Transpositions and Move Groups in Monte Carlo Tree Search

Benjamin E. Childs James H. Brodeur Levente Kocsis

***Abstract*— Monte Carlo search, and specifically the UCT algorithm, has contributed to a significant improvement in the game of Go and has received considerable attention in other applications. This article investigates two enhancements to the UCT algorithm. First, we consider the possible adjustments to UCT when the search tree is treated as a graph (and information amongst transpositions are shared). The second modification introduces move groupings, which may reduce the effective branching factor. Experiments with both enhancements were performed using artificial trees and in the game of Go. From the experimental results we conclude that both exploiting the graph structure and grouping moves may contribute to an increase in the playing strength of game programs using UCT.**

I. Introduction

Monte Carlo search, and specifically the UCT algorithm [1], has both contributed to a significant improvement in the game of Go [2] and has received considerable attention in other applications as well (e.g., general game playing [3], or parameter optimization [4]). Monte Carlo programs simulate a large number of games starting from the current position, and select the move that led to a win most often. During simulation, the UCT algorithm treats the selection of moves as a multi-armed bandit problem, using the UCB[1] algorithm [5] for selection. Since UCB converges to the best arm in the bandit setting, UCT similarly converges to the best move (see [1] for a theoretical analysis of the algorithm).

In the 'basic' UCT algorithm, the space of the game is treated as a tree, which can lead to having multiple nodes for the same position. This is the case when several paths are leading to the same positions (referred to usually as transpositions). In game programs based on some alpha-beta variants, previously visited positions are identified by storing them in a transposition table. In chess, transposition tables are one of the most basic building blocks, and their use and design has been extensively studied (see e.g., [6]). In the context of Monte Carlo search, handling transpositions efficiently may have an important effect as well; nevertheless, it has received little attention so far. In this article we investigate various solutions to implement transpositions in UCT. Moreover, we consider an additional situation when transpositions appear 'artificially'; that is, when moves are intentionally grouped. This is motivated by the presumption that the information gained for one move generalizes to other moves in the group, and the technique can lead to a reduction of the effective branching factor[2] as discussed in Section III. For move groupings, additional nodes are introduced. Group selection is then handled similarly to move selection. Grouping moves has been discussed in [7], but only non-overlapping groups were considered. If groups overlap, the same move is included in many paths (corresponding to the different groups it is a member of) in the tree, and that leads to artificial transpositions (nodes reached by identical move sequences, but different node sequences). This may lead to a significant increase in transposition count, which raises even further the importance of efficiently handling the transpositions.

While attempting to adapt UCT to deal with transpositions, we consider only directed acyclic graphs (DAGs). The presence of cycles in a simulated game is related to the Graph History Interaction (GHI) problem (see e.g., [8] or in Go [9]) that is induced by leaving out some information about preceding positions in the state representation that are relevant for future play (violating essentially the Markov property). In principle, to overcome the GHI problem in UCT it would suffice to store all relevant history information for a position (e.g., in Go it would be necessary to identify ko or superko). However, in practical tournament programs for optimal performance only the small cycles are worth detecting (in Go that would amount to detecting ko). Cycles may pose a problem in UCT even if the state representation includes all relevant information; more precisely, if we update the statistics for a node as many times as it occurs in a particular simulated game. Similar problems occur for Monte Carlo policy evaluation for Markov Decision Processes (see e.g., [10]) and solutions that are suitable there are likely to work in UCT as well. A simple solution would be to update the node only the first time it is visited (as in first-visit Monte Carlo methods).

The article is organized as follows. The UCT algorithm is described briefly in Section II. Enhancements to the algorithm to deal with transpositions and move groups are discussed in Section III, while the experiments on artificial trees and Go are described in Section IV. Section V presents the conclusions.

Benjamin E. Childs is with the Computer Science Department of Worcester Polytechnic Institute, 100 Institute Rd, Worcester MA, 01609, U.S.A; email:bchilds@alum.wpi.edu.

James H. Brodeur is with the Computer Science Department of Worcester Polytechnic Institute, 100 Institute Rd, Worcester MA, 01609, U.S.A; email:jbrodeur@alum.wpi.edu.

Levente Kocsis is with the Computer and Automation Research Institute of the Hungarian Academy of Sciences, Kende u. 13-17, 1111 Budapest, Hungary; email: kocsis@sztaki.hu.

We would like to thank Gábor Sarközy and Stanley Selkow for their assistance with our research.

[1]UCB stands for Upper Confidence Bounds, while UCT stands for UCB applied to trees.

[2]The branching factor is the number of moves available in a position. In the alpha-beta context the effective branching factor relates to the increase in tree size, when the search goes one ply deeper. In Monte Carlo search it can be related to the number of simulations that are necessary to allocate to sub-optimal alternatives. Reducing the effective branching factor in games like 19x19 Go can have a significant impact for UCT based programs.

II. The UCT algorithm

The UCT algorithm [1] is a best-first search method that builds its tree by repeatedly simulating game episodes starting from an initial position. The tree is built by adding the information gathered during a single episode to the tree in an incremental manner. In a basic setting each node of the tree corresponds to a position that has been visited. The information stored for a node s after t games have been simulated includes the number of times the node has been visited $N_s(t)$ and the average result for the games going through $Q_s(t)$. These statistics can also be defined for a move a available in position s: the number of times the move has been selected in the position $N_{s,a}(t)$ and the corresponding average result $Q_{s,a}(t)$. At termination the search returns the move with the best average result.[3]

The above description fits most Monte Carlo tree search algorithms. The critical part is the selection of moves during the simulation. In UCT the selection amongst moves available in a position is comparable to a multi-armed bandit problem. More precisely, the UCB1 algorithm [5] is employed as follows. During the tth simulated game, for each move available in a current position s, an upper confidence bound is computed. This is the sum of the average result, $Q_{s,a}(t-1)$, and a bias term, $c_{N_s(t-1),N_{s,a}(t-1)}$. According to UCB1, the bias is chosen to be

$$c_{n,m} = \sqrt{\frac{2 \ln n}{m}}. \quad (1)$$

The move with the highest upper confidence bound is selected for simulation. In the simplest implementation, when a move has not been tried before, a maximum (infinite) confidence bound is assumed.

There are several variants of the algorithm described above. These involve both the decision of when to add a position to the tree, and how to select moves in unvisited (or under sampled) positions. In the experiments with artificial trees described in [1] all visited positions were added to the tree. The same approach is followed for artificial trees in Section IV-A. In most Go programs for a simulated game only the first position not yet present in the tree is added. In most game programs the sampling policy for unvisited positions is based on game dependent knowledge, which often is employed in a diminishing manner for positions that were visited a few times.

III. UCT enhancements

A. Transpositions

The algorithm described in Section II builds a tree that can have multiple nodes for the same position. To share information amongst such nodes, such transpositions have to be identified first. In most games, transposition tables (see e.g., [6]) are the usual choice for storing information about positions, and the sharing of information to subsequent occurrences of the same position during the search. While identifying transpositions is a rather simple task, adapting the move selection is less straightforward. In the following, we outline several selection rules. The preference for any of these may depend on several factors including the frequency of transpositions, the speed of updating the information in the nodes, and other such implementation issues.

In Section II we introduced some notations; we refine these, along with some additional notations. The set of available actions in a position s is denoted by $A(s)$, and the position that results after playing move a in position s by $g(s,a)$. Regarding the information collected after t games were simulated: $N_{s,a}(t)$ is the number of times move a has been simulated in position s, $N_s(t)$ is the number of times position s has been visited, $Q_s(t)$ is the average result of the simulated games going through s, and $Q_{s,a}(t)$ is the average result of the simulated games when move a was selected in s. The choice of a selection policy π in the tth simulation is denoted by $I_s^\pi(t)$. The selection policies will include the bias term $c_{n,m}$ as defined by Equation 1. In the following we outline four UCT variants:

(0) The most simple choice is to exclude detecting transpositions, and use the basic variant unaltered. This may be convenient if few transpositions occur in the game. This selection rule will be referred to as UCT0.

(1) If transpositions are identified, it is sensible to share the information that relates to the move selection in the particular position. The selection rule in this case is same as the basic variant, but the statistics are cumulated for the position independently of where it occurs in the tree (while in UCT0 the statistics are dependent on the path to the position):

$$I_s^{UCT1}(t+1) = \underset{a \in A(s)}{\operatorname{argmax}} \left(Q_{s,a}(t) + c_{N_s(t),N_{s,a}(t)} \right)$$

(2) A step further in using the shared information is replacing the $Q_{s,a}$ by $Q_{g(s,a)}$, thus using a refined value estimate[4] for the move selection of the parent. The selection rules is as follows:

$$I_s^{UCT2}(t+1) = \underset{a \in A(s)}{\operatorname{argmax}} \left(Q_{g(s,a)}(t) + c_{N_s(t),N_{s,a}(t)} \right)$$

It would be tempting to use the number of simulations that lead to the evaluation of the transposition for the bias term as well. However, this could lead to the value of the parent converging to incorrect value, if most of the simulations for the position that follows the best move in the position is reached frequently through different paths (i.e. when $N_{s,a}/N_s$ is much lower than $N_{g(s,a)}/N_s$).

(3) In UCT2, the refined value estimation is used only in selecting the moves in the parent node. Since $Q_{g(s,a)}$ is a more reliable estimate than $Q_{s,a}$ and the estimate of the parent is a combination of the estimation of the child nodes, it makes sense to refine the estimate of the parent with the extra information available for the child nodes. This can be

[3] Some implementations prefer the most explored move instead, which will often be the move with the best average result as well.

[4] $Q_{g(s,a)}$ includes at least the samples of $Q_{s,a}$, but potentially additional samples as well, and therefore is a statistically more accurate estimate.

done by weighting the estimate of the child node with the number of times the move that lead to the child has been simulated. Weighting with $N_{s,a}$ is necessary for the correct convergence (cf. the comment for UCT2). The selection rule for this variant is as follows:

$$I_s^{UCT3}(t+1) = \underset{a \in A(s)}{\operatorname{argmax}} \left(Q_{g(s,a)}^{UCT3}(t) + c_{N_s(t),N_{s,a}(t)} \right)$$

$$Q_s^{UCT3}(t) = \sum_{a \in A(s)} \frac{N_{s,a}(t)}{N_s(t)} Q_{g(s,a)}^{UCT3}(t)$$

In the previous variants only the statistics that involve nodes along the simulated path have to be updated. In this case all nodes that preceded in some previous simulation the positions included in the current simulation have to be updated, which is much more cumbersome and time consuming. It is possible to update, however if references to all parents are stored, and it might be worthwhile if the simulation is significantly more time consuming than the update.

The updating procedure should have two components. For those non-terminal (or internal) nodes s that were present in the tth simulated game, the value of Q is updated by $\Delta Q_s^{UCT3}(t) = Q_{g(s,I_s^{UCT3}(t))}^{UCT3}/N_s(t)$. Thus the Q value of the child node sampled is added to the sum of results (instead of the actual result obtained in the simulation). Additionally, for all internal nodes s that have some child nodes with altered Q value[5] the update rule is $\Delta Q_s^{UCT3}(t) = \sum_{a \in A(s)} N_{s,a}(t) \Delta Q_{g(s,a)}^{UCT3}(t)/N_s(t)$. Algorithmically the simplest choice to implement these update rules is to propagate from the leaf node towards the root vectors identifying the altered nodes for a specific depth. The experiments in Section IV-A.1 with UCT3 are based on this implementation.

The variants outlined above appear to us as the main solutions to adapt UCT in the presence of transpositions. There are additional variants similar to these that may use some specificity of the graph. For example, in the case of the artificial transpositions introduced by grouping moves, it may be interesting to exploit the fact that some paths are differing only in the group nodes.

B. Grouping moves

In some games a move consists of several steps (e.g., in Amazons first a piece is moved and then arrow is shot). In the program described in [11], first the piece move was selected (according to UCT), and then in a similar way the target of the arrow was chosen. If the moves are selected according the UCB1, the frequency of selecting suboptimal moves after n visits is $\alpha N \log(n)/n$, where N is the number of available moves and α is a variable that depends on how fast the optimal move is identified. In the particular case of Amazons when a move is made in two steps, and denoting N_1 the number of moves available for the player's pieces, and N_2 the number of places the arrow can land, the frequency of selecting suboptimal moves is $\alpha(N_1 + N_2)\log(n)/n$ for the two stage selection and $\alpha N_1 N_2 \log(n)/n$ for selecting amongst all combined moves. Thus, in this example, the frequency of selecting suboptimal moves can be reduced by $N_1 N_2/(N_1 + N_2)$ times. If the branching factor of the game is high, this technique can reduce the effective branching factor significantly.

Similar benefits can be achieved if the available moves are grouped. One presumes that grouping moves can have beneficial effect if there is some form of correlation amongst the moves in a group, and therefore the information gained for on move generalizes to the other moves. In the game of Go, grouping moves was tested with some success by [7]. As stated in the introduction, additional nodes are introduced for move groupings. Group selection is then handled similarly to move selection. Although [7] considered only non-overlapping (disjoint) groups, it is natural to extend to overlapping groups as well. If groups overlap, the same move is included in many paths (corresponding to the different groups it is a member of) in the tree, and that leads to artificial transpositions (nodes reached by identical move sequences, but different node sequences). This may lead to a significant increase in transposition count, which raises even further the importance of efficiently handling the transpositions.

IV. Experiments

In order to quantify the effect of the modifications to UCT, we performed a number of experiments using artificial game trees and computer Go.

A. Artificial Game Trees

Due to the notoriously large size of computer Go game trees, it is impractical to calculate the correctness of any given search algorithm. In order to determine with certainty if the search algorithm selects the correct move, one must know the correct move. This is not possible in Go, except in very limited end game scenarios. However, by generating small artificial game trees, which can be solved using a standard minimax search with alpha beta pruning, one can then measure the correctness of any random search algorithm given a certain amount of time or iterations.

One form of artificial game tree is called a P-Game tree. A P-game tree [12] is a minimax tree that is meant to model games where at the end of the game the winner is decided by a global evaluation of the board position where some counting method is employed. The final scores in leaf nodes are computed by first assigning values to moves and summing up the values along the path to the terminal state. If the sum is positive, the result is a win for MAX; if it is negative, the result is a win for MIN, whilst it is draw if the sum is 0. For the purposes of our experiments, we modified the version used in [1], where values for the moves of MAX are chosen uniformly from the interval $[0, 127]$ and for the moves of MIN from the interval $[-127, 0]$. Our modifications added support for directed acyclic graph (DAG) based P-Games and groupings. The source code of P-Games and UCT variants are available at http://svn.bchilds.com/mqp/branches/ggame.

[5] We need to update only parent nodes with $N_{s,a}(t) > 0$, which is rather convenient in games where parents are identified only after the move leading to the particular node was sampled.

In the following, each experiment consisted of generating 200 different P-Game trees and running a particular algorithm 200 times on each tree. Thus, there were a total of 40,000 individual results per experiment.

1) Transpositions: In order to support testing on directed acyclic graphs, the tree generation routine was modified in order to create P-Game DAGs where moves at any one level might lead to common nodes on the following level. It should be noted that this results in the generation of a special type of DAG where all of the paths to any given node are of the same length. While this is not strictly the case in Go, it is a reasonably close approximation and only paths involving piece captures in Go may differ from this special case. In order to simulate this in the P-Game we added a parameter to the tree construction that specified the number of nodes at any particular level that should be combined. Using higher values for this parameter, referred to as DAG combination, reduces the total number of nodes at any particular level. This is shown in Figure 1. Comparatively, for a five ply search in Go the search tree is reduced by approximately ten times if transpositions are identified.

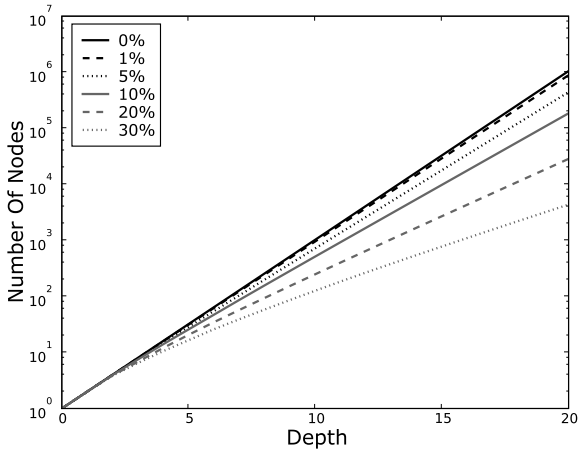

Fig. 1. Graph Size by Level for various DAG combination values

The node combination procedure operates per level of the tree. When generating the next level it selects nodes to combine using the DAG combination bias, the distance between the nodes in the graph (biasing towards closer nodes), and the difference in scores[6] between the two parent nodes. Then it combines the nodes and adds them to the game graph. When two nodes are combined, it selects the maximum score from either of the current nodes and then sets the corresponding values for the moves to the combined node (this is to ensure that moves never provide negative score for the current player). Finally, this procedure repeats until all levels are completed.

The generation procedure for a P-Game DAG is simple but does have some complexity in order to ensure that all paths to any node result in the same overall score. If this were not the case, the algorithms would fail as their base assumption that the game is path-independent would not be upheld.

[6]The score of a node here refers to the sum of values of those moves that lead from the root to the particular node. This score is intended to be independent of the path we are taking.

With this modification we evaluated the performance of UCT1, UCT2, and UCT3 vs. UCT0 with varying levels of DAG combination. Since UCT0 did not use the smaller size of the DAG to its advantage, it treated the DAG as if it were a tree; thusly, UCT1, UCT2, and UCT3 generally outperformed UCT0 in terms of the total number of Monte Carlo simulations required to converge to the correct answer.

The results of one of these experiments is shown in Figure 2. This figure shows the ratio of the average errors for UCT0, UCT1, UCT2 and UCT3 using 20% DAG combination. This particular experiment used trees with branching factor 2, and depth 20. Also shown are error bars of the upper bound of a 95% confidence interval of the average error. In this experiment we notice a significant improvement in performance of UCT1, UCT2 and UCT3 over UCT0. UCT3 has slightly better results then the simpler variants (UCT1 and UCT2), but the time necessary for updating the nodes was much larger. UCT1 and UCT2 had similar time overhead, and UCT2 seems to be slightly better than UCT1, therefore in the remaining experiments we use UCT2 as a good compromise between speed and performance. We note, however, that if the time necessary for simulating games dominates the time overhead of the UCT3 updates, UCT3 can be an interesting choice as well.

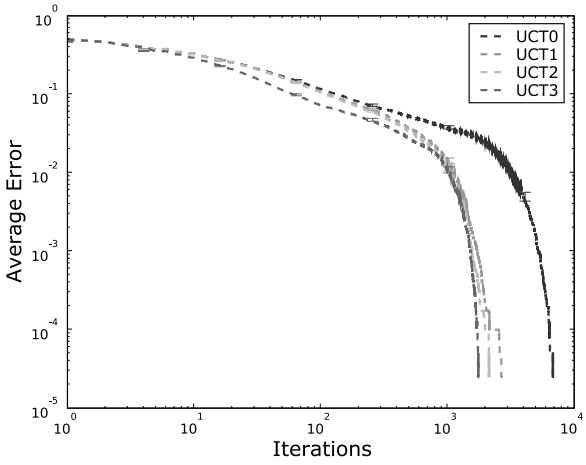

Fig. 2. Average Error vs. Number of Iterations with 20% DAG combination

Following, in Figure 3, are the results of the same experiment but showing the ratio of average error between UCT0 and UCT2 with varying DAG combinations. Predictably, UCT2 shows larger improvements with a higher DAG combination. However, discounting the noise towards the end of the plot, one can also see that UCT2 never performs worse than UCT0. This plot also shows the lower bound of a 95% confidence interval.

As a result of these experiments with UCT2 on artificial game trees, one can be reasonably confident that this modifi-

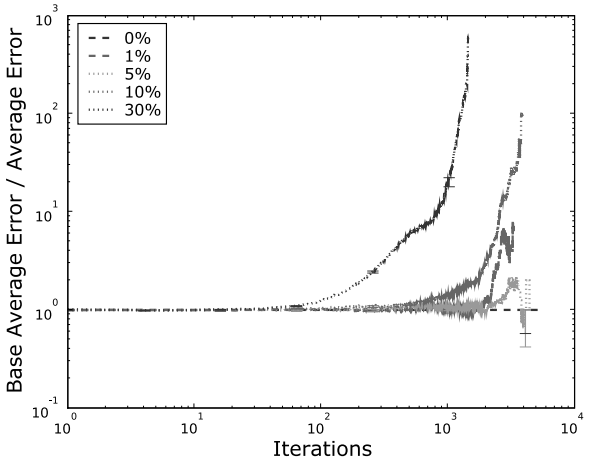

Fig. 3. Base Average Error / Average Error vs. Number of Iterations with varying DAG combination

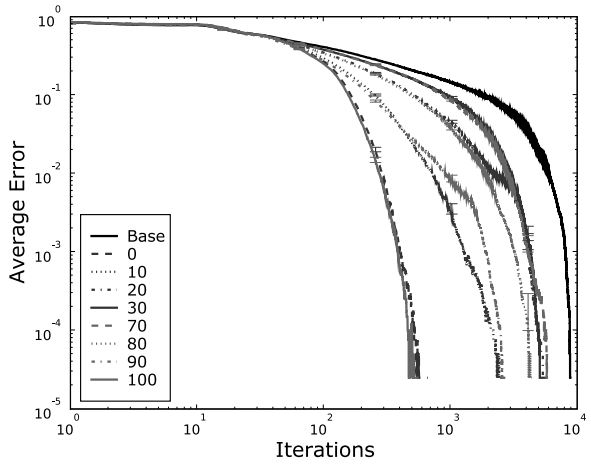

Fig. 4. Average Error vs. Number of Iterations with varying group bias and no overlap

cation can improve performance in situations where a game tree can be better represented as a directed acyclic graph.

2) Groups: The addition of move groups to P-Game required the addition of several parameters to the P-Game generation algorithm. The first parameter, *group bias*, determined how likely it was for a group to contain winning moves vs. losing moves. With a low group bias, losing moves would be more likely to be in a group, whereas with a high group bias, winning moves would be more likely to be grouped. In addition, *group size* and the *number of groups* were added as parameters. Finally, there were two modes of operation, in one groups were disjoint (without overlap), in the other mode groups could overlap and all nodes would be a member of the 'ungrouped' group.

Shown below are the results of two experiments varying the group bias with and without overlap using game trees with branching factor 10 and depth 6. In these experiments there were just two groups: the biased group, and the ungrouped group. With overlap, the ungrouped group included all available moves, while without overlap it included only those moves that were not present in the biased group. The biased group consisted of usually five moves that were selected depending on the bias parameter. The size of this group may be different if there are not enough good (or bad) moves in the node. Without group overlap, group bias performs in a symmetrical manner. Shown in Figure 4, experiments with group bias closer to 100% or 0% perform better than those with group bias closer to 50%. This is predictable as in this case the ungrouped group contains the complement of the grouped group.

With group overlap turned on, however, the algorithm performs even better with group biases closer to 100%, but performs worse with group biases closer to 0%. These results are shown in Figure 5. In this case the ungrouped group contains all of the possible moves. Thus, the algorithm will be biased towards exploring those nodes that are grouped as there will be twice as many paths to such nodes (both through the group nodes and the ungrouped nodes). This important effect of grouping would not be as effective without UCT2 as the multiple paths would all be treated independently. It is also interesting to note that with group overlap enabled, UCT2 converges to the correct answer even more quickly than before.

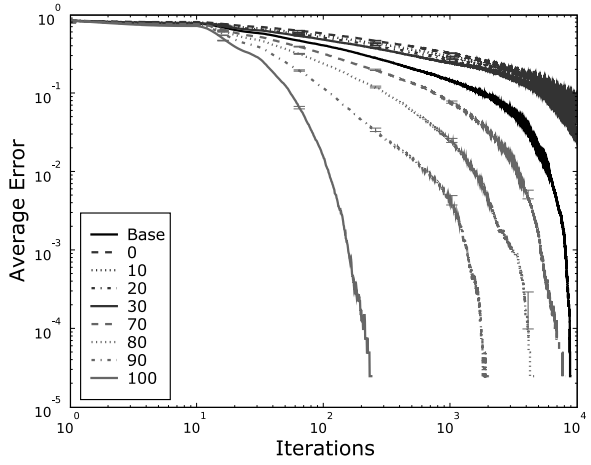

Fig. 5. Average Error vs. Number of Iterations with varying group bias and group overlap

Overall, grouping shows promise with and without overlap. However, this is only the case with sufficiently accurate groupings (for example, groups where a majority of the moves correlate to winning moves). When groups include mostly good moves, having overlap with other groups may help.

B. Go

The most significant results with UCT have been obtained in the game of Go, and therefore we performed some experiments involving the suggested enhancements in Go as

well. We chose to implement the enhancements in an open source Go library, namely libEGO (Library of Effective GO routines) available at [13]. While this implementation is not heavily optimized, we could still measure the enhancements' relative effect on the performance. We leave it to further work testing them in more complete Go programs.

In the experiments, we used two versions of libEGO. The first was the baseline experiment with UCT0. The second was using UCT2 and a Manhattan distance grouping, similar to the one used in [7]. For UCT2, rather than completely re-implementing the UCT algorithm, we kept the tree structure of libEGO, and added a transposition table to detect duplicate nodes in the various parts of the tree. For some number of recently visited board positions, we stored the result of all Monte Carlo simulations performed from that position. In this way the UCT2 algorithm could use the statistics based on board position rather than path through the game tree. The implementation of the Manhattan distance grouping included moves within two intersections of the last two moves. This is slightly different from the Manhattan grouping used in [7] where they would only consider intersections around the single previous move. Moreover, the other group included all legal moves, resulting thus an overlapping grouping scheme. In [7] the groups were disjoint.

We used the freely available GNU Go [14] program as an opponent. While not as strong as the best Go programs, it is one of the standard opponents in computer Go, and due to the un-optimized nature of our implementation it provided sufficient challenge for our purposes. We ran experiments on 9x9 and 13x13 boards. The 9x9 experiments consisted of 1,000 games for each configuration with 20 seconds per move, while the 13x13 experiments, 800 games with 40 seconds per move.

In Figure 6 and Figure 7 we show the results of these experiments along with a 90% confidence interval.

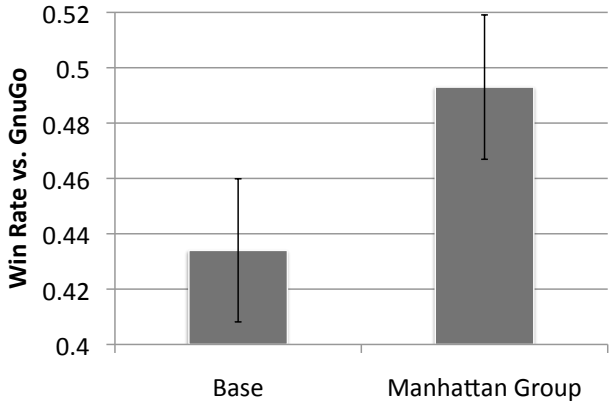

Fig. 6. Results of Go Experiment on 9x9 board with 20 second moves

On the 9x9 board, the Manhattan Grouping shows approximately a 14% improvement over the base algorithm bringing libEgo to almost even footing with GNU Go.

On the 13x13 board the results are quite similar. In fact, UCT2 and grouping show even larger improvements with a 25% higher win rate than the base algorithm.

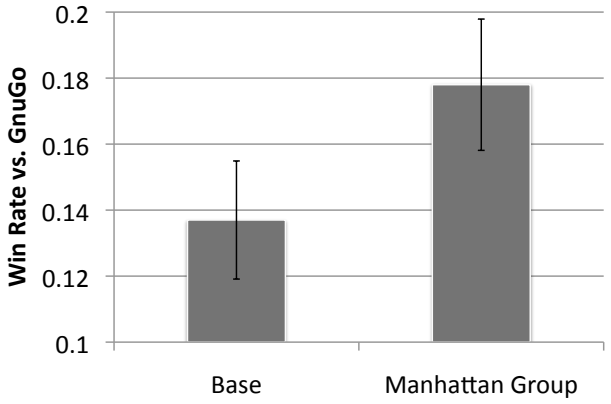

Fig. 7. Results of Go Experiment on 13x13 board with 40 second moves

Overall, the computer Go experiments showed significant gains through the use of grouping and transposition tables. This confirms the results seen for artificial trees, and also the usefulness of grouping moves observed in [7]. Using transposition tables seems a natural choice in Go programs, although it will be interesting to test UCT3 for programs with heavy playouts (i.e. when significant amount of knowledge is used in simulating games). For artificial trees, we noted that groups of mostly good moves are helping even in the presence of overlap. It is a challenging task in Go to identify those groups of moves that correlate to a common fate, yet simultaneously contain enough promising moves to be worth consideration as a group.

V. Conclusions

In this article we discussed two enhancements to the UCT algorithm.

The first enhancement relates to the possible adjustment to UCT when the search tree is treated as a graph. We have outlined three variants (UCT1, UCT2 and UCT3) of the basic algorithm (UCT0) that share information amongst transpositions. These variants have varying implementation complexity and time overhead during the search. In the artificial tree experiments, all three variants had improved performance compared to the basic algorithm. Out of the two 'simpler' variants (UCT1 and UCT2), UCT2 seems to be slightly better. The performance of the more complex UCT3 was slightly better than that of UCT2, and therefore appears as an interesting candidate when the time required for simulating the games dominates the time required for updating the statistics collected in the nodes.

The second enhancement consists of grouping moves, and thus reducing the effective branching factor. Grouping moves without overlap has shown promising results in Go by [7]. Improved performance was also visible in the experiments with artificial trees, and additionally we observed that if most moves in a group have winning scores, then overlap amongst groups do not appear to worsen the performance obtained by grouping. However, having groups of poor moves with

overlapping groups appear to result in worse performance than without grouping. This result may offer a better insight of how to design groups in games like Go. The enhancements were also incorporated in a publicly available Go library, and it resulted in a significant performance increase compared to the original version.

Although both enhancements look promising in the artificial tree and the Go experiments, further work is needed to validate and benchmark these results in full-strength Go programs. It is also interesting how these adjustments to the UCT algorithm could be adapted for other Monte Carlo search algorithms, for instance to BAST [4].

Another possible application would be in parameter optimization (see also [4]). In this case, the parameter space is partitioned hierarchically, and a move in the tree corresponds to selecting a subpartition of the current interval. When splitting an interval, a difficult question is which parameter to split, and where to split. If different partitionings are performed simultaneously, we could be obtaining overlapping intervals that resemble the overlapping move groups; and, it may be possible to similarly share information amongst these interval groups, in a similar way as it is done with transpositions.

REFERENCES

[1] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Proceedings of ECML-06, LNCS/LNAI 4212*, 2006, pp. 282–293.

[2] S. Gelly and D. Silver, "Achieving master level play in 9 x 9 computer go," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, 2008, pp. 1537–1540.

[3] H. Finnsson and Y. Björnsson, "Simulation-based approach to general game playing," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, 2008, pp. 259–264.

[4] P.-A. Coquelin and R. Munos, "Bandit algorithms for tree search," in *Proc. 23rd Conference on Uncertainty in Artificial Intelligence (UAI)*, 2007, pp. 67–74.

[5] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. Schapire, "The non-stochastic multiarmed bandit problem," *SIAM Journal on Computing*, vol. 32, pp. 48–77, 2002.

[6] D. Breuker, J. W. H. M. Uiterwijk, and H. J. van den Herik, "Replacement schemes for transposition tables," *ICCA Journal*, vol. 17, no. 4, pp. 183–193, 1994.

[7] J.-T. Saito, M. H. Winands, J. W. H. M. Uiterwijk, and H. J. van den Herik, "Grouping nodes for Monte-Carlo tree search," in *BNAIC 2007*, 2007, pp. 276–283.

[8] D. M. Breuker, H. J. V. D. Herik, J. W. H. M. Uiterwijk, and L. V. Allis, "A solution to the ghi problem for best-first search," *Theoretical Computer Science*, vol. 252, no. 1–2, pp. 121–149, 2001.

[9] A. Kishimoto and M. Müller, "A general solution to the graph history interaction problem," in *Nineteenth National Conference on Artificial Intelligence (AAAI'04)*, 2004, pp. 644–649.

[10] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, ser. Bradford Book. MIT Press, 1998.

[11] J. Kloetzer, H. Iida, and B. Bouzy, "The Monte-Carlo approach in Amazons," in *Proceedings of the Computer Games Workshop*, 2007, pp. 185–192.

[12] S. Smith and D. Nau, "An analysis of forward pruning," in *AAAI*, 1994, pp. 1386–1391.

[13] L. Lew, "Library for effective go routines." [Online]. Available: http://www.mimuw.edu.pl/~lew/hg/libego

[14] "GNU go." [Online]. Available: http://www.gnu.org/software/gnugo/gnugo.html