Short Papers

A Problem Case for UCT

Cameron Browne

***Abstract*—This paper examines a simple 5 × 5 *Hex* position that not only completely defeats flat Monte Carlo search, but also initially defeats plain upper confidence bounds for trees (UCT) search until an excessive number of iterations are performed. The inclusion of domain knowledge during playouts significantly improves UCT performance, but a slight negative effect is shown for the rapid action value estimate (RAVE) heuristic under some circumstances. This example was drawn from an actual game during standard play, and highlights the dangers of relying on flat Monte Carlo and unenhanced UCT search even for rough estimates. A brief comparison is made with RAVE failure in *Go*.**

***Index Terms*—Bridge heuristic, flat models, *Go*, *Hex*, Monte Carlo methods, Monte Carlo tree search (MCTS), rapid action value estimate (RAVE) failure, upper confidence bounds (UCBs), upper confidence bounds for trees (UCT).**

I. Introduction

The recent success of Monte Carlo tree search (MCTS) methods for computer *Go* [1] and other domains [2] has seen a resurgence of interest in Monte Carlo (MC) methods for AI move planning in general. They are simple to implement, can make reasonable decisions in the absence of any heuristic knowledge, and can be surprisingly effective if suitably enhanced. Even flat[1] (i.e., nontree) MC players can provide convenient baseline players against which to test more sophisticated players.

The benefit of tree-based MC methods over flat MC is that they capture the opponent model,[2] for example, the upper confidence bounds for trees (UCT) algorithm has been shown to converge to the true minimax tree given sufficient time [3]. While a previous study showed how the lack of an opponent model can be disastrous for flat MC methods [4], this failure was only demonstrated for a contrived theoretical example. In this paper, we describe a board position that arose during standard play in a well-known game, which not only demonstrates similar pathological behavior for flat MC, but also troubles plain (unenhanced) UCT within reasonable search times.

In this section, we introduce the problem and its solution. Section II describes the MC approaches applied to this problem. Section III analyses their performance. Section IV discusses similar cases in *Go*. Section V outlines the conclusions that can be drawn.

Manuscript received April 02, 2012; revised May 29, 2012; accepted September 17, 2012. Date of publication September 21, 2012; date of current version March 13, 2013. This work was supported in part by the Engineering and Physical Sciences Research Council (EPSRC) under Grant EP/I001964/1 as part of the research project "UCT for Games and Beyond."

The author is with the Computational Creativity Group, Department of Computing, Imperial College London, London SW17, U.K. (e-mail: camb@doc.ic.ac.uk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2012.2220138

[1]Flat MC methods are also called pure MC methods in the literature.

[2]The concept of one or more agents actively making choices to minimize the player's reward.

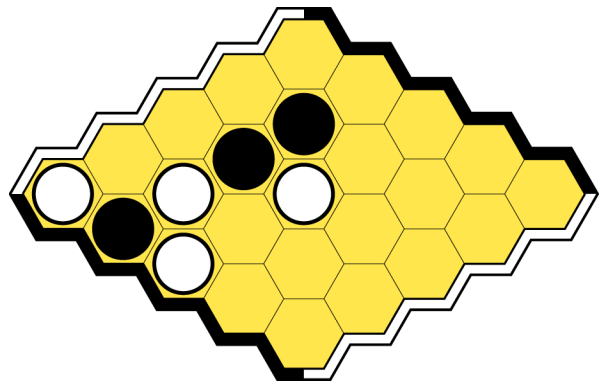

Fig. 1. *Hex* position with Black to play.

A. Statement of the Problem

Hex is a two-player, zero-sum game played on a hexagonally tessellated rhombus [5]. Players alternate placing a piece of their color at an empty cell, and win by connecting their sides of the board with a chain of pieces of their color. The standard board size is 11 × 11, but the game scales well to other sizes. Exactly one player must win each game, hence a result is guaranteed even if the board is randomly filled with pieces. This makes *Hex* well suited to MC methods.

Fig. 1 shows a 5 × 5 game of *Hex* in progress, with Black to play. The problem is to determine Black's best move. This position arose during standard play while testing a related algorithm [6], and constitutes a simple puzzle that most players—human or computer—should be able to handle without much trouble. Strong UCT-based *Hex* players such as MoHex and Wolve, with all enhancements on, find the solution instantly [7].

B. Analysis

In *Hex* terminology, a virtual connection (VC) between two sets of same-colored pieces is a pattern of pieces and empty cells which guarantees a connection for the owner of those pieces, regardless of whose turn it is. A virtual semiconnection (SC) is a weaker pattern that only guarantees a connection if it is the owner's turn. Note that the board edges are considered as groups of the appropriate color for these definitions [8].

Figs. 2 and 3 show two SCs for White, which guarantee a win for White with next move. Fig. 4 (shaded) shows the region of overlap of these two White SCs, which constitutes the "must play" region within which Black *must* play in order to avoid defeat. Moves outside this "must play" region are dead cells that constitute suboptimal (losing) moves.

Cell b may appear to be a reasonable move, as it intrudes into both of White's SCs while establishing two SCs for Black at the same time (Fig. 5). However, these potential threats have a critical point of overlap at cell e, where White can play next turn to establish its own VC that guarantees victory (Fig. 6).

Similarly, cell d may appear to be a reasonable move, as it intrudes into both White SCs while establishing a Black SC (Fig. 7) and reduces Black's distance-to-goal to only two moves. However, again White has a winning reply to d with the forcing sequence $\{a, b, e, f, l\}$.

Cell e is the only correct (winning) move for Black. Fig. 8 shows the Black VC established by this move, which White cannot block. All

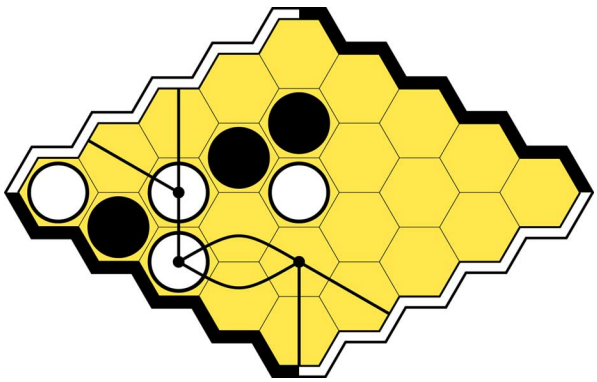

Fig. 2. An SC for White.

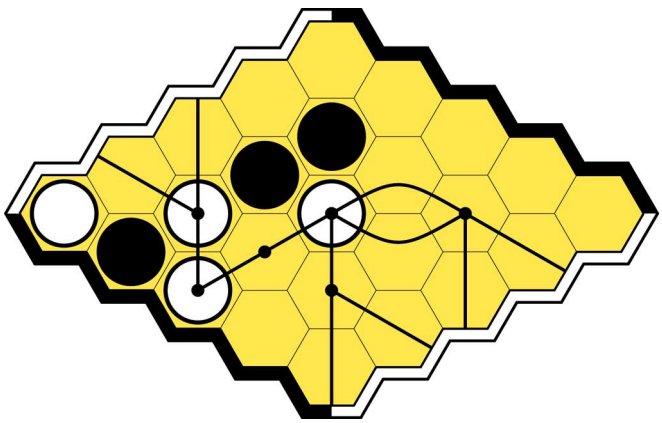

Fig. 3. Another SC for White.

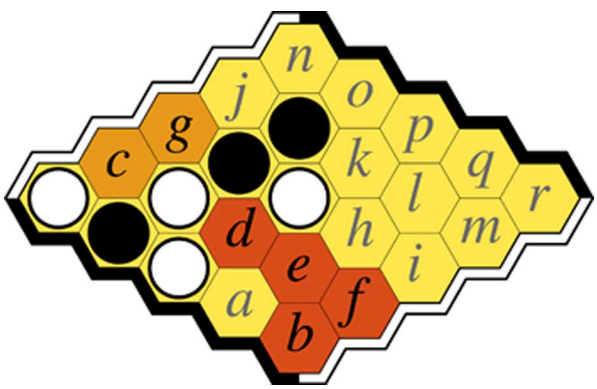

Fig. 4. Black's "must play" region of overlap.

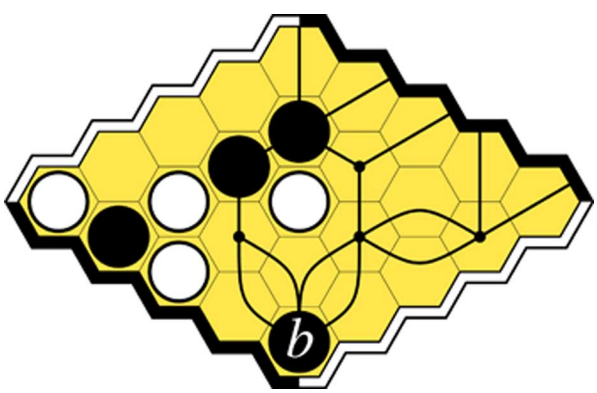

Fig. 5. Black move b looks promising...

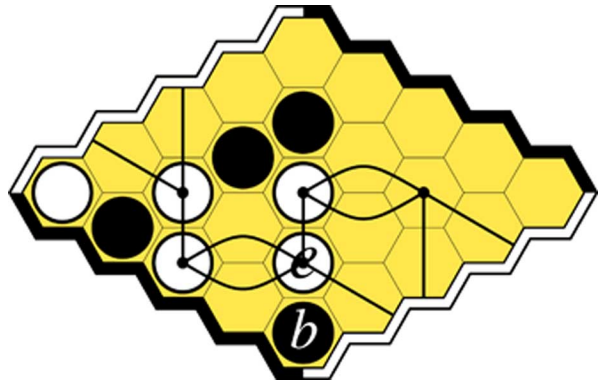

Fig. 6. …But it is defeated by White reply e.

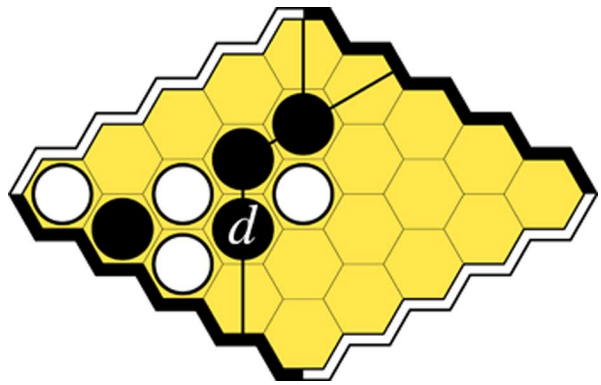

Fig. 7. Black move d minimizes distance-to-goal.

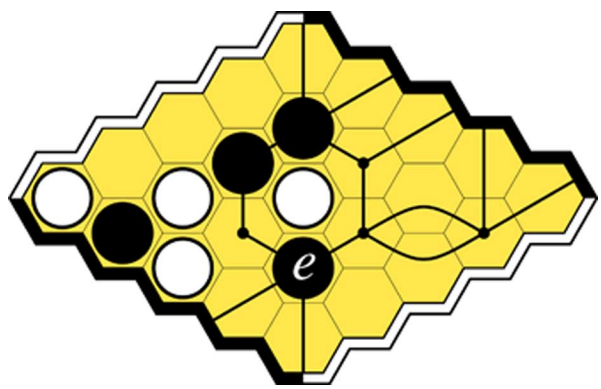

Fig. 8. Black move e gives a virtual (winning) connection.

other moves are provable losses for Black, except for c and g, which are delaying moves that must be followed by e next turn.

II. MC Players

We now describe a number of MC approaches for tackling this problem.

A. Flat Models

Flat MC involves running a number of simulations from the current state and selecting the action with the highest average reward. Coquelin and Munos [9] describe the more sophisticated flat UCB approach, which uses upper confidence bounds (UCBs) to select which action to apply first per playout. Flat UCB retains the adaptivity of standard UCT, while improving its regret bounds in certain worst cases for which UCT is overly optimistic.

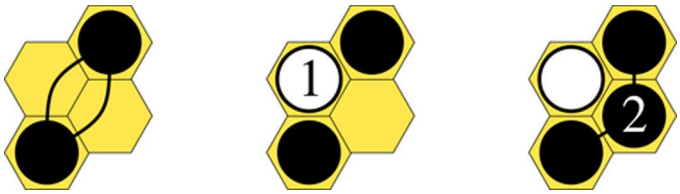

Fig. 9. Bridge pattern intrusion and completion.

B. Tree-Based Models

UCT is the standard tree-based MC algorithm [2], which balances the exploration of the search space with the exploitation of learned rewards. We distinguish between the tree policy that dictates action selection within the search tree, and the default policy that dictates action selection during playouts at the tree boundary [2].

UCT captures the opponent model and converges to the true minimax tree of a problem given infinite time [3]. However, the actual time required for reasonable convergence in any real-world sense remains an open question, and the algorithm can perform poorly if not enhanced or adapted to the problem being modeled. We consider two such enhancements: one domain dependent and the other domain independent.

C. Domain Dependent: Bridge Heuristic

Domain-dependent enhancements involve heuristic knowledge of the domain being modeled. These are typically most effective, but have the disadvantage of tying the application to one particular domain.

For example, Fig. 9 shows the most fundamental VC pattern in *Hex* called the bridge. The two empty cells are described as the carriers of this pattern. If White intrudes into either carrier (1), then Black can play at the other carrier (2) to restore the connection.

This heuristic knowledge can be incorporated into any MC playout by detecting such bridge intrusions and automatically playing the matching carrier as the next move. If a move creates more than one intrusion, then one matching completion is selected at random. This is one of the enhancements used by the world champion computer *Hex* player MoHex [8]. Such playouts, in which move choices are biased by domain knowledge, are described as heavy playouts.

D. Domain Independent: Rapid Action Value Estimate (RAVE)

Domain-independent enhancements, on the other hand, have general application to all domains. These are attractive in theory due to their broad potential, but tend to be less effective in practice.

RAVE is a domain-independent enhancement based on the all-moves-as-first (AMAF) heuristic, used for many years in computer *Go* [10]. UCT statistics are updated for each action selected by the tree policy as usual, but sibling nodes corresponding to other actions selected later in the playout are also updated in separate RAVE statistics. These separate statistics are used to bias future selections, and have the effect of "warming up" the tree, so that relatively unvisited nodes have at least some information to work with. RAVE is an essential part of all current strong computer *Go* players [2] and is also used by MoHex [8].

III. Performance

The performance of various flat and tree-based MC players is now compared for the example *Hex* position.

A. Experiment

The following MC players were implemented in Java:

- MC: Flat MC player;
- MC_h: Flat MC player with heavy playouts;

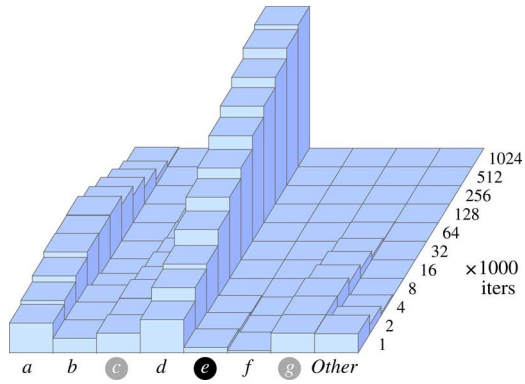

Fig. 10. Success plot for the MC player.

- UCB: Flat UCB player;
- UCB_h: Flat UCB player with heavy playouts;
- UCT: Plain (unenhanced) UCT player;
- UCT_h: UCT player with heavy playouts;
- UCT_r: UCT player enhanced with RAVE;
- UCT_{rh}: UCT player with RAVE and heavy playouts.

The tree-based players use standard UCB1 with exploration constant $C = 1/\sqrt{2}$ for node selection within the tree policy. This value was empirically found to give the best convergence for this problem. For the final (root child) action selection, the MC and MC_h players necessarily use greedy selection based on reward, while all other methods use a robust child (i.e., most popular root child) selection [11], [12], due to its slightly better performance over other root selection methods for this problem. The RAVE decay parameter was set to $k = 100$ for UCT_r and UCT_{rh} for this small problem.

Each player was applied to the *Hex* position shown in Fig. 1 over a number of epochs. The computational budget for each epoch e was set to a fixed number of iterations $i = 1000 \times 2^e$, giving searches of 1000, 2000, 4000, 8000, 16 000, 32 000, 64 000, 128 000, 256 000, 512 000, and 1 024 000 iterations. One thousand searches were performed for each player for each epoch, and the selected move recorded for each.

B. Results

Figs. 10–17 show the experimental results in the form of success plots by epoch and player type. Each row describes one epoch, with the height of the histogram columns along each row indicating the number of times each cell was selected over the 1000 searches for that epoch. The key move e is indicated in black, while the nonlosing delaying moves c and g are indicated in gray; these are the target solutions, with e preferred.

Fig. 10 shows the flat MC player's strong preference for losing move d. The inclusion of heuristic knowledge only makes MC_h converge to the wrong move more quickly (Fig. 11). Figs. 12 and 13 show a similar preference for losing move d in the flat UCB players UCB and UCB_h. These both converge to a consistent result more quickly than the flat MC players, but it is the wrong result.

Note that the flat players enhanced with the bridge completion heuristic (MC_h and UCB_h) initially select the correct moves c, e, and g with reasonable frequency, for very low iteration counts. However, this correct behavior is quickly drowned out by an increasing confidence in losing move d for higher iteration counts.

Fig. 14 shows the plain UCT player initially preferring losing move d for very low iteration counts, then stabilizing on losing move b until around 250 000 iterations, until finally converging on the correct move e. Around a million iterations are required to more or less guarantee correct behavior, but even then UCT will still occasionally

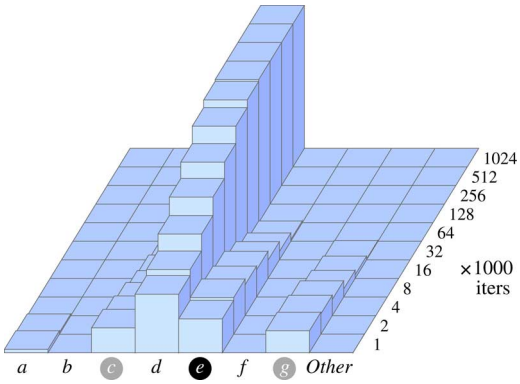

Fig. 11. Success plot for the MC_h player.

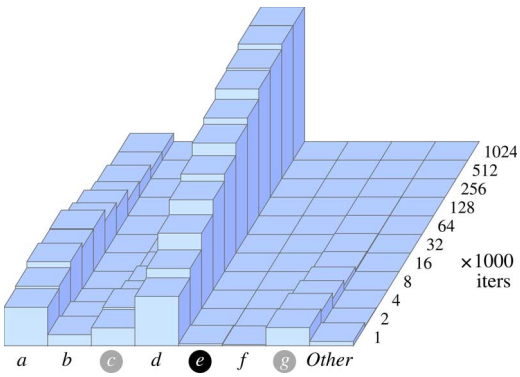

Fig. 12. Success plot for the UCB player.

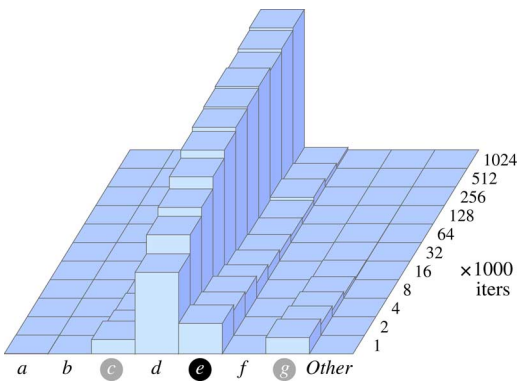

Fig. 13. Success plot for the UCB_h player.

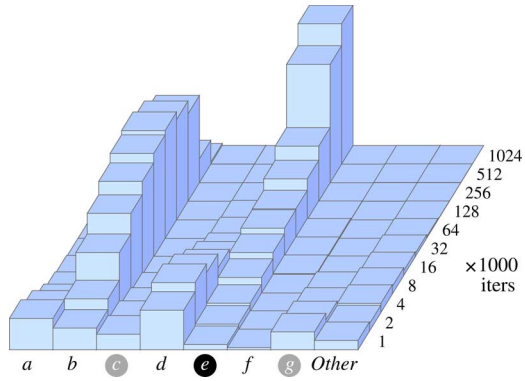

Fig. 14. Success plot for the UCT player.

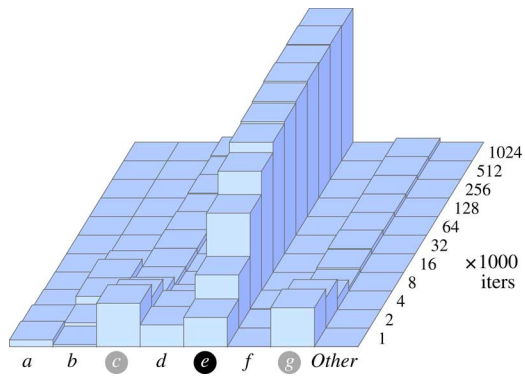

Fig. 15. Success plot for the UCT_h player.

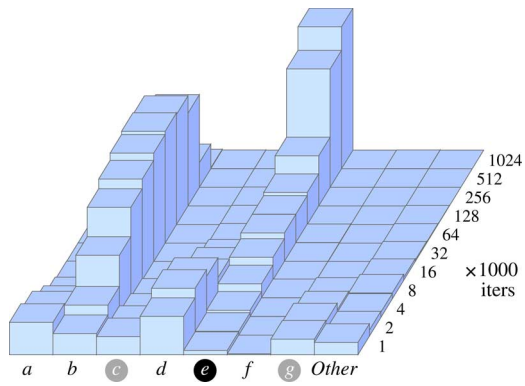

Fig. 16. Success plot for the UCT_r player.

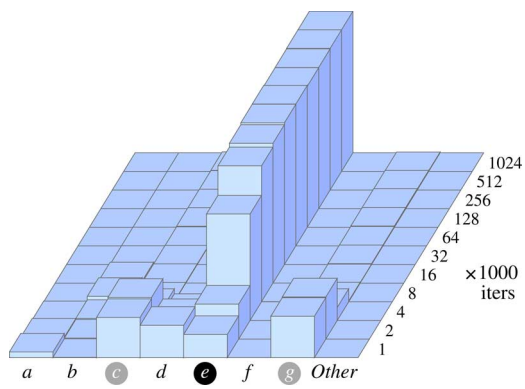

Fig. 17. Success plot for the UCT_{rh} player.

select losing move b. By contrast, the UCT_h player enhanced with heuristic knowledge only takes around 16 000 iterations to converge to the correct solution (Fig. 15). Furthermore, UCT_h will occasionally prefer delaying moves c and g regardless of the number of iterations, correctly indicating some degree of confidence in them.

Fig. 16 shows the success plot for UCT_r enhanced with the RAVE heuristic. Its performance looks similar to that of plain UCT, but it actually performs slightly worse than plain UCT in all epochs, even though this difference is not significant. This is surprising, given that RAVE is an essential part of the strongest computer *Go* players [2] and *Hex* players [7]. However, Fig. 17 shows that the RAVE heuristic is working correctly, as UCT_{rh}, enhanced with both RAVE and heavy playouts, gives the best performance of all players tested.

C. Timings

Table I shows the average time taken by each player during the 1 024 000 epoch, in thousands of iterations per second. Heavy playouts slow down the flat players by over 50%, but only cause a relatively minor slowdown of around 20% for the UCT players. The RAVE heuristic causes an additional slowdown of around 50%.

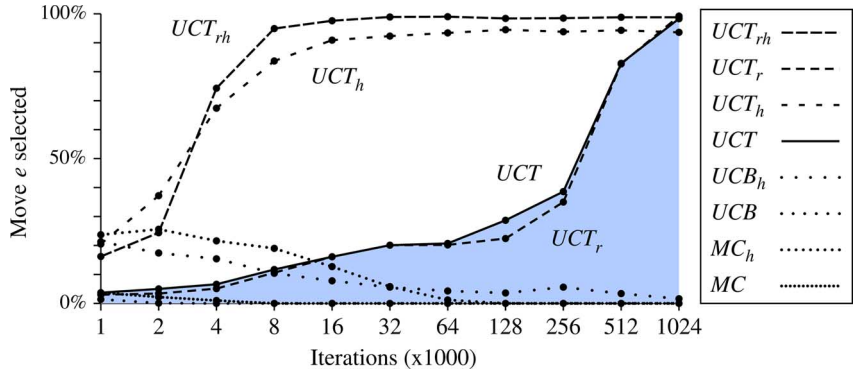

Fig. 18. Relative success in choosing winning move e.

TABLE I
ITERATIONS PER SECOND

Player	Iterations/s
MC	396 000
MC_h	139 000
UCB	294 000
UCB_h	124 000
UCT	101 000
UCT_h	83 000
UCT_r	50 000
UCT_{rh}	39 000

The experiment was run on a dual-core Macbook Pro with *i*5 processor, using Java 1.6.0_29 under JVM 20.1 with Eclipse Helios. The code has not been optimized.

D. Discussion

The most striking aspects of these results are:

- the flat player preference for the same losing move;
- the slow convergence of plain UCT;
- the enormous benefit of the simple bridge heuristic;
- the negative effect of RAVE with random playouts.

Fig. 18 summarizes these results in a relative success plot, which shows the percentage of times that winning move e is selected for each epoch. The shaded area indicates baseline UCT performance, which plateaus around the 25 000–100 000 range as it focuses on losing move b before correcting itself. The fact that plain UCT requires around a million iterations to reliably find the correct move for such a simple problem—and even then not with 100% certainty—indicates the danger of relying on plain UCT without enhancement.

The success rates of the UCT methods generally increase monotonically, while the flat models start poorly and just get worse. Of the flat models, those enhanced with heavy playouts deteriorate less quickly. The preference of the flat players for move d makes sense as this move intrudes into both of White's strongest SCs (Figs. 2 and 3), while at the same time reducing Black's distance-to-goal to only two moves. The flat methods clearly fail to capture the opponent model and the subtleties of these SCs. Bridge completion alone is not sufficient to correctly handle Black's more complex SC on the right-hand side of the board, but it is conjectured that the inclusion of more complex connection templates would address this problem.

The plain UCT player initially prefers the same losing move d as the flat players after 1000 iterations (Fig. 14). This is to be expected, as the early part of the search is dominated by random play, until the opponent model begins to emerge. However, the UCT player's preference for move b over epochs 4000–256 000 warrants deeper analysis.

Cell b is an edge cell, so a Black piece played there is guaranteed to connect with 100% certainty. On the other hand, the carrier of cell e to the same edge may be occupied by either {W, W}, {W, B}, {B, W}, or {B, B}, giving it only a 75% chance of connection on average with random fill. Forced bridge completion guarantees $d's$ connection to this edge with 100% certainty, while strictly random playouts incorrectly suggest that b is more strongly connected than e, and it requires many iterations of the tree policy until White's killer reply to b is recognized as being more detrimental than this apparent benefit.

This preference for cell b appears to be a manifestation of UCT's susceptibility to deceptive structures, as described by Hashimoto *et al.* [13]. The reason that the flat players prefer losing move d over losing move b appears to be that d has a more direct winning line that is more likely to be realized with random play.

Taking into account the speeds of the respective player types and the number of iterations, UCT requires around 50× more processing time than UCT_{rh} to find the correct move in this case. It has been pointed out that this may not surprise *Chess* programmers, who commonly see *Chess* positions for which an extra two or three plies of $\alpha\beta$ search (which can take around 50× more time) can mean the difference between finding the correct move or not. However, the point remains that such additional processing seems excessive for such a simple case as the 5 × 5 *Hex* problem.

The negative effect of the RAVE heuristic for UCT_r is another surprise, even if this effect is not significant. This can be explained by examining the results for the unenhanced flat MC players; if random playouts consistently converge to a losing move, then biasing the search with this information can only steer it in the wrong direction. In other words, the default policy provides the wrong information to the search, and this is not corrected until the tree policy grows sufficient structure to override this false information. Heavy playouts, on the other hand, give more realistic results, and bias the search in the right direction for UCT_{rh}.

IV. RAVE FAILURE IN *GO*

Hex is not the only game to suffer such RAVE failures. RAVE is an essential enhancement for top performance in MCTS *Go* players, as stated in Section II-D, but there are occasional cases in which its use results in the wrong move being chosen for *Go*, as well. This issue is discussed by *Go* researcher Martin Müller in his "RAVE problems" thread in the Computer-Go mailing list [14].

Fig. 19 shows one of the examples provided by Müller to demonstrate the problem (White to move). White's correct move is B2, which guarantees the capture of the two black stones and additional points for White: three points for the group's three freedoms and two points for the captured stones. Moving elsewhere would allow Black to play A5, which would put the B2 area in *seki*; neither player would want to play there for fear of capture, hence that area would not be owned by either player and not count toward either score. White risks losing this advantage by playing elsewhere, and hence losing the game, but that is

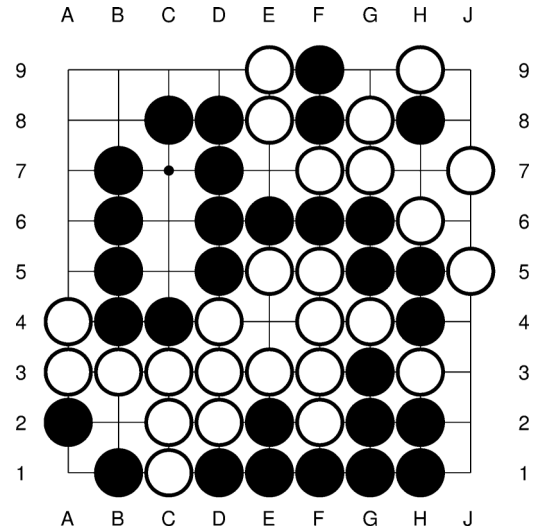

Fig. 19. RAVE failure in *Go*: FUEGO fails to play B2.

exactly what the highly ranked program FUEGO does, by choosing to play D9 when RAVE is enabled.

A clue to the reason for this degenerate behavior lies in the name of the concept on which RAVE is based: "all moves as first" [15]. B2 is a high-value (i.e., winning) move if played immediately, but a low-value (i.e., losing) move if played subsequently, hence its reward, if played later in the simulation, is a very poor "first move" estimation. This will occur for any situation in which there is a significant disparity between the true value of a move played immediately or subsequently.

In the same thread [14], Sheppard identifies two main cases relevant to *Go*:

1) RAVE searches a bad move because it is good later, e.g., breaking into an opponent's territory makes moves in that area good;
2) RAVE will not search the best move because it is bad later, e.g., *seki* situations as described above.

Sheppard suggests some possible approaches for addressing the problem, such as special case handling in his own *Go* program PEBBLES, or the use of FUEGO's "clump correction" facility for identifying and correcting move patterns known to be bad. Note, however, that clump correction is disabled in FUEGO to avoid other biases [16]. Hillis also proposes the use of "context codes," in which RAVE values are only updated for moves that match known moves in certain contexts [17]. Zhao and Müller also report the failure of RAVE to improve the results in a local form of UCT search for the game of *Go* [18].

We include this comparison to show that degenerate RAVE behavior is known and is not specific to *Hex*, even though the reasons are different in both cases. RAVE fails in the *Go* example because the correct move is valuable *only* if it is played immediately, whereas RAVE fails in the *Hex* example because the default policy converges to the wrong result, regardless of move order. The problem is easily rectified in the *Hex* case described above by improving the playouts using domain knowledge, as per UCT_{rh}. However, the problem occurs in *Go* even with highly optimized heavy playouts, and its general solution is still an open question.

V. CONCLUSION

This paper describes a simple 5×5 *Hex* position that arose during actual play, and the surprisingly poor performance of both flat MC and plain (unenhanced) UCT for this problem. Plain UCT requires around a million iterations to reliably find the correct move—far more than might be reasonably expected—and even then does not guarantee 100% accuracy. Given more search time, flat MC players only converge to the same (losing) move with greater certainty; they are worse than simply selecting a random move without any form of playout.

The inclusion of domain knowledge in the form of heavy playouts with bridge completion dramatically improves UCT performance, to reliably give correct results in a fraction of the time. Heavy playouts delay the convergence of the flat players to the wrong move, but otherwise do not help.

It is shown that RAVE can actually have a negative effect on performance, if the information gleaned from playouts is unreliable. This example highlights the dangers of relying on flat MC methods, plain UCT, and even UCT enhanced with RAVE, even for rough baseline estimates or comparative test players, even for simple problems.

ACKNOWLEDGMENT

The author would like to thank R. Hayward for feedback on an earlier draft, M. Müller for clarifications, S. Tavener for analysis, and the anonymous reviewers for useful suggestions.

REFERENCES

[1] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Proc. 5th Int. Conf. Comput. Games*, Turin, Italy, 2006, pp. 72–83.
[2] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.
[3] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Proceedings of the 17th European Conference on Machine Learning*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2006, vol. 4212, pp. 282–293.
[4] C. Browne, "On the dangers of random playouts," *Int. Comput. Games Assoc. J.*, vol. 34, no. 1, pp. 25–26, 2011.
[5] C. Browne, *Hex Strategy: Making the Right Connections*. Natick, MA, USA: A.K. Peters, 2000.
[6] C. Browne and S. Tavener, "Bitwise-parallel reduction for connection tests," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 2, pp. 112–119, Jun. 2012.
[7] R. Hayward, *Personal Correspondence.* 2012.
[8] B. Arneson, R. B. Hayward, and P. Henderson, "Monte Carlo tree search in Hex," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 4, pp. 251–257, Dec. 2010.
[9] P.-A. Coquelin and R. Munos, "Bandit algorithms for tree search," in *Proc. Conf. Uncertainty Artif. Intell.*, Vancouver, BC, Canada, 2007, pp. 67–74.
[10] S. Gelly, "A Contribution to reinforcement learning; Application to Computer-Go," Ph.D. dissertation, Informatique, Univ. Paris-Sud, Orsay cedex, France, 2007.
[11] F. C. Schadd, "Monte-Carlo search techniques in the modern board game Thurn and Taxis," M.S. thesis, Dept. Comput. Sci., Maastricht Univ., Maastricht, The Netherlands, 2009.
[12] G. M. J.-B Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy, "Progressive strategies for Monte-Carlo tree search," *New Math. Natural Comput.*, vol. 4, no. 3, pp. 343–357, 2008.
[13] J. Hashimoto, A. Kishimoto, K. Yoshizoe, and K. Ikeda, "Accelerated UCT and its application to two-player games," in *Advances in Computer Games*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2012, vol. 7168, pp. 1–12.
[14] M. Müller, "RAVE problems, Computer-Go mailing list," 2009 [Online]. Available: http://www.mail-archive.com/computer-go@computer-go.org/msg12189.html
[15] B. Brügmann, "Monte Carlo Go," Max Planke Inst. Phys,, Munich, Germany, 1993.
[16] M. Enzenberger and M. Müller, "FUEGO—An open-source framework for board games and Go engine based on Monte-Carlo tree search," Univ. Alberta, Edmonton, AB, Canada, 2009.
[17] D. Hillis, "RAVE problems, Computer-Go mailing list," 2009 [Online]. Available: http://www.mail-archive.com/computer-go@computer-go.org/msg12204.html
[18] L. Zhao and M. Müller, "Using artificial boundaries in the game of Go," in *Computers and Games*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2008, vol. 5131, pp. 81–91.