# UCD : Upper Confidence bound for rooted Directed acyclic graphs

Abdallah Saffidine and Tristan Cazenave
*LAMSADE*
*Université Paris-Dauphine*
*Paris, France*
*Abdallah.Saffidine@gmail.com cazenave@lamsade.dauphine.fr*

Jean Méhat
*LIASD*
*Université Paris 8*
*Saint-Denis France*
*jm@ai.univ-paris8.fr*

***Abstract*—In this paper we present a framework for testing various algorithms that deal with transpositions in Monte-Carlo Tree Search (MCTS). When using transpositions in MCTS, a Directed Acyclic Graph (DAG) is progressively developed instead of a tree. There are multiple ways to handle the exploration exploitation dilemma when dealing with transpositions. We propose parameterized ways to compute the mean of the child, the playouts of the parent and the playouts of the child. We test the resulting algorithms on LeftRight an abstract single player game and on Hex. For both games, original configurations of our algorithms improve on state of the art algorithms.**

***Keywords*-Monte-Carlo Tree Search; UCT; Transpositions; DAG;**

## I. Introduction

Monte-Carlo Tree Search (MCTS) is a very successful algorithm for multiple complete information games such as Go [1], [2], [3], [4] or Hex [5]. Monte-Carlo programs usually deal with transpositions the *simple way*: they do not modify the UCT formula and develop a DAG instead of a tree.

Transpositions are widely used in combination with the Alpha-Beta algorithm [6] and they are a crucial optimization for games such as Chess. Transpositions are also used in combination with the MCTS algorithm but little work has been done to improve their use or even to show they are useful. The only works we are aware of are the paper by Childs and Kocsis [7] and the paper by Méhat and Cazenave [8].

We will use the following notations for a given object $x$. If $x$ is a node, then $c(x)$ is the set of the edges going out of $x$, similarly if $x$ is an edge and $y$ is its destination, then $c(x) = c(y)$ is the set of the edges going out $y$. We indulge in saying that $c(x)$ is the set of children of $x$ even when $x$ is an edge. If $x$ is an edge and $y$ is its origin, then $b(x) = c(y)$ is the set of edges going out of $y$. $b(x)$ is the set of the "siblings" of $x$ plus $x$. During the backpropagation step, payoffs are cumulatively attached to nodes or edges. We denote by $\mu(x)$ the mean of payoffs attached to $x$ (be it an edge or a node), and by $n(x)$ the number of payoffs attached to $x$. If $x$ is an edge and $y$ is its origin, we denote by $p(x)$ the total number of payoffs the children of $y$ have received: $p(x) = \sum_{e \in c(y)} n(e) = \sum_{e \in b(x)} n(e)$. Let $x$ be a node or an edge, between the apparition of $x$ in the tree and the first apparition of a child of $x$, some payoffs (usually one) are attached to $x$, we denote the mean (resp. the number) of such payoffs by $\mu'(x)$ (resp. $n'(x)$). We denote by $\pi(x)$ the best move in $x$ according to a context dependant policy.

Before having a look at transpositions in the MCTS framework, we first use the notation to express a few remarks on the plain UCT algorithm (when there is no transpositions). The following equalities are either part of the definition of the UCT algorithm or can easily be deduced. The payoffs available at a node or an edge $x$ are exactly those available at the children of $x$ and those that were obtained before the creation of the first child: $n(x) = n'(x) + \sum_{e \in c(x)} n(e)$. The mean of a move is equal to the weighted mean of the means of the children moves and the payoffs carried before creation of the first child:

$$\mu(x) = \frac{\mu'(x) \times n'(x) + \sum_{e \in c(x)} \mu(e) \times n(e)}{n' + \sum_{e \in c(x)} n(e)} \quad (1)$$

The plain UCT value [9] with an exploration constant $c$ giving the score of a node $x$ is written

$$u(x) = \mu(x) + c \times \sqrt{\frac{\log p(x)}{n(x)}} \quad (2)$$

The plain UCT policy consists in selecting the move with the highest UCT formula: $\pi(x) = \max_{e \in c(x)} u(e)$. When enough simulations are run at $x$, the mean of $x$ and the mean of the best child of $x$ are converging towards the same value [9]:

$$\lim_{n(x) \to \infty} \mu(x) = \lim_{n(x) \to \infty} \mu(\pi(x)) \quad (3)$$

Our main contribution consists in providing a parametric formula adapted from the UCT formula 2 so that some transpositions are taken into account. Our framework encompasses the the work presented in [7]. We show that the simple way is often surpassed by other parameter settings on an artificial one player game as well as on the two player game Hex. We do not have a definitive explanation on how parameters influence the playing strength yet. We show that storing aggregations of the payoffs on the edge rather than

on the nodes is preferable from a conceptual point of view and our experiment show that it also often lead to better results.

The rest of this article is organized as follows. We first recall the most common way of handling transpositions in the MCTS context. We study the possible adaptation of the backpropagation mechanism to DAG game trees. We present a parametric framework to define an adapted score and an adapted exploration factor of a move in the game tree. We then show that our framework is general enough to encompass the existing tools for transpositions in MCTS. Finally, experimental results on an artificial single player game and on the two players game Hex are presented.

## II. Motivation

Introducing transpositions in MCTS is challenging for several reasons. First, equation 1 may not hold anymore since the children moves might be simulated through other paths. Second, UCT is based on the principle that the best moves will be chosen more than the other moves and consequently the mean of a node will converge towards the mean of its best child ; having equation 1 holding is not sufficient as demonstrated by figure 2 where equation 3 is not satisfied.

The most common way to deal with transpositions in the MCTS framework, beside ignoring them completely, is what will be referred to in this article as the *simple way*. Each position encountered during the descent corresponds to a unique node. The nodes are stored in hash-table with the key being the hash value of the corresponding position. Mean payoff and number of simulations that traversed a node during the descent are stored in that node. The plain UCT policy is used to select nodes.

The simple way shares more information than ignoring transpositions. Indeed, the score of every playout generated after a given position $a$ is cumulated in the node representing $a$. To the contrary, playouts generated after $a$ when transpositions not detected are divided among all represents of $a$ in the tree depending on the moves that preceded them.

It is desirable to maximize the usage of a given amount of information because it allows to make better informed decisions. In the MCTS context, information is in the form of playouts. If a playout is to be maximally used, it may be necessary to have its payoff available outside of the path it took in the game tree. For instance in figure 3 the information provided by the playouts were only propagated on the edges of the path they took. There is not enough information directly available at $a$ even though a sufficient number of playouts has been run to assert that $b$ is a better position than $c$.

Nevertheless, it is not trivial to share the maximum amount of information. A simple idea is to keep the DAG structure of the underlying graph and to directly propagate the outcome of a playout on every possible ancestor path. It is not always a good idea to do so in a UCT setting, as demonstrated by the counter-example 2. We will further study this idea under the name *update-all* in section III-B.

## III. Possible Adaptations of UCT algorithm to deal with Transpositions

The first requirement of using transpositions is to keep the DAG structure of the partial game tree. The partial game tree is composed of nodes and edges, since we are not concerned with memory issues in this first approach, it is safe to assume that it is easy to access the outgoing edges as well as the in edges of given nodes. When a transposition occurs, the subtree of the involved node is not duplicated. Since we keep the game structure, each possible position corresponds to at most one node in the DAG and each node in the DAG corresponds to exactly one possible position in the game. We will indulge ourselves to identify a node and the corresponding position. We will also continue to call the graph made by the nodes and the moves *game tree* even though it is now a DAG.

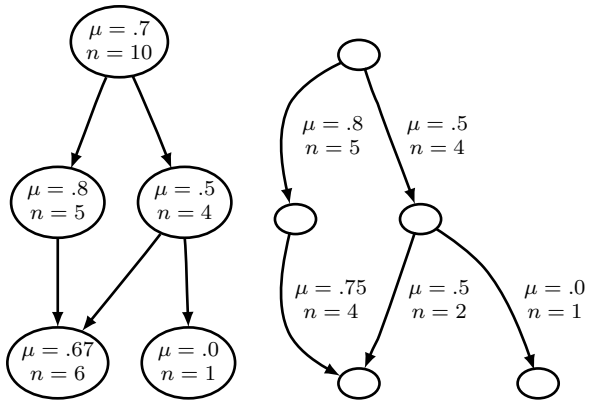
### A. Storing results in the edges rather than in the nodes

In order to descend the game tree, one has to select moves from the root position until reaching an end of the game-tree. The selection uses the results of the previous playouts which need to be attached to moves. A move corresponds exactly to an edge of the game tree, however it is also possible to attach the results to nodes of the game tree. When the game tree is a tree, there is a one to one correspondence between edges and nodes, save for the root node. To each node but the root, correspond a unique parent edge and each edge has of course a unique destination. It is therefore equivalent to attach information to an edge $(a, b)$ or to the destination $b$ of that edge. MCTS implementations seem to prefer attaching information to nodes rather than to edges for implementation simplicity reasons. When the game tree is a DAG, we do not have this one to one correspondence so there may be a difference between attaching information to nodes or to edges.

In the following we will assume that aggregations of the payoffs are attached to the edges of the DAG rather than to the nodes (1 shows the two possibilities for a toy tree). The payoffs of a node $a$ can still be accessed by aggregating[1] the payoffs of the edges arriving in $a$. No edge arrives at the root node but the results at the root node are usually not needed. On the other hand, the payoffs of an edge cannot be easily obtained from the payoffs of its starting node and its ending node, therefore storing the results in the edges is more general than storing the results only in the nodes[2].

[1]The particular aggregation depends on the backpropagation method used (see section III-B): in the update-all case, the data of a node is equivalent to the data of the edge with the biggest number of playouts.

[2]As an implementation note, it is possible to store the aggregations of the edges in the start node provided one associates the relevant move.




(a) Storing the results in the nodes

(b) Storing the results in the edges

Figure 1. Example of the update-descent backpropagation results stored on nodes and on edges for a toy tree.

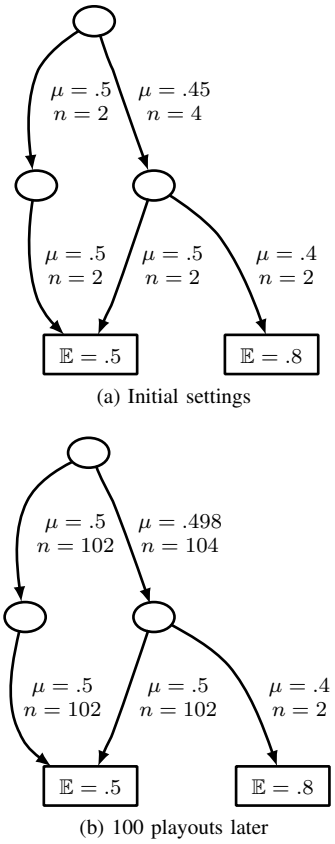
### B. Backpropagation

After the tree was descended and a simulation lead to a payoff, information has to be propagated upwards. When the game tree is a plain tree, the propagation is straightforward. The traversed nodes are exactly the ancestors of the leaf node from which the simulation was performed. The edges to be updated are thus easily accessed and for each edge, one simulation is added to the counter and the total score is updated. Similarly, in the hash-table solution, the traversed edges are stored on a stack and they are updated the same way.

In the general DAG problem however, many distinct algorithms are possible. The ancestor edges are a superset of the traversed edges and it is not clear which need to be updated and if and how the aggregation should be adapted. We will be concerned with three possible ways to deal with the update step: updating every ancestor edge, updating the descent path, updating the ancestor edges but modifying the aggregation of the edge not belonging to the descent path.

Updating every ancestor edge without modifying the aggregation is simple enough, provided one takes care that each edge is not updated more than once after each playout. We call this method *update-all*. Update-all might suffer from deficiencies in schemata like the counter-example presented in figure 2. The problem in update-all made obvious by this counter-example is that the distribution of playouts in the different available branches does not correspond to a distribution as given by UCT: assumption 3 is not satisfied.

The other straightforward method is to update only the traversed edges, we call it *update-descent*. This method is very similar to the standard UCT algorithm implemented on a regular tree and it is used in the simple way. When such a backpropagation is selected, the selection mechanism can be adjusted so that transpositions are taken into account when evaluating a move. The possibilities for the selection mechanism are presented in the following section.




(a) Initial settings

(b) 100 playouts later

Figure 2. Counter-example for the update-all backpropagation procedure. If the initial estimation of the edges is imperfect, the UCT policy combined with the update-all backpropagation procedure is likely to lead to errors

The backpropagation procedure advocated in [7] for their selection procedure UCT3 is also noteworthy. We did not implement it because the same behaviour could be obtained directly with update-descent backpropagation (see section III-C).

### C. Selection

The descent of the game tree can be described as follows. Start from the root node. When in a node $a$, select a move $m$ available in $a$ using a selection procedure. If $m$ corresponds to an edge in the game tree, move along that edge to another node of the tree and repeat. If $m$ does not correspond to an edge in the tree, consider the position $b$ resulting from playing $m$ in $a$. It is possible that $b$ was already encountered and there is a node representing $b$ in the tree, in this case, we have just discovered a transposition, build an edge from $a$ to $b$, move along that edge and repeat the procedure from $b$. Otherwise construct a new node corresponding to $b$ and create an edge between $a$ and $b$, the descent is finished.

The selection process consists in selecting a move that maximizes a given formula. State of the art implementations usually rely on complex formulae that embed heuristics or domain specific knowledge, but the baseline remains the

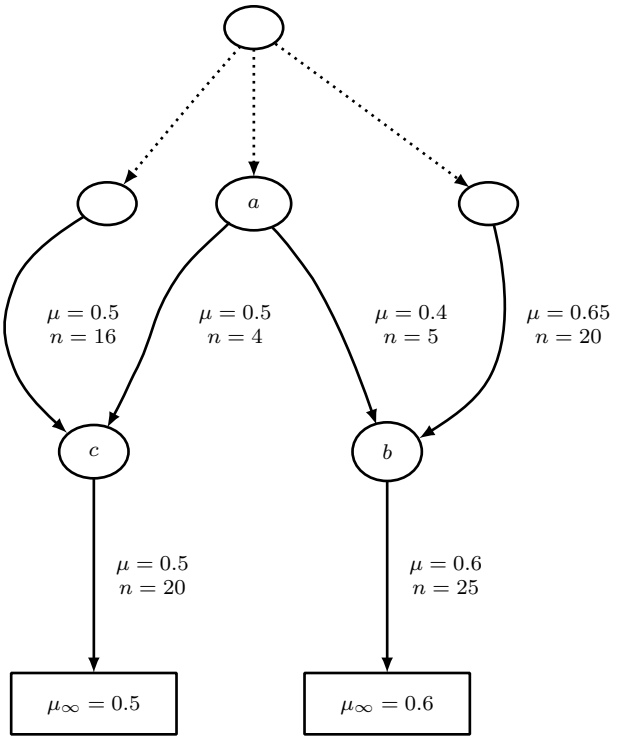



Figure 3. There is enough information in the game tree to know that position $b$ is better than position $c$, but there is not enough local information at node $a$ to make the right decision.

UCT formula[3] defined in equation 2.

When the game tree is a DAG and we use the update-descent backpropagation method, the equation 1 does not hold anymore, so it is not absurd to look for another way of estimating the value of a move than the UCT value. Simply put, equation 1 says that all the needed information is available locally, however deep transpositions can provide useful information that would not be accessible locally.

For instance in the partial game tree in figure 3, it is desirable to use the information provided by the transpositions in node $b$ and $c$ in order to make the right choice at node $a$. The local information in $a$ is not enough to decide confidently between $b$ and $c$, but if we have a look at the outgoing edges of $b$ and $c$ then we will have more information. This example could be adapted so that we would need to look arbitrarily deep to get enough information.

We define a parametric *adapted score* to try to take advantage of the transpositions to gain further insight in the intrinsic value of the move. The adapted score is parameterized by a depth $d$ and is written for an edge $e$ $\mu_d(e)$. $\mu_d(e)$ uses the number of playouts, the mean payoff and the adapted score of the descendants up to depth $d$. The adapted score is given by the following recursive formula.

$$\mu_0(e) = \mu(e)$$
$$\mu_d(e) = \frac{\sum_{f \in c(e)} \mu_{d-1}(f) \times n(f)}{\sum_{f \in c(e)} n(f)}$$

The UCT algorithm uses an exploration factor to balance concentration on promising moves and exploration of less known paths. The exploration factor of an edge tries to quantify the information directly available at it. It does not allow to acknowledge that transpositions occurring after the edge offer additional information to evaluate the quality of a move. So just as we did above with the adapted score, we define a parametric *adapted exploration factor* to replace the exploration factor. Specifically, for an edge $e$, we define a parametric *move exploration* that accounts for the adaptation of the number of payoffs available at edge $e$ and is written $n_d(e)$ and a parametric *origin eploration* that accounts for the adaptation of the total number of payoffs at the origin of $e$ and is written $p_d(e)$. The parameter $d$ also refers to a depth. $n_d(e)$ and $p_d(e)$ are defined by the following formulae.

$$n_0(e) = n(e)$$
$$n_d(e) = \sum_{f \in c(e)} n_{d-1}(f)$$
$$p_d(e) = \sum_{f \in b(e)} n_d(f)$$

In the MCTS algorithm, the tree is built progressively as the simulations are run. So any aggregation of edges built after edge $e$ will lack the information available in $\mu'(e)$ and $n'(e)$. This can lead to a leak of information that becomes more serious as the depth $d$ grows. If we attach $\mu'(e)$ and $n'(e)$ along $\mu(e)$ and $n(e)$ to an edge it is possible to avoid the leak of information and to slightly adapt the above formulae to also take advantage of this information. Another advantage of the following formulation is that is avoids to treat separately edges without any child.

$$\mu_0(e) = \mu(e)$$
$$\mu_d(e) = \frac{\mu'(e) \times n'(e) + \sum_{f \in c(e)} \mu_{d-1}(f) \times n(f)}{n'(e) + \sum_{f \in c(e)} n(f)}$$
$$n_0(e) = n(e)$$
$$n_d(e) = n'(e) + \sum_{f \in c(e)} n_{d-1}(f)$$
$$p_d(e) = \sum_{f \in b(e)} n_d(f)$$

If the height of the partial game tree is bounded by $h$[4], then there is no difference between $d_i = h$ and $d_i = h+x$ for $i \in \{1, 2, 3\}$ and $x \in \mathbb{N}$. When $d_i$ is chosen sufficiently big,

[3]Although these heuristics tend to make the exploration term unnecessary.

[4]for instance if the game cannot last more than $h$ move or if one node is created after each playout and there will not be more than $h$ playouts

we write $d_i = \infty$ to avoid the need to specify any bound. Since the underlying graph of the game tree is acyclic, if $h$ is a bound on the height of an edge $e$ then $h-1$ is a bound on the height of any child of $e$, therefore we can write the following equality which recalls equation 1.

$$\mu_\infty(e) = \frac{\mu'(e) \times n'(e) + \sum_{f \in c(e)} \mu_\infty(f) \times n(f)}{n'(e) + \sum_{f \in c(e)} n(f)}$$

The formulae proposed do not ensure that any playout will not account for more than once in the values of $n_d(e)$ and $p_d(e)$. However a playout can only be counted multiple times if there are transpositions in the subtree starting after $e$. It is not clear to the authors how a transposition in the subtree of $e$ should affect the confidence in the adapted score of $e$. Thus, it is not clear whether such playouts need to be accounted several times or just once. Admitting several accounts gives rise to a simpler formula and was chosen for this reason.

We can now adapt formula 2 to use the adapted score and the adapted exploration to give a value to a move. We define the adapted value of an edge $e$ with parameters $(d_1, d_2, d_3) \in \mathbb{N}^3$ and exploration constant $c$ to be $u_{d_1,d_2,d_3}(e) = \mu_{d_1}(e) + c \times \sqrt{\frac{\log p_{d_2}(e)}{n_{d_3}(e)}}$. The notation $(d_1, d_2, d_3)$ makes it easy to express a few remarks about the framework.

- When no transposition occur in the game, such as when the board state includes the move list, every parameterization gives rise to exactly the same selection behavior which is also that of the plain UCT algorithm.
- The parameterization $(0, 0, 0)$ is not the same as completely ignoring transpositions since each position in the game appears only once in the game tree when we use parameterization $(0, 0, 0)$.
- The simple way (see section II) can be obtained through the $(1, 1, 1)$ parameterization.
- The selection rules in [7] can be obtained through our formalism: UCT1 corresponds to parameterization $(0, 0, 0)$, UCT2 is $(1, 0, 0)$ and UCT3 is $(\infty, 0, 0)$.
- It is possible to adapt the UCT value in almost the same way when the results are stored in the nodes rather than in the edges but it would not be possible to have a parameterization similar to any of $d_1$, $d_2$ or $d_3$ equaling to zero.

## IV. Experimental results

### A. Tests on LeftRight

*LeftRight* is an artificial one player game already used in [10] under the name "left move", at each step the player is asked to chose to move Left or to move Right ; after a given number of steps the score of the player is the number of steps walked towards Left. A position is uniquely determined by the number of steps made towards Left and the total number of moves played so far, transitions are therefore very frequent[5].

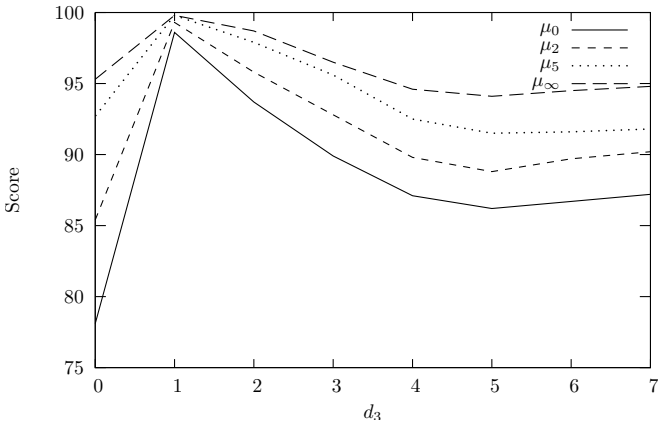



Figure 4. LeftRight results.

We used 300 moves long games for our tests. Each test was run 200 times and the standard error is never over 0.3% on the following scores.

The UCT algorithm performs well at LeftRight so the number of simulations had to be low enough to get any differentiating result. We decided to run 100 playouts per move. The plain UCT algorithm without detection of transpositions with an exploration constant of 0.3 performs 81.5 %, that is in average 243.5 moves out of 300 were Left. We also tested the update-all backpropagation algorithm which scored 77.7 %. We tested different values for all three parameters but the scores almost did not evolve with $d_2$ so for the sake of clarity we present results with $d_2$ set to 0 in figure 4.

The best score was 99.8% with the parameterization $(\infty, 0, 1)$ which basically means that in average less than one move was played to the Right in each game. Setting $d_3$ to 1 generally constituted a huge improvement. Raising $d_1$ was consistently improving the score obtained, eventually culminating with $d_1 = \infty$.

### B. Tests on Hex

*Hex* is two-player zero sum game that cannot end in a draw. Every game will end after at most a certain number of moves and can be labeled as a win for Black or as a win for White. Rules and details about Hex can be found in [11]. Various board sizes are possible, sizes from 1 to 8 have been computer solved [12]. Transpositions happen frequently in Hex because a position is completely defined by the sets of moves each player played, the particular order that occurred before has no influence on the position. MCTS is quite successful in Hex [5], hence Hex can serve as a good experimentation ground to test our parametric algorithms.

[5]if there are $h$ steps the full game tree has only $\frac{h \times (h-1)}{2}$ nodes if transpositions are recognized but $2^h$ nodes otherwise

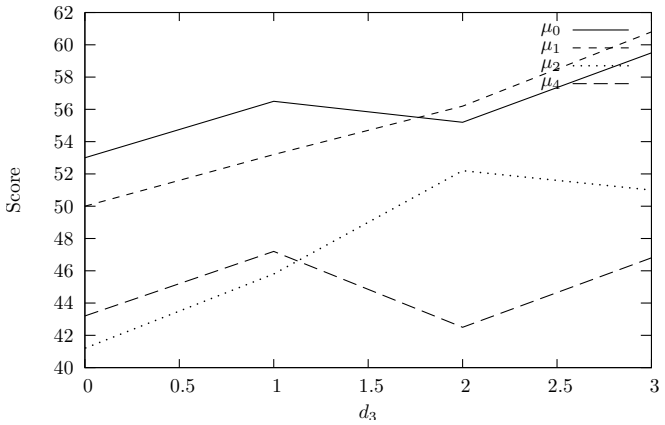



Figure 5. Hex results with $d_2$ set to 0

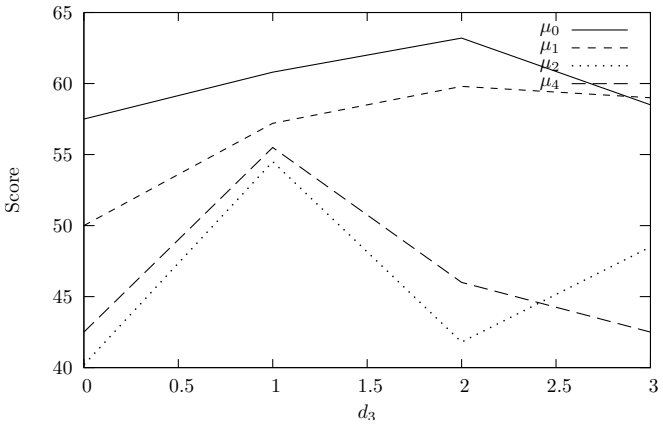



Figure 6. Hex results with $d_2$ set to 1

Hex offers a strong advantage to the first player and it is common practice to balance a game with a compulsory mediocre first move[6]. We used a size 5 board with an initial stone on *b2*. Each test was a 400 games match between the parameterization to be tested and a standard A.I. In each test, the standard A.I. played Black on 200 games and White on the remaining 200 games. The reported score designates the average number of games won by a parameterization. The standard error was never over 2.5%.

The standard A.I. used the plain UCT algorithm with an exploration constant of 0.3, it did not detect transpositions and it could perform 1,000 playouts at each move. We also ran a similar 400 games match between the standard A.I. and an implementation of the update-all backpropagation algorithm with an exploration constant of 0.3 and 1,000 playouts per move. The update-all algorithm scored 51.5% which means that it won 206 games out of 400. The parameterization to be tested also used a 0.3 exploration constant and 1,000 playouts at each move. The results are presented in figure 5 for $d_2$ set to 0 and in figure 6 for $d_2$ set to 1.

The best score was 63.5 % with the parameterization $(0, 1, 2)$. It seems that setting $d_1$ as low as possible might improve the results, indeed with $d_1 = 0$ the scores were consistently over 53% while having $d_1 = 1$ led to having scores between 48% and 62%. Setting $d_1 = 0$ is only possible when the payoffs are stored per edge instead of per node as discussed in section III-A.

## V. Conclusion and Future Work

We have presented a parametric algorithm to deal with transpositions in MCTS. Different parameters did improve on usual MCTS algorithms for two games: LeftRight and Hex.

[6]Even more common is the swap rule or pie-rule.

In this paper we did not deal with the graph history interaction problem [13]. In some games the problem occurs and we might adapt the MCTS algorithm to deal with it.

We have defined a parameterized value for moves that integrates the information provided by some relevant transpositions. The distributions of the values for the available moves at some nodes do not necessarily correspond to a UCT distribution. An interesting continuation of our work would be to define an alternative parametric adapted score so that the arising distributions would still correspond to UCT distributions.

Another possibility to take into account the information provided by the transpositions is to treat them as contextual side information. This information can be integrated in the value using the Rapid Action Value Estimation (RAVE) formula [14], or to use the episode context framework described in [15].

## References

[1] R. Coulom, "Efficient selectivity and back-up operators in monte-carlo tree search," in *Computers and Games 2006*, ser. Volume 4630 of LNCS. Torino, Italy: Springer, 2006, pp. 72–83.

[2] ——, "Computing Elo ratings of move patterns in the game of Go," *ICGA Journal*, vol. 30, no. 4, pp. 198–208, December 2007. [Online]. Available: http://remi.coulom.free.fr/Amsterdam2007/MMGoPatterns.pdf

[3] S. Gelly and D. Silver, "Achieving master level play in 9 x 9 computer go," in *AAAI*, 2008, pp. 1537–1540.

[4] G. Chaslot, L. Chatriot, C. Fiter, S. Gelly, J.-B. Hoock, J. Perez, A. Rimmel, and O. Teytaud, "Combiner connaissances expertes, hors-ligne, transientes et en ligne pour l'exploration Monte-Carlo. Apprentissage et MC," *Revue d'Intelligence Artificielle*, vol. 23, no. 2-3, pp. 203–220, 2009.

[5] T. Cazenave and A. Saffidine, "Utilisation de la recherche arborescente Monte-Carlo au Hex," *Revue d'Intelligence Artificielle*, vol. 23, no. 2-3, pp. 183–202, 2009.

[6] D. Breuker, "Memory versus search in games," Universiteit Maastricht, PhD thesis, 1998.

[7] B. E. Childs, J. H. Brodeur, and L. Kocsis, "Transpositions and move groups in monte carlo tree search," in *CIG-08*, 2008, pp. 389–395.

[8] J. Méhat and T. Cazenave, "Combining UCT and nested Monte-Carlo search for single-player general game playing," *to appear*, 2010.

[9] L. Kocsis and C. Szepesvàri, "Bandit based monte-carlo planning," in *ECML*, ser. Lecture Notes in Computer Science, vol. 4212. Springer, 2006, pp. 282–293.

[10] T. Cazenave, "Nested monte-carlo search," in *IJCAI*, 2009, pp. 456–461.

[11] C. Browne, *Hex Strategy: Making the Right Connections*, A. K. Peters, Ed. MA: Natick, 2000.

[12] P. Henderson, B. Arneson, and R. B. Hayward, "Solving 8x8 Hex," in *IJCAI*, C. Boutilier, Ed., 2009, pp. 505–510.

[13] A. Kishimoto and M. Müller, "A general solution to the graph history interaction problem," in *AAAI*, 2004, pp. 644–649.

[14] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," in *ICML*, 2007, pp. 273–280.

[15] C. D. Rosin, "Multi-armed bandits with episode context," in *Proceedings ISAIM*, 2010.