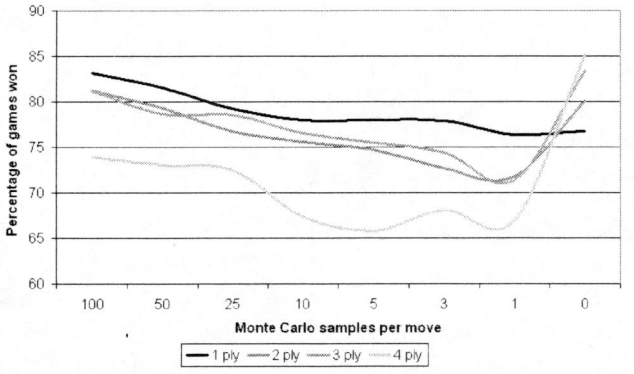

Fig. 8. Volatile player against random player

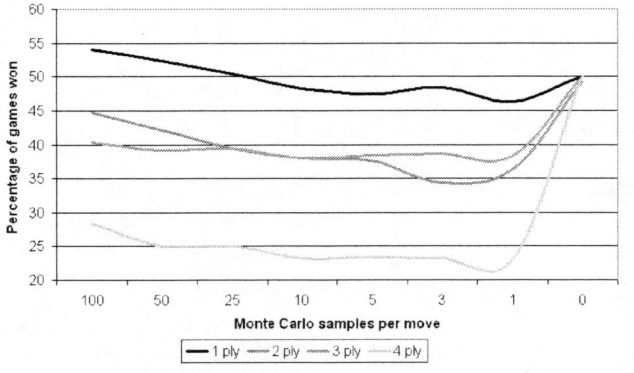

Fig. 9. Volatile player against Minimax player

adaptive parameters. For example, in the techniques used to prune leaf nodes or frontier nodes in game trees, some measure of the expected volatility in a standard game position is required in order to assess whether a poorly performing branch might still climb back above α on subsequent ply. Techniques known as futility pruning and delta cuts both use such a measure. In the default implementation, this buffer value is usually set to a fixed value, assumed to be sufficiently large to avoid (most) accidental false prunings. However, the addition of a volatility measure could be used to assign a dynamic buffer value which would make the pruning far more effective by encouraging more cuts in safer positions and avoiding erroneous cuts in highly volatile positions.

VII. Conclusions

We have presented a number of interesting results concerning the application of Computational Intelligence techniques to the game of Chain Reaction.

1) We have shown how the Monte Carlo technique of statistical sampling may be applied to the process of choosing a best move for any given game position.
2) We have shown that the Monte-Carlo method can, with moderate search time, outperform a simple $\alpha - \beta$ pruning minimax search, to depth of 4 ply or less. We anticipate that Monte Carlo players can also outperform deeper-searching minimax players, though this remains open for future work. Unfortunately, the time required for such a simulation increases significantly with the minimax depth.
3) We have identified a Progressive Pruning technique which is able to reduce the number of game samples required in a standard Monte Carlo search by approximately 30% for an average game situation.
4) We have investigated the introduction of a volatility estimate into the standard $\alpha - \beta$ search algorithm, compensating for the difficulty of implementing a quiescence search within Chain Reaction.

Acknowledgments

Dafyd Jenkins would like to acknowledge funding from the EPSRC, UK, in support of his MSc research. Colin Frayn would also like to acknowledge financial support from Advantage West Midlands. We also acknowledge Brian Damgaard and Lee Haywood, for interesting discussions during the earlier stages of this research.

References

[1] F. Hsu, M. S. Campbell, and A. J. Hoane, Jr., "Deep blue system overview," in *Proceedings of the 9th international conference on Supercomputing*. New York: Association for Computing Machinery, 03-07 July 1995.

[2] C. Justiniano and C. M. Frayn, "The chessbrain project: A global effort to build the world's largest chess supercomputer," *Journal of the International Computer Games Association*, vol. 26, no. 2, pp. 132–138, 2003.

[3] J. Schaeffer, J. C. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron, "A world championship caliber checkers program," *Artificial Intelligence*, vol. 53, no. 2-3, pp. 273–289, 1992.

[4] G. Kendall, R. Yaakob, and P. Hingston, "An investigation of an evolutionary approach to the opening of Go," in *Proceedings of the 2004 Congress on Evolutionary Computation CEC2004*. COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea: IEEE Press, 20-23 June 2004, pp. 2052–2059.

[5] K. Chellapilla and D. B. Fogel, "Anaconda defeats hoyle 6-0: A case study competing an evolved checkers program against commercially available software," in *Proceedings of the 2000 Congress on Evolutionary Computation CEC00*. La Jolla Marriott Hotel La Jolla, California, USA: IEEE Press, 6-9 2000, pp. 857–863.

[6] M. Enzenberger, "Evaluation in go by a neural network using soft segmentation," in *Proceedings of the 10th Advances in Computer Games Conference*. Berlin, Germany: Springer Science+Business Media, 20-23 June 2003.

[7] D. Jenkins and C. Frayn, "An evolutionary alternative to classical board game ai approaches," Master's thesis, School of Computer Science, University of Birmingham, UK, January 2005, contact authors for copies.

[8] C. S. E. Project, "Introduction to monte carlo methods," 1995, eBook, http://csep1.phy.ornl.gov/CSEP/MC/MC.html.

[9] B. Brügmann, "Monte carlo go," Max-Planck-Institute of Physics, Tech. Rep., 1993.

[10] B. Bouzy and B. Helmstetter, "Monte-carlo go developments," in *10th Advances in Computer Games conference, Graz 2003*. Kluwer Academic Publishers, 2003, pp. 159–174.

[11] B. Abramson, "Expected-outcome: A general model of static evaluation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 2, pp. 182–193, 1990.