Combining UCT and Nested Monte-Carlo Search for Single-Player General Game Playing

Jean Méhat and Tristan Cazenave

Abstract—Monte-Carlo tree search has recently been very successful for game playing particularly for games where the evaluation of a state is difficult to compute, such as Go or General Games. We compare Nested Monte-Carlo Search (NMC), Upper Confidence bounds for Trees (UCT-T), UCT with transposition tables (UCT+T) and a simple combination of NMC and UCT+T (MAX) on single-player games of the past GGP competitions. We show that transposition tables improve UCT and that MAX is the best of these four algorithms. Using UCT+T, the program Ary won the 2009 GGP competition. MAX and NMC are slight improvements over this 2009 version.

Index Terms—General Game Playing, UCT, Nested Monte-Carlo Search, Single-player games.

1 Introduction

General Game Playing (GGP) tries to address the shortcomings of current specialized game playing programs that cannot adapt to other domains than the game they were programmed for. The goal is to find general algorithms for games, or to automatically classify games according to the algorithms that play them well. In GGP the programs are required to have more general intelligence than game-specific programs.

Programs play games they have never seen before. They are given the rules of the game, and after an initial autonomous analysis performed in a predefined time, they play the game. A specificity of GGP is that programs are given the rules of the game explicitly, in the Game Description Language (GDL), hence they can analyze and manipulate these rules.

GGP addresses single-player, two-player and multi-player games with simultaneous or turn taking moves. Games may be zero sum or not. GGP does not currently address incomplete information games and infinite games. The Stanford Logic Group organizes an annual GGP competition between GGP players at the AAAI conference.

In this paper we focus on Monte-Carlo Tree Search algorithms. The basis of Monte-Carlo Tree Search is to play random games called playouts. Using the results of the previous playouts in order to improve the future playouts was shown by Rémi Coulom to be very efficient in the game of Go [10]. A simple and related algorithm named UCT was published the same year [16] and is now widely used as a basis for Monte-Carlo Tree Search algorithms, even though some of the best programs do not use the UCT formula anymore [17], [6], [7]. Improvements on the basic UCT algorithm came from learning patterns to direct the search and to bias the playouts [11] and from biasing the choice of moves in the tree with statistics on the results of the previous playouts where a move was played [13]. Building on these successes in two-player games, Monte-Carlo Tree Search algorithm were also recently used in single-player games [4], [20], [5]. In single-player General Game Playing, other approaches such as Answer Set Programming are also possible [22].

In this paper we address single player General Game Playing. The second section is about the Game Description Language. The third section deals with UCT. The fourth section

- *J. Méhat is at LIASD, Université Paris 8, Saint-Denis, France. E-mail: jm@ai.univ-paris8.fr*
- *T. Cazenave is at LAMSADE, Université Paris-Dauphine, 75016 Paris, France. E-mail: cazenave@lamsade.dauphine.fr*

TABLE 1
The GDL representation of simultaneous play, binary version of the game *My father has more money that yours*.

```
(ROLE left) (ROLE right)       ; players
(INIT (played no))        ; initial
(LEGAL (DOES ?player (tell 0))); moves
(LEGAL (DOES ?player (tell 1)))
(<= (NEXT (value ?p ?x))
    (DOES ?p (tell ?x)))
(<= (NEXT (played yes)))
(<= TERMINAL (TRUE (played yes)))
(<= (other ?x ?y) (role ?x) (role ?y)
    (DISTINCT ?x ?y))
(<= (GOAL ?p 0) (TRUE (value ?p 0))
    (other ?p ?op) (TRUE (value ?op 1)))
(<= (GOAL ?p 50) (TRUE (value ?p ?x))
    (other ?p ?op) (TRUE (value ?op ?x)))
(<= (GOAL ?p 100) (TRUE (value ?p 1))
    (other ?p ?op) (TRUE (value ?op 0)))
```

deals with Nested Monte-Carlo Search. The fifth section details the structure of Ary. The sixth section details experimental results.

2 The Game Description Language

The Game Description Language (GDL) [19] is used to describe a game. It is based on first order logic, hence cannot easily express arithmetic computations.

The rules indicate the players with `ROLE`, the initial state with INIT, the legal moves with `LEGAL`, the effects of moves with `NEXT`, and the reward with `GOAL`. The game is ended when `TERMINAL` is provable.

We give in table 1 a very simple example in GDL of a binary version of the simultaneous play game *My father has more money than yours* [2]: each player names a figure, here 0 or 1 and the winner is the player naming the bigger number. Keywords of GDL are written in upper case.

In the following, we call the current situation of the game the *board status*, even for games that are not played on a board like the one described in table 1.

3 The UCT algorithm

The basic idea of UCT [16] is to add to Monte-Carlo explorations of the abstract move tree an informed way to choose the branches that will be explored. A move tree is constructed incrementally, with a new node added for each Monte-Carlo exploration. On the next exploration, a path is chosen in the already built move tree by choosing the branch whose estimated gain is maximum. This gain is estimated by the mean result of the previous Monte-Carlo explorations plus the Upper Confidence Bound in the estimation. The Upper Confidence Bound is calculated by a function of the number of explorations of the node t and of the number of exploration of the branch s as $\sqrt{\frac{\log(t)}{s}}$. When arriving at a leaf node of the move tree, if it is not a terminal state, a new node is added to the tree and a Monte-Carlo simulation is started to obtain an evaluation of this node and update the evaluation of its parent nodes.

UCT is used with success as a basis for Monte-Carlo Go programs [10], [14], [17] and to many other games. Different ways to use a transposition table with UCT were investigated in [8]. UCT was also successfully applied to the single-player game SameGame [20].

When there are some unexplored moves, UCT will choose to explore them. When all the branches from a node have been explored, UCT will tend to re-explore the most promising ones: this tendency is controlled by a constant C, that is used to multiply the Upper Confidence Bound $\sqrt{\frac{\log(t)}{s}}$. The exact formula used for UCT is $m + C \times \sqrt{\frac{\log(t)}{s}}$ where m is the mean of the playouts under the node. The higher the constant C, the more UCT will explore unpromising nodes. At the limit, when the whole game tree has been explored, UCT is favoring the better branches and tend to converge to the same choice as a Minimax exploration.

3.1 Representation of the UCT move tree

The game tree is stored as a set of nodes. Each node contains the current situation, the legal moves for each player when the node is not terminal, an evaluation of the gain of each

player and the number of explorations of this node.

At the beginning of the match, the root node is seeded with the initial situation.

3.2 UCT path selection

For each exploration, a path is followed from the root of the game tree with UCT, until arriving either at the root of a fully explored subtree or at a node where some children have not yet been tried at least once. For multiplayer games, we use a UCT constant of 40 while for one player games the larger value 100 is used to favor exploration. In GGP, scores range from 0 to 100. These constants were experimentally determined: they are the ones that gave the best mean outcomes out of many experiments on many different games. Trying to adapt the constant to the game after an analysis of the game properties did not give satisfactory results.

When a node is terminal, the score of each player is stored in the node. When the selected node is the root of a fully explored subtree, a value is computed using the maximum value for the player among the children. This value is exact for sequential moves games. The selected move and its exact reward are stored in the node for use by future playouts.

When the selected node has at least one unexplored edge, the tree explorator selects pseudo-randomly one of these unexplored arcs, applies its moves to the game situation described by the node, obtaining a new game situation. It then searches for a transposition: in the existing nodes, if one does contain the same situation, it is a transposition: an edge is added from the selected node to the newly found one and the descent in the tree proceeds from there.

The known advantages of the transposition tables in move tree exploration work well for GGP: the nodes can use a large quantity of memory and the moves generation and application via the interpretation of GDL are costly. Using the transposition tables helps reducing both the number of nodes and the number of move generations and applications. It also protects somewhat the program against game descriptions that would offer many different but completely equivalent moves. In GDL the current situation is fully described by the literals and in GGP the games are required not to have cycles so the transposition table is free of the graph history interaction problem.

There are multiple ways to implement a transposition table with UCT [8]. We have chosen to store the results of the playout and the number of visits in the nodes and not in the edges between nodes. We update only the nodes visited during the playout and not all the parents of a node. The number of visits of a parent node used in the UCT formula is obtained summing the number of visits of all its children. This way, when a children has been visited many times by other parents, the UCT formula will drive the exploration towards less explored children.

If no transposition is found, a new node is built. If it is terminal the gain of each player is computed via GDL interpretation. For non terminal nodes, a playout is run, giving an initial evaluation.

In single-player games UCT memorizes the best playout and its score. When the program has to send back a move, it selects the first move of this playout.

3.3 Update of nodes values

Once an evaluation is obtained, either by reaching a terminal node or the root of an explored subtree, or with a playout, the estimated value of each node traversed during the descent is updated and the number of explorations of these nodes is incremented. The value is updated computing the mean results of the playouts that went through the node (i.e. dividing the sum of the results of the playouts by the number of explorations).

The transposition tables are not used in this step: the values are updated only in the parent node used in this playout. Doing otherwise could lead UCT to avoid some nodes as explained in Figure 1.

3.4 Update of the move tree after player moves

When our program receives a notification of the moves from the Game Master, it sets the child

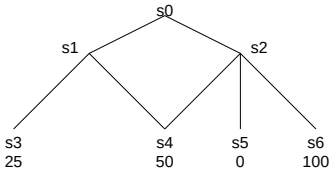

Fig. 1. This simple one player game illustrates a situation where updating the reward estimation in all the parents of a node could induce UCT in error. If all the nodes are visited before s_6, UCT will favor the branch s_0s_1 over s_0s_2. If both s_4 parents were updated, the number of experiments in s_1 and s_2 would grow simultaneously and the branch $s_0s_2s_6$ would not be explored.

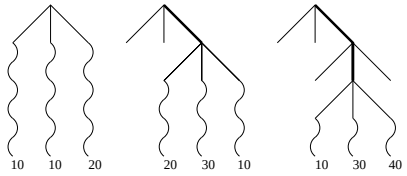

Fig. 2. This figure explains three steps of a level n search. At each step of the playout of level n shown here with a bold line, an NMC of level n performs an NMC of level $n-1$ (shown with wavy lines) for each available move and selects the best one. At level 0, a simple pseudo-random playout is used. For example in the third step on the tree to the right of the figure, the move that starts the playout with result 40 will be chosen and played at level n.

from this branch as the new root. If the child does not already exist in the tree, it is computed and created. Then all nodes unreachable from this new root are discarded.

4 The Nested Monte-Carlo Search algorithm

The basic idea of Nested Monte-Carlo (NMC) is to perform a principal playout with a bias on the selection of each move based on the results of a Monte-Carlo tree search [5].

Figure 2 illustrates a level n Nested Monte-Carlo search. Three selections of moves at level n are shown. The leftmost tree shows that Nested Monte-Carlo searches of level $n-1$ starting with each legal move are performed. The rightmost move has the best result of 20, therefore this is the first move played at level n in the middle tree. After this first move, Nested Monte-Carlo searches of level $n-1$ are again performed for each possible move following the first move. One of the moves has result 30 which is the best playout result among his siblings. So the game continues with this move as shown in the rightmost tree.

The algorithm for higher levels is algorithm 1. At each move of a playout of level 1 it chooses the move that gives the best score when followed by a single random playout. Similarly for a playout of level n it chooses the move that gives the best score when followed by a playout of level $n-1$.

For a tree of height h and branching factor a, the total number of playout steps of a NMC of level n will be $t_n(h,a) = a \times \sum_{0<i<h} t_{n-1}(i,a)$ with $t_0(h,n) = h$. So a NMC of level 1 will perform $a \times h^2/2$ playout steps. The complexity of a NMC of level n is $O(a^n h^{n+1})$.

Algorithm 1 Nested Monte-Carlo search

```
nested (position, level)
best playout ← {}
while not end of game do
  if level = 1 then
    move ← argmax_m(sample (play (position, m)))
  else
    move ← argmax_m(nested (play (position, m), level − 1))
  end if
  if score of playout after move > score of the best playout then
    best playout ← playout after move
  end if
  position ← play (position, move of the best playout)
end while
return score (position)
```

Searches at the highest level are repeatedly performed until the thinking time is elapsed.

Nested Monte-Carlo search has been successful in establishing world records in single player games such as Morpion Solitaire or SameGame. It provides a good balance between exploration and exploitation and it automatically adapts its search behavior to the

problem at hand without parameters tuning.

When a Monte-Carlo engine is available implementing Nested Monte-Carlo search is simple and straightforward; in our program, it is done in about 200 lines of C. NMC makes a good compromise between exploration and exploitation on many games. At each level, it is important to memorize the best playout found so far in order to play its moves if no better playout is found.

5 The structure of Ary

In this section we present Ary, the program used for the experiments.

Ary is written in C. It uses Monte-Carlo Tree Search (MCTS) and a Prolog interpreter as an inference engine. Ary won the 2009 GGP competition.

Its architecture is somewhat similar to the one of CadiaPlayer [12], [3] since it also uses UCT and Prolog, however it implements transposition tables and NMC. It is different from the winners of the 2005 and 2006 competitions, ClunePlayer and FluxPlayer [9], [21] that used evaluation functions.

5.1 Data structures used in Ary

The basic objects are atoms, used to represent all the constituents of the game description: keywords of GDL, operators of the logic, predicates, names, integers and variables used in the description of the game. On its creation, each atom receives a random 64 bits hash key.

The basic objects are combined in lists, represented with a cell containing two pointers on the head and tail of the list. Each cell has also a hash code: the hashcode of the head is shifted and xored with the hashcode of the tail. This hashcode is used to compare quickly lists that differ (elements of identical lists have to be compared, as different lists may have the same hash code) and find them in hash tables. We did not implement garbage collection: the core of the program identifies lists that will not be referenced and adds them explicitly to the free list.

Most of the lists are stored in list tables. The table contains an array and a hash table. In the array, the elements are stored sequentially, giving an easy way to iterate over all the elements. The hash table allows to quickly verify if an element is already present in the table. When a table is full, it is reallocated with a greater number of slots.

A *node* data structure is used to represent the move tree built by the UCT algorithm, as described in the section on UCT.

5.2 Interface with Prolog

We use a Prolog Interpreter as an inference engine. We currently use YAP Prolog or SWI Prolog because of their availability, speed and good interface with the C language.

After loading, the game is translated from GDL to Prolog. Similarly, we have a function that is used to translate back the answers of the Prolog interpreter into the internal form.

The game description in Prolog is sent once for all to the interpreter, with the initial board status. The Prolog interpreter is then used to identify terminated games, find the scores of the players in these states, enumerate the legal moves and the consequences of moves on the board status, and modify the image of the board status in the interpreter.

The transition from one board status to another is done incrementally: Ary retracts what is present only in the old state, and asserts what appears only in the new state. Unmodified characteristics of the board status do not imply exchange with the Prolog interpreter. So in games like Tic Tac Toe where a move modifies the status of one cell described by one assertion, the transition from one board status to the next is done with one retraction and one assertion.

6 Experimental results

We tested Ary on all the single player games of the competitions of the last three years that were available on the Dresden GGP server [1]. We conducted two different sets of experiments with the following algorithms: UCT without transposition tables (UCT-T), UCT with transposition tables (UCT+T) and Nested Monte-Carlo Search at level 1 (NMC).

TABLE 2
The average scores of four Monte-Carlo tree search algorithms on twenty single player games from the GGP competitions

	MAX2		MAX1		NMC		UCT+T		UCT-T		# experiences
asteroidsparallel	71	±9	67	±11	68	±10	60	±12	59	±12	156/233/157/157/157/
asteroidsserial	85	±12	84	±12	85	±12	87	±12	86	±12	157/233/157/157/157/
duplicatestatelarge	67	±11	67	±8	51	±9	62	±17	42	±10	156/231/157/157/157/
duplicatestatemedium	95	±5	96	±5	84	±11	93	±12	53	±14	156/233/156/156/156/
hanoi	80	±0	80	±0	80	±0	80	±0	80	±2	156/232/156/156/156/
hanoi7 bugfix	70	±9	70	±9	58	±5	70	±10	49	±9	156/231/156/156/156/
incredible	75	±11	74	±15	71	±14	73	±11	50	±11	156/232/156/156/156/
knightmove	97	±4	97	±4	97	±4	95	±5	95	±5	157/232/157/157/156/
knightstour	100	±0	99	±1	96	±3	100	±1	100	±1	157/232/157/157/157/
lightsout	6	±23	2	±13	2	±13	3	±15	4	±19	155/232/156/156/156/
lightsout2	8	±26	3	±17	1	±11	5	±21	4	±20	157/232/157/157/157/
max knights	95	±8	87	±13	95	±9	68	±18	67	±18	156/232/642/619/620/
pancakes6	89	±1	89	±1	81	±4	89	±1	87	±3	156/232/156/156/156/
pancakes88	59	±9	54	±7	50	±4	61	±9	57	±9	157/231/157/157/157/
peg bugfixed	92	±3	91	±2	91	±2	90	±3	90	±3	156/232/156/156/156/
queens	76	±9	74	±9	79	±8	68	±10	70	±11	157/232/157/157/157/
statespacelarge	52	±8	50	±9	51	±8	42	±10	42	±11	156/232/156/156/156/
statespacemedium	84	±9	81	±11	83	±11	52	±12	50	±10	157/232/157/157/157/
sudoku simple	40	±7	36	±7	39	±7	32	±7	32	±7	157/231/530/467/467/
tpeg	99	±3	97	±4	98	±3	95	±7	94	±6	156/232/156/156/156/
total	1440		1398		1360		1325		1211		

We added a fourth algorithm, MAX, a combination of UCT+T and NMC. For each move, both of the algorithms which make up MAX evaluate independently the position and provide their evaluation of the best move found. The move with the best associated score is then transmitted to the Game Master. Both algorithms were run in two separate processes on the same multicore machine used for the other experiments, effectively competing for memory (NMC only uses very little memory and UCT stops creating nodes when memory is filled).

MAX was run in two configurations:

- in MAX1, both processes ran for half of the time given to NMC and UCT, so MAX1 had the same total amount of CPU time than NMC or UCT.
- in MAX2, both processes ran for the same time as NMC and UCT, giving a fair estimation of what can be expected on a dual core machine under real tournament conditions.

The version of Ary that won the 2009 competition at IJCAI used UCT+T. We added some code to handle UCT-T, NMC and MAX so as to compare with the same underlying GGP engine.

The MAX approach is basically a portfolio approach [15], [18] which is becoming very popular in many areas such as SAT [23] and planning.

All tests were run on a multicore PC at 2.33 GHz with 4 Gigabytes of memory, running Linux 2.6.28. In the MAX experiments, two cores were used independently by NMC and UCT. In all others experiments only one core was used by the programs. It is the same machine that we used during the 2009 competition.

6.1 Experiments under tournament-like conditions

The first set of experiments consists in testing the algorithms under realistic time limits for competition conditions. The initial thinking time is set to ten seconds, then the program is given ten seconds for each move. The results are given in table 2.

This table contains only the games that are not too hard nor too simple for Monte-Carlo tree search (either NMC or UCT). In games such as maze for example, MCTS always scores 100, so these games are not included in the table. On other games such as 8puzzle for ex-

ample, MCTS always scores 0, so these games are not included in the table either. On these games it is a property of Monte-Carlo Tree Search algorithms to score either 0 or 100 since UCT-T, UCT+T, NMC and MAX all give the same results. The twenty games that are not too hard and not too easy are given in table 2, with the results of NMC, UCT-T, UCT+T, MAX1 and MAX2.

On some games such as max_knights, the standard error is high even if many games have been played. Figure 3 gives the number of occurrences of the scores for max_knights. UCT+T and UCT-T behave similarly with three local maxima at 50, 75 and 100. NMC and MAX2 have a maximum at 100 and a smaller local maximum at 75. The distribution of the scores for UCT+T and UCT-T explains the large standard error (18) of these algorithms at max_knights. Max_knights consists in placing as many knights as possible on an 8x8 chess board. As soon as a knight can capture another one, the game is over. The score is based on the number of placed knights. An optimal solution consists in placing all the knights on squares of the same color. The distribution of the scores is due to the structure of the game: bad moves at the beginning cannot be recovered and lead to a local maximum.

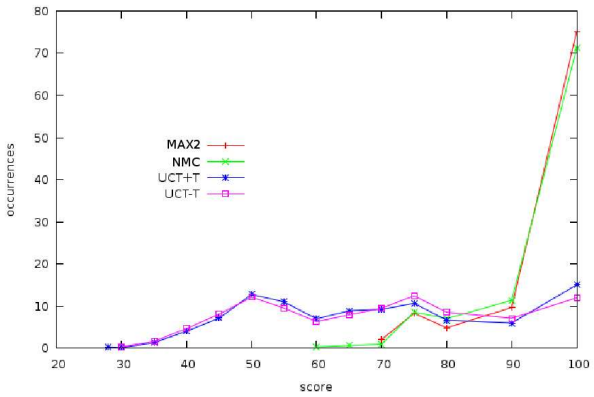

Fig. 3. Percentage of occurrence of scores for the four algorithms at max_knights.

Particular games are lightsout and lightsout2. The only possible scores are 0 and 100. All the algorithms found a few solutions, and the associated error is very high. It seems nonetheless that the better scores of UCT are due to the reuse of the previous explorations in the UCT tree, that speeds up the explorations, and MAX2 advantage is due to the still greater number of explorations as it uses twice more CPU time.

The first conclusion we can draw from table 2 is that UCT+T is equivalent or slightly better than UCT-T on almost all of the games. It is clearly better at duplicatestatelarge (62 vs. 42), duplicatestatemedium (93 vs. 53), hanoi7_bugfix (70 vs. 49) and incredible (73 vs. 50). Concerning duplicatestatelarge and duplicatestatemedium, it is not surprising that UCT+T is better because detecting duplicate states is an important property of these games.

The second conclusion is that NMC has overall results that are better than UCT-T and slightly better than UCT+T. However the results differ according to the games. In some games it is clearly better (statespacemedium, max_knights, statespacelarge) and in other clearly worse (duplicatestatelarge, pancakes6, hanoi7_bugfix). NMC is better than UCT+T at asteroidsparallel (68 vs 60), sudoku_simple (39 vs 32), statespacemedium (83 vs 51), queens (79 vs 68), max_knights (95 vs 68), statespacelarge (51 vs 42). An explanation concerning max_knights is that it behaves like a constraint satisfaction problem. The way it is defined in GGP is that once a losing move is played the game is over. UCT does not avoid direct losing moves in its playouts while NMC naturally avoids them. In queens the bad moves are also avoided due to the nested playouts. This could be an explanation of why NMC is better than UCT in these games. In five other games UCT+T gets better results than NMC: duplicatestatelarge (62 vs 52), duplicatestatemedium (93 vs 84) pancakes88 (61 vs 51), pancakes6 (89 vs 81), hanoi7_bugfix (70 vs 58). Concerning duplicatestatelarge and duplicatestatemedium this is clearly due to the detection of duplicate states. On the other eight games NMC and UCT+T are more or less equivalent.

Despite having less CPU time for each algorithm, MAX1 almost always scores better than the worst of UCT+T and NMC. It is usually close to the best of UCT+T and NMC on most of the games. Overall it is slightly better than both NMC and UCT+T.

MAX2 gets similar or slightly better results than MAX1 thanks to its additional CPU time. The benefit of using more CPU time is particularly clear for lightsout and lightsout2. In these games that are hard for Monte-Carlo Tree Search, the doubling of CPU time increases the chances of randomly finding a solution.

In addition to the increase in score due to the chances of randomly finding a solution, the score also increases due to the mix of NMC and UCT+T at each move. If we take for each game the best score between UCT+T and NMC, the scores sum to 1424. This is better than the score of MAX1 (1398) but worse than the score of MAX2 (1440). What we see here is that calling UCT+T and NMC at each move of the game and choosing the best is better than running them completely separate.

6.2 Experiments on the evolution of the evaluation with time

The first set of experiments evaluates the algorithms on current competition standards and on current hardware. The hardware might improve and the algorithms could be parallelized. To foresee the behavior of the algorithms with these possible evolutions, we performed a second set of experiments. It consists in observing the evolution of the results of the algorithms with thinking time. The basic algorithms were run a hundred times for 12.8 seconds on each problem. The best score was recorded after each power of two multiplied by 0.1 second. We give figures that plot the average best score over the hundred runs for each recorded time. The MAX2 algorithm behaves as the best algorithm for each game.

We have identified four different patterns that occur in multiple games. For each pattern we only show its behavior on one or two games using only one or two figures, however the behavior is similar for multiple games.

In multiple games UCT+T, UCT-T and NMC behave very similarly. In other games such as as tpeg, queens, snake_2008, duplicatestatelarge, pancakes6, asteroidsserial, knightmove, peg_bugfixed and max_knight, NMC has better results than UCT+T and UCT-T and dominates for all thinking times except very short ones: at the beginning the score only moderately increases since the playouts of level 0 rapidly ends with low scores. When the playout of level 1 advances, the score increases since it avoids losing moves and makes the game last and score higher. The evolution of the average best score for tpeg and max_knights is given in Figures 9 and 5. The figures for the other games are similar.

Another class of games are the games where NMC starts with lower best scores than UCT but gets better than UCT. This is the case for example in statespacemedium as shown in Figure 6. The games where it happens are statespace and duplicatestate (without considering the results of UCT+T). In these games the rewards have a non standard distribution: in a subtree, a leaf with a 100 reward is alone among 0 reward leaves. UCT is not likely to explore this subtree while NMC has better chances to explore it due to its restart strategy.

On the contrary, at knightstour, NMC starts better than UCT but it is only at the end of the considered thinking time that both UCT+T and UCT-T get better (Figure 8). UCT performs well in this type of puzzles where it is possible to obtain a better solution by improving a good one.

There are games where using the transposition tables definitely improves UCT and enables it to become better than NMC. This is the case for hanoi (Figure 7) and duplicatestatemedium (Figure 4).

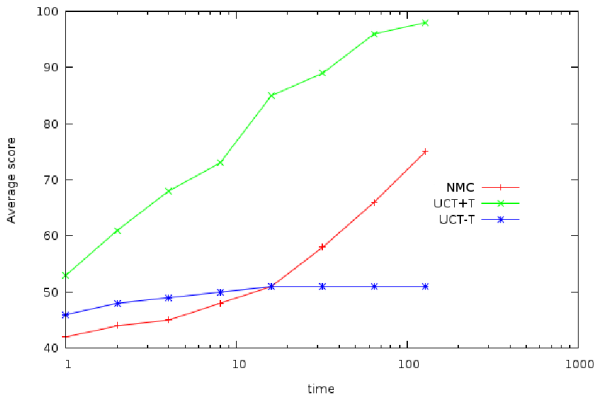

Fig. 4. Average score with time at duplicatestatemedium.

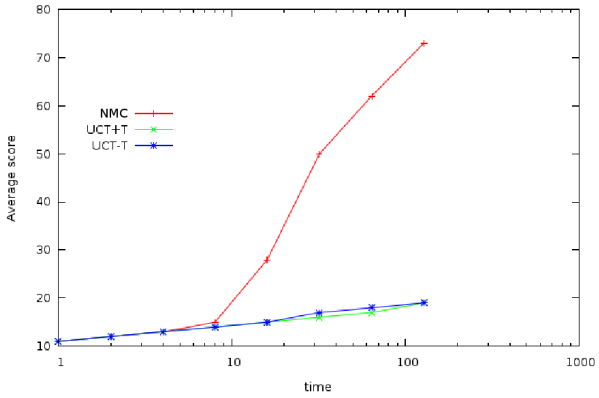

Fig. 5. Average score with time at max_knight.

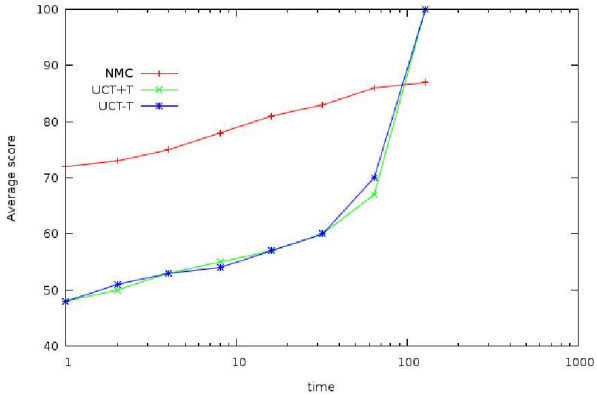

Fig. 8. Average score with time at knightstour.

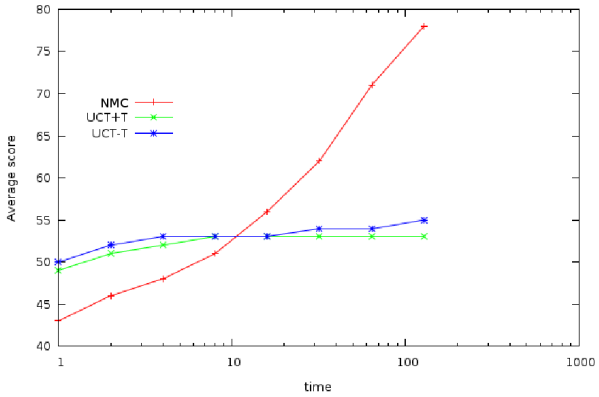

Fig. 6. Average score with time at states-pacemedium.

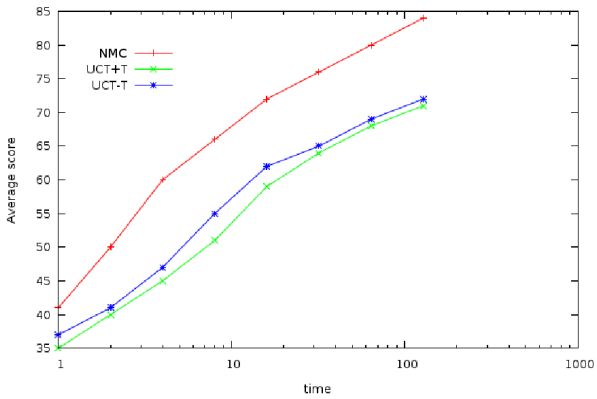

Fig. 9. Average score with time at tpeg.

7 Conclusion

We have compared four different algorithms for twenty single player games of the previous GGP competitions: UCT without transposition tables (UCT-T), UCT with transposition tables (UCT+T), Nested Monte-Carlo Search (NMC) and a simple combination of UCT+T and Nested Monte-Carlo Search (MAX).

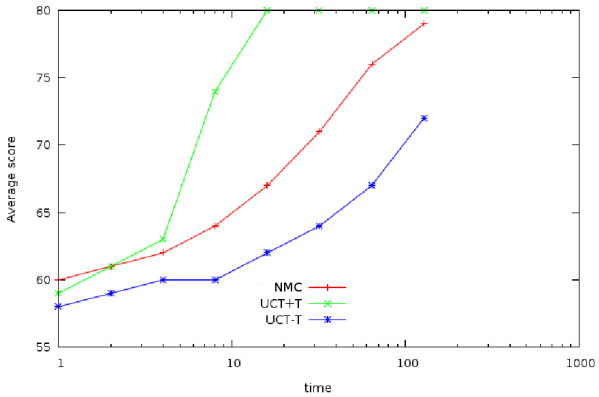

Fig. 7. Average score with time at hanoi.

On these games, UCT+T always gets better or equivalent results to UCT-T.

UCT+T works better in some games than NMC while it is the contrary in other games. On average NMC works slightly better.

The evolution of the scores of UCT+T and NMC with time is shown to be problem dependent. On some problems, one algorithm is always better than the other, while on other problems one algorithm starts worse and gets better than the other within a longer period of time.

MAX gets the performances of both algorithms by taking at each step the move with the best independent evaluation.

We combined UCT+T and NMC the most simple way, spending equal time for both algorithms. It could also be interesting to use

different ratios of the times spent for UCT+T and NMC. For example, 30% for UCT+T and 70% for NMC, or other ratios.

REFERENCES

[1] Dresden GGP server. web page, http://euklid.inf.tu-dresden.de:8180/ggpserver/, 2010.

[2] E. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways*. Academic Press, 1982.

[3] Yngvi Björnsson and Hilmar Finnsson. Cadiaplayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15, 2009.

[4] T. Cazenave. Reflexive Monte-Carlo search. In *Computer Games Workshop*, pages 165–173, Amsterdam, The Netherlands, 2007.

[5] Tristan Cazenave. Nested Monte-Carlo search. In *IJCAI*, pages 456–461, 2009.

[6] Tristan Cazenave and Abdallah Saffidine. Utilisation de la recherche arborescente Monte-Carlo au Hex. *Revue d'Intelligence Artificielle*, 23(2-3):183–202, 2009.

[7] Guillaume Chaslot, L. Chatriot, Christophe Fiter, Sylvain Gelly, Jean-Baptiste Hoock, Julien Perez, Arpad Rimmel, and Olivier Teytaud. Combiner connaissances expertes, hors-ligne, transientes et en ligne pour l'exploration Monte-Carlo. apprentissage et mc. *Revue d'Intelligence Artificielle*, 23(2-3):203–220, 2009.

[8] B. E. Childs, J H. Brodeur, and L. Kocsis. Transpositions and move groups in Monte Carlo tree search. In *CIG-08*, pages 389–395, 2008.

[9] James Clune. Heuristic evaluation functions for general game playing. In *AAAI*, pages 1134–1139, 2007.

[10] R. Coulom. Efficient selectivity and back-up operators in Monte-Carlo tree search. In *Computers and Games 2006*, Volume 4630 of LNCS, pages 72–83, Torino, Italy, 2006. Springer.

[11] Rémi Coulom. Computing "elo ratings" of move patterns in the game of go. *ICGA Journal*, 30(4):198–208, 2007.

[12] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI*, pages 259–264, 2008.

[13] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *ICML*, pages 273–280, 2007.

[14] Sylvain Gelly and David Silver. Achieving master level play in 9 x 9 computer go. In *AAAI*, pages 1537–1540, 2008.

[15] C.P. Gomes and B. Selman. Algorithm portfolio design: Theory vs. practice. In *Proceedings of UAI-97*, pages 190–197, 1997.

[16] L. Kocsis and C. Szepesvàri. Bandit based Monte-Carlo planning. In *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[17] C. S. Lee, M. H. Wang, G. Chaslot, J. B. Hoock, A. Rimmel, O. Teytaud, S. R. Tsai, S. C. Hsu, and T. P. Hong. The computational intelligence of mogo revealed in taiwan's computer go tournaments. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):73–89, 2009.

[18] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. A portfolio approach to algorithm selection. In *International Joint Conference on Artificial Intelligence*, volume 18, pages 1542–1543. Citeseer, 2003.

[19] N. Love, T. Hinrichs, and M. Genesereth. General game playing: Game description language specification. Technical report, Stanford University, 2006.

[20] Maarten P. D. Schadd, Mark H. M. Winands, H. Jaap van den Herik, Guillaume Chaslot, and Jos W. H. M. Uiterwijk. Single-player Monte-Carlo tree search. In *Computers and Games*, pages 1–12, 2008.

[21] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *AAAI*, pages 1191–1196, 2007.

[22] Michael Thielscher. Answer set programming for single-player games in general game playing. In *ICLP*, pages 327–341, 2009.

[23] L. Xu, F. Hutter, H.H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32(1):565–606, 2008.