International Journal on Artificial Intelligence Tools

Monte-Carlo Expression Discovery

TRISTAN CAZENAVE
LAMSADE, Université Paris-Dauphine, 75016, Paris, France
cazenave@lamsade.dauphine.fr

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Monte-Carlo Tree Search is a general search algorithm that gives good results in games. Genetic Programming evaluates and combines trees to discover expressions that maximize a given fitness function. In this paper Monte-Carlo Tree Search is used to generate expressions that are evaluated in the same way as in Genetic Programming. Monte-Carlo Tree Search is transformed in order to search expression trees rather than lists of moves. We compare Nested Monte-Carlo Search to UCT (Upper Confidence Bounds for Trees) for various problems. Monte-Carlo Tree Search achieves state of the art results on multiple benchmark problems. The proposed approach is simple to program, does not suffer from expression growth, has a natural restart strategy to avoid local optima and is extremely easy to parallelize.

Keywords: Monte-Carlo Tree Search ; Expression Discovery ; Nested Monte-Carlo Search ; UCT.

1. Introduction

Recently Monte-Carlo Tree Search (MCTS) has been very successful in games such as Go [6,9], General Game Playing [8,15,16,17], Hex [4,1], and Puzzles [5]. We propose to adapt the method to discover expressions that maximize a fitness function [3].

Expression discovery is usually addressed with Genetic Programming [12]. In Genetic Programming a set of random expressions is built, then the set of expressions is evolved for multiple generations. At each generation the expressions are evaluated and sorted according to their evaluation (i.e. their fitness or their score). The highest rated expressions (i.e. individuals) of a generation are then bred to create the next generation. Breeding two expressions consists in exchanging two subtrees of the two expressions represented as trees. The principle underlying Genetic Programming is that the subtrees of the expressions are building blocks that can be used to build new expressions. Subtrees that are useful in one expression are often useful in other expressions and these successful subtrees guide the search toward the good expressions. In contrast, Nested Monte-Carlo Search favors expressions that have the same successful atoms at the start of the stack representing the expression.

Genetic Programming can suffer from bloat (i.e. uncontrolled growth of the expressions with increasing numbers of generations) and from over-specialization which leads to a lack of diversity in the population and to suboptimal expressions. Restart strategies are usually used to overcome this undesired behavior.

When using MCTS to generate expressions, the size of the generated expressions as well as the restarts of the algorithm are naturally handled. In our experiments the expression generated with MCTS are usually more simple and provide scores at least equivalent to the scores of the expressions generated with Genetic Programming for the same problems. Moreover the algorithm is more simple and requires less tuning than Genetic Programming.

The second section explains Nested Monte-Carlo Search, the third section details its application to expression discovery, the fourth section outlines the application of the UCT algorithm to expression discovery, the fifth section deals with the application of Iterative Deepening to expression discovery, the sixth section gives experimental results for different problems.

2. Nested Monte-Carlo Search

The basic idea of Nested Monte-Carlo Search 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].

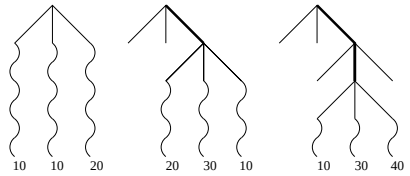

Fig. 1. At each step of the principal playout shown here with a bold line, an NMC of level n performs a NMC of level $n-1$ (shown with wavy lines) for each available move and selects the best one. At level 0, a simple pseudo-random playout is used.

Algorithm 1 Playing a playout with random sampling

```
sample (position)
while not end of game do
  position ← play (position, random move)
end while
return score (position)
```

The base level of the search plays random games (i.e. playouts), algorithm 1 shows that random moves are played until the end of the game at this level. When the game is over the score of the position that has been reached is sent back. The base level is level zero, it is called by level one and only by level one.

The algorithm for higher levels is algorithm 2. For example, at each move of a playout of level 1 the algorithm chooses the move that gives the best score when followed by a random playout (a random playout is a search at level zero). Similarly for a playout of level n it chooses the move that gives the best score when followed by a playout of level $n-1$. A search at level n calls a search at level $n-1$ to direct its search.

Algorithm 2 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)
```

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 $O(a^n h^{n+1})$.

It is important to memorize the best playout of a given level and to play its moves if no better playout has been found at the lower level. This memorization is necessary for Nested Monte-Carlo search to work at level greater than one [5]. It has also been shown that Nested Monte-Carlo search works well when the repartition of the scores is a binomial distribution, which is often the case in single player games.

When a search at the highest level is finished and there is time left, another search is performed at the highest level, and so on 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 [5,18] or SameGame [5]. It provides a good balance between exploration and exploitation and it automatically adapts its search behavior to the problem at hand without parameters tuning.

3. Nested Monte-Carlo Expression Discovery

Expressions are generated with sampling at level zero and with nested searches at higher levels. The first subsection explains the sampling algorithm. The second subsection explains the nested search algorithm.

3.1. *Random sampling*

Expressions can be seen as trees. An atom is a node of a tree. A terminal atom is a node that has no children, usually terminal atoms are the variables and the constants of a problem. For example +, -, *, / are non terminal atoms since they have two children, whereas 1, 2, 3,

x are terminal atoms.

In random sampling the tree is represented as a stack. Sampling consists in randomly filling the stack, stopping when the expression is complete (i.e. there are no more leaves to complete in the corresponding tree). A maximum number of nodes is used to prevent too large expressions. When the minimal number of potential nodes in the final tree is greater than this threshold, only terminal atoms are added to the tree in order to have less than the maximum number of nodes.

The algorithm used for sampling is given in algorithm 3. The $index$ variable is the index of the top of the stack, the $numberLeaves$ variable is the number of leaves of the current tree that have not yet been assigned. When calling sampling with an empty stack, the $numberLeaves$ variable is equal to one (i.e. there is one unassigned leaf). The $maximumNodes$ variable is the maximum number of atoms allowed for a generated expression. In Nested Monte-Carlo expression discovery, there is a stack of atoms for each level of the search. This is represented with an array of stacks. Since sampling is a search at level zero, this is $stack[0]$ which is filled with sampling.

Algorithm 3 Random sampling

```
sample (index, numberLeaves, maximumNodes)
while numberLeaves > 0 do
  if index + numberLeaves ≥ maximumNodes then
    randomly choose a terminal atom a
  else
    randomly choose an atom a
  end if
  stack[0][index] ← a
  index ← index + 1
  numberLeaves ← numberLeaves + nbChildren(a) - 1
end while
return score (stack[0])
```

Figure 2 gives an example of a partially assigned tree and of the corresponding stack. In this example, the $index$ variable is equal to three and the $numberLeaves$ variable is equal to two.

3.2. *Nested search*

The nested search maintains a stack for each level of search. At a given level; and at each step, all the possible atoms are tried and followed by a lower level search. The atom that results in the best expression is then chosen and the playout continues until the tree is completed (i.e. there are no more leaves to develop in the tree).

For example, if a level n search has reached the stack of figure 2, and the three possible atoms are +, - and 4. The next atom to choose at the top of the stack is the right child of -

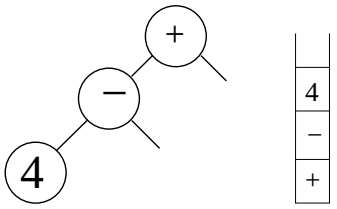

Fig. 2. A partial tree and the corresponding stack.

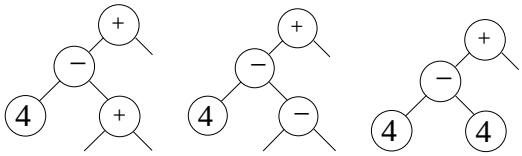

Fig. 3. The partial trees that are explored with a nested search after the tree of figure 2.

in the tree of figure 2. There are three possible atoms to put at the top of the stack, so the nested search tries them and starts a level $n-1$ search for each tree of figure 3. If one of these three searches gives a better result than the best expression found at level n, then the best expression (i.e. the stack of the best tree) is updated. After having tried the three level $n-1$ searches, the next atom of the best expression is pushed on the stack of level n and the level n search continues for the next atoms.

Intuitively, it means that if the beginning of a stack gives good results, the algorithm will guide the exploration towards expressions that have the same beginning of the stack.

The parameters of algorithm 4 are the $index$ in the stack of level $level$ ($stack[level]$), the number of unaffected leaves of the current stack ($numberLeaves$) and the maximum number of atoms of the current stack ($maximumNodes$). Initially the algorithm is called with $index$ zero, $numberLeaves$ equals to one, $maximumNodes$ depending on the problem and usually ranging from ten to one hundred, and $level$ being the level of the highest level search.

Then the best stack of the level is set to empty and the search starts and continues while there are leaves to fill (i.e. $numberLeaves > 0$).

In the loop that fills the stack, the first thing to do is to try all possible atoms on the top of the stack so as to evaluate the stacks that start with each possible atom and to choose the best one. After an atom is tried, the number of elements of the stack is updated ($index \leftarrow index + 1$) as well as the number of leaves. Then the current stack is copied to the lower level to start a search at the lower level starting at the current stack. If the level is one, the stack is completed randomly using the $sample$ function, it the level is greater than one then the stack is completed with a search at the lower level. When the search at the lower level has completed the stack, if the score of the completed stack is better than the score of the best stack of the current level, the best stack is updated. At the end of the for loop, when all possible atoms have been tried, the atom of the best expression of the current level

Algorithm 4 Nested search

```
nested (index, numberLeaves, maximumNodes, level)
best expression ← {}
while numberLeaves > 0 do
  for a in all possible atoms do
    if index + numberLeaves < maximumNodes or nbChildren (a) = 0 then
      stack[level][index] ← a
      index ← index + 1
      numberLeaves ← numberLeaves + nbChildren(a) - 1
      for i ← 0 to index do
        stack[level − 1][i] ← stack[level][i]
      end for
      if level = 1 then
        score = sample (index, numberLeaves, maximumNodes)
      else
        score = nested (index, numberLeaves, maximumNodes, level - 1)
      end if
      if score > score of best expression then
        best expression ← stack[level − 1]
      end if
      index ← index − 1
      numberLeaves ← numberLeaves - nbChildren(a) + 1
    end if
  end for
  stack[level][index] ← best expression [index]
  numberLeaves ← numberLeaves + nbChildren(stack[level][index]) - 1
  index ← index + 1
end while
return score (stack[level])
```

is pushed onto the stack. At the end of the main while loop when the stack represents a complete tree, the stack is evaluated and its score is sent back.

4. UCT Expression Discovery

UCT [10] and its enhancements [6,9] are the current best algorithms for games such as Go, Hex and General Game Playing. The principle guiding UCT is to memorize the playouts in a tree and to guide the search according to the information stored in the nodes of this tree. The information used is the mean of all the playouts that are below the node and the number of playouts. These numbers are used to descend the tree with the UCB formula. UCT addresses the exploration/exploitation tradeoff. Exploiting the available information consists in choosing the node that has the greatest mean. Exploring consists in trying nodes that have a lower mean but that can get better if they are tried. Each node is evaluated according

to the UCB formula and the policy consists in descending in the node that maximize this formula. If μ is the mean of the playouts below the node, $child$ the number of playouts of the node and $parent$ the number of playouts of the parent node, the UCB formula is: $\mu + Constant \times \sqrt{\frac{\log(parent)}{child}}$.

The constant has to be optimized for each application domain. Small constants favor exploitation while greater constants favor exploration. In Go or General Game Playing for example, for playout results being 0 or 1, a constant of 0.3 is adapted.

Algorithm 5 gives the UCT algorithm applied to expression discovery. The principle is to memorize all the playouts in a tree. Each playout corresponds to a stack of atoms representing an expression. All possible atoms are tried once at a node before growing the tree below the node. When a node has all possible children, the best child is selected using the UCB formula. A particularity of UCT applied to expression discovery is that some part of the tree may soon become completely explored. The algorithm detects the fully explored parts so as not to explore them again.

5. Depth-first Iterative Deepening Expression Discovery

Iterative Deepening is a popular method used in combination with search algorithms such as $\alpha\beta$ or A* [11]. It consists in performing a depth-first search limited by a number. For $\alpha\beta$ the limiting number is the depth of the search, for A* it is the cost of the path (the f function). When applying Iterative Deepening to Expression discovery it appears natural to limit the search with the number of atoms of the expression. The algorithm starts trying all the valid expressions containing exactly one atom, then the valid expression containing exactly two atoms, and so on.

The application of Iterative Deepening to expression discovery is detailed in algorithm 6.

6. Experimental Results

We have experimented Nested Monte-Carlo expression discovery for different problems. For each problem and each level of the nested search we have run the algorithm one hundred times. We have evaluated the results of the algorithms according to the number of expressions that have been evaluated. At each power of two we have recorded the best score. The algorithms were stopped after 2^{19} expression evaluations. The resulting figures have an x-axis with a logarithmic scale that counts the number of expression evaluations (i.e. the number of calls to the score function), ranging from 1 to 2^{19}. The y-axis gives the average best score obtained among one hundred runs.

In this section we comment the results for the different problems: Kepler's Third Law, the Santa-Fe ant trail, Prime generating polynomials, Finite algebra, the Parity problem, the N-prisoners puzzle, the MAX problem and the target number problem.

Algorithm 5 UCT search

UCT ($node$, $index$, $numberLeaves$, $maximumNodes$)
while all the possible atoms following $node$ have not been tried **do**
  if $index+numberLeaves+numberofchildrenofatom \leq maximumNodes$ **then**
    $stack[index] \leftarrow atom$
    $index \leftarrow index + 1$
    $numberLeaves \leftarrow numberLeaves$ + nbChildren($atom$) - 1
    $score \leftarrow$ sample ($index$, $numberLeaves$, $maximumNodes$)
    $n \leftarrow$ new node
    $n.atom \leftarrow atom$
    $n.sumScores \leftarrow score$
    $n.nbVisits \leftarrow 1$
    if $numberLeaves = 0$ **then**
      $n.explored \leftarrow$ true
    end if
    add n to the children of $node$
    $node.atom \leftarrow nextAtom(node.atom)$
    $node.sumScores \leftarrow node.sumScores + score$
    $node.nbVisits \leftarrow node.nbVisits + 1$
    if all children are explored and all the possible atoms have been tried **then**
      $node.explored \leftarrow$ true
    end if
    return $score$
  end if
  $node.atom \leftarrow nextAtom(node.atom)$
end while
$bestScore \leftarrow -\infty$
for all children of $node$ **do**
  $UCB \leftarrow child.mean + Constant \times \sqrt{\frac{\log(node.nbVisits)}{child.nbVisits}}$
  if not $child.explored$ **then**
    if $UCB > bestScore$ **then**
      $bestScore \leftarrow UCB$
      $bestChild \leftarrow child$
    end if
  end if
end for
$stack[index] \leftarrow bestChild.atom$
$index \leftarrow index + 1$
$numberLeaves \leftarrow numberLeaves$ + nbChildren($bestChild.atom$) - 1
$score \leftarrow UCT(bestChild, index, numberLeaves, maximumNodes)$
$node.sumScores \leftarrow node.sumScores + score$
$node.nbVisits \leftarrow node.nbVisits + 1$
if all children are explored **then**
  $node.explored \leftarrow$ true
end if
return $score$

Algorithm 6 Depth-first search

```
dfs (maximumNodes, nbNodes, numberLeaves)
if index + numberLeaves > maximumNodes then
  return
end if
if numberLeaves = 0 then
  if nbNodes = maximumNodes then
    evaluate the expression in the stack
    return
  end if
end if
for a in all possible atoms do
  stack[nbNodes] ← a
  nbNodes ← nbNodes + 1
  numberLeaves ← numberLeaves + nbChildren(a) - 1
  dfs (maximumNodes, nbNodes, numberLeaves)
  numberLeaves ← numberLeaves - nbChildren(a) + 1
  nbNodes ← nbNodes − 1
end for
```

6.1. *Kepler's Third Law and the Santa-Fe ant trail*

Rediscovering Kepler's Third Law was one of the early application of Genetic Programming [12]. The score of an expression is computed from the difference between the observed data and the result of applying the expression. Using random sampling Kepler's Third Law, i.e. the expression * (d , sqrt d), is rediscovered extremely fast, a nested search is useless for this problem.

The Santa-Fe ant trail is also an early Genetic Programming problem [12,14]. It consists in finding an automata that finds all of the 89 food cells distributed non-uniformly on a map within a given number of steps. The score of an automata is the number of food cells it has visited after the fixed number of steps. The components of an automata are TurnLeft, TurnRight, Move, IfFoodAhead, Prog2 (that executes the two following commands) and Prog3 (that executes the three following commands).

On this problem random sampling finds an optimal solution in less than 20,000 samples on average. A level one nested search improves a little on random sampling since the average score is higher than a level zero search (see figure 4), however, on average, it finds the optimal solution in the same number of evaluations. So a nested search is of limited usefulness for this problem since a random search rapidly solves the problem.

6.2. *Prime generating polynomials*

Cartesian Genetic Programming has been applied to the problem of finding polynomials that generate prime numbers [20]. The goal of the problem is to find a polynomial that gen-

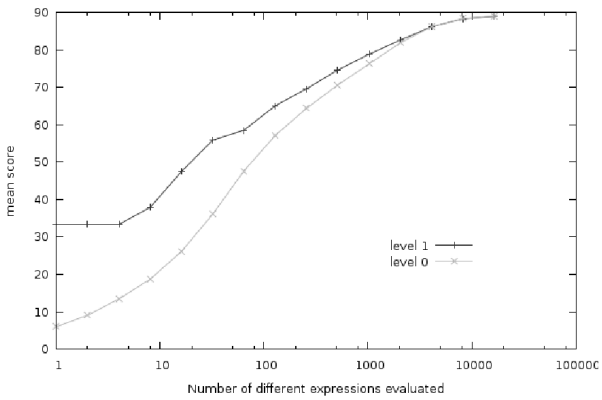

Fig. 4. Average best score with time for Nested Monte-Carlo Search applied to the Sante-Fe ant trail problem.

erates as many different primes in a row as possible.

In our experiments all the prime numbers lower than 100 were given as constants as well as the numbers between 1 and 10, the four basic operations: +, -, *, / were also used as well as the variable x. The score of an expression was the number of different primes it generates in a row for integer values of x starting at zero and increasing by one at each step.

A level two nested search found + (* (- (+ (4 , 13) , * (1 , x)) , - (4 , x)) , 83) which generates 51 primes in a row and 40 different primes. It is better than the polynomial found in [20] ($x^2 - 3x + 43$) that generates 43 primes in a row and 40 different primes. It is also better than the well known Euler polynomial ($x^2 - x + 41$) that generates 40 primes in a row, all different. Note that the Euler polynomial as well as other polynomials that give forty different primes were also found by level two nested searches.

The best score averaged over one hundred runs is given in figure 5 for levels zero to three of Nested Monte-Carlo Search. The y-axis is the average score obtained and the x-axis is the number of call to the score function. We can observe level one slightly improves on level zero, level two greatly improves on levels zero and one, and level three is slightly worse than level two.

Figure 6 gives the average score for UCT and figure 7 gives it for Iterative Deepening. We can see that UCT peaks at 15 and that Iterative Deepening peaks at 5 when Nested Monte-Carlo search peaks at 35. We can conclude Nested Monte-Carlo Search is better for this problem.

6.3. *Finite algebra*

The Finite Algebra problems consist in finding terms that satisfy some equalities [19]. We did some tests on the A_2 primal algebra from [19]. The A_2 algebra has a single operator and three elements, the operator table is given in table 1.

A discriminator term for an algebra is a term t such that $t(x, y, z) = x$ if $x \neq y$, and $t(x, x, z) = z$. It is known that if an algebra has a discriminator term then the variety

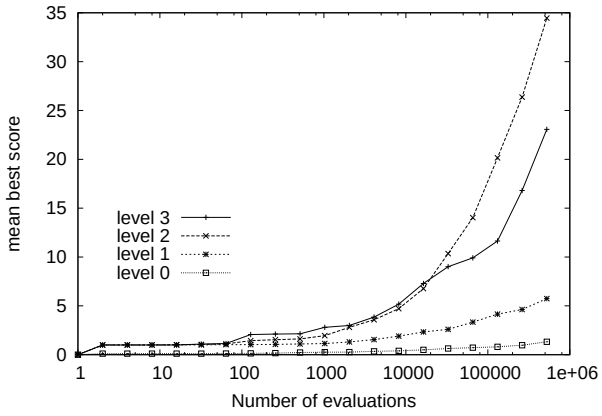

Fig. 5. Average best score for Nested Monte-Carlo search applied to the prime generation problem.

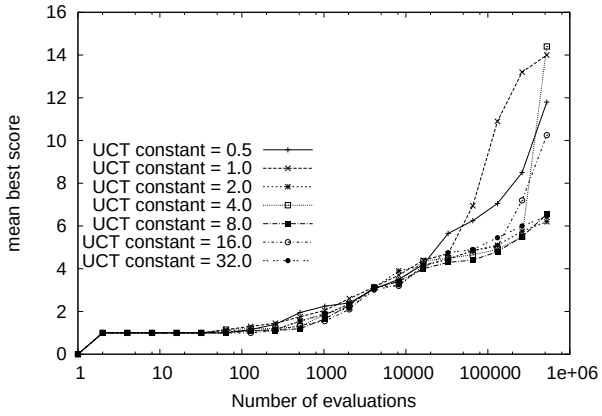

Fig. 6. Average best score with time for UCT applied to the prime generation problem.

*	0	1	2
0	2	0	2
1	1	0	2
2	1	2	1

generated by the algebra has a decidable first order theory [21].

Recently, a recursive method was provided to construct discriminator terms for a primal algebra [19], however the resulting terms are usually very long (i.e. contains millions of operations). We are interested in generating shorter terms.

An expression is only composed of the binary operator * and of the three variables x, y

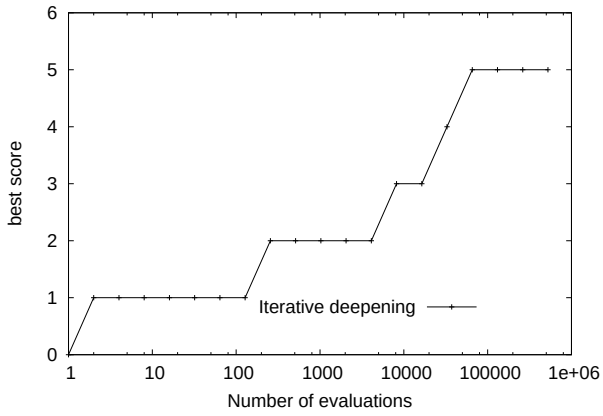

Fig. 7. Average best score with time for Iterative Deepening applied to the prime generation problem.

and z.

For each possible combination of x, y and z, the expression is evaluated and the result is compared to the desired result. There are twenty seven possible combinations (each variable can take three values), so the score of a expression is between zero and twenty seven. A score of twenty seven corresponds to a discriminator term.

For algebra A_2, a nested level three search quickly found (after 8,704,083 evaluations) a discriminator term containing 31 operations (in [19] an optimized GP found a 51 operations term after 238,000,000 evaluations). The 31 operations discriminator term is:

* (* (* (x , x) , * (x , * (* (* (* (* (* (y , * (x , x)) , x) , x) , z) , * (* (z , x) , x)) , y))) , * (* (* (z , * (* (z , * (y , * (* (* (* (z , y) , x) , z) , x))) , z)) , * (x , * (z , * (y , x)))) , * (* (x , y) , * (* (* (z , y) , z) , x))))

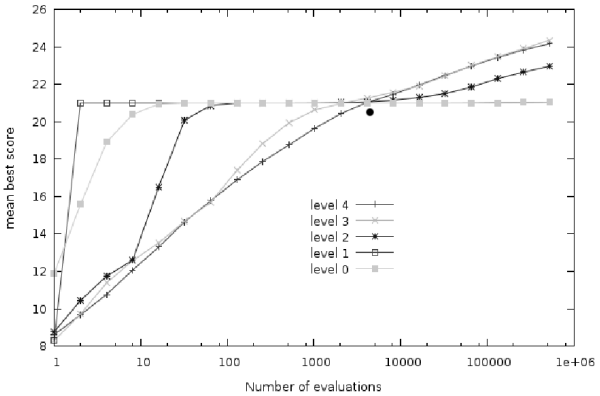

Fig. 8. Average best score with time for Nested Monte-Carlo search applied to the algebra discriminator problem.

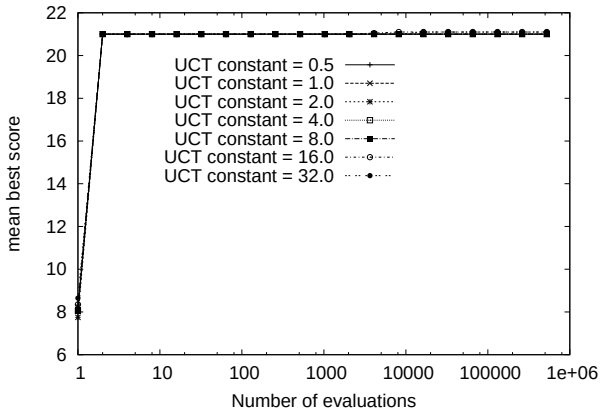

Fig. 9. Average best score with time for UCT applied to the algebra discriminator problem.

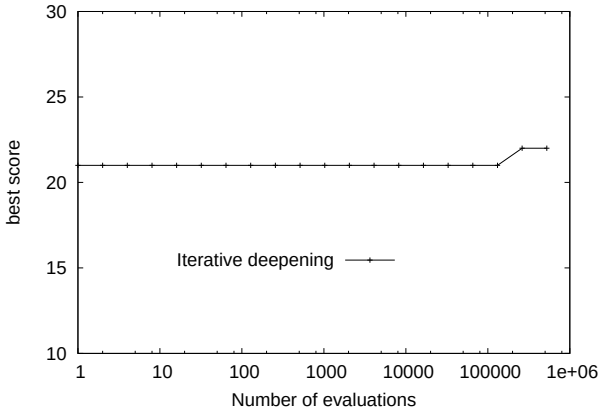

Fig. 10. Average best score with time for Iterative Deepening applied to the algebra discriminator problem.

The average best score for different levels of nested search are given in figure 8. For this problem the behavior of nested search is unusual. A level one search does not improve on random sampling, a level two search starts worse than random sampling but gets better after ten thousand evaluations. A level three search starts even worse but eventually gets better than the previous levels. A level four search behaves similarly to a level three search.

Figure 9 gives the average score for UCT with different constants. Figure 10 gives it for Iterative Deepening. We can observe that these two algorithms have difficulties getting better than a score of 21 which is the score of a very simple expression. For this problem too Nested Monte-Carlo search gets better results.

6.4. *The Parity problem*

The parity problem [12] takes as input a specified number of bits. It consists in finding an expression that returns true if the number of bits that equals to one is even. This problem is easy for Genetic Programming if the xor function can be used, it is difficult otherwise.

The score of an expression is computed after all the possible combinations of the inputs. When there are six bits, there are $2^6 = 64$ different possible inputs. The parity of each input is matched to the result of the expression for this input. The score of an expression is the number of times it gives the correct answer for the parity.

Given the functions and, or, nor, nand, xor and the inputs b1 to b6, a nested search of level one finds a solution to the parity of six booleans in 155,158 evaluations: xor (not b5 , xor (xor (b1 , xor (b4 , xor (b3 , b2))) , b6))

When only the functions and, or, nor, nand are used, the problem is much more difficult to solve. Figure 11 gives the average best scores found by nested searches of different levels. We can observe that a level one search improves much on random sampling, that a level two search improves slightly on a level one search, but that a level three search gives the same results as a level two search.

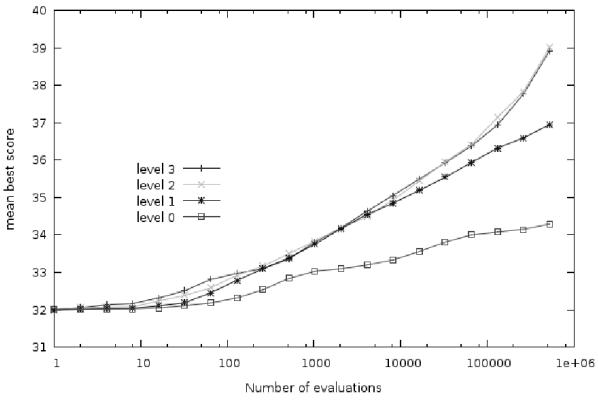

Fig. 11. Average best score with time for Nested Monte-Carlo search applied to the parity six problem.

An example of an expression that scores forty found by a level two nested search is:

and (nor (and (b2 , b3) , nor (nand (and (or (b2 , b3) , nand (b5 , b6)) , and (nand (b4 , or (and (b3 , nor (b4 , b2)) , b1)) , or (b5 , b6))) , and (nor (b4 , b5) , b4))) , or (b1 , b4))

Figure 12 gives the average score with the number of calls to the score function for the UCT algorithm. Figure 13 gives it for the Iterative Deepening algorithm. UCT is here almost equivalent to random sampling and Iterative Deepening is worse and does not have time to improve on a trivial expression.

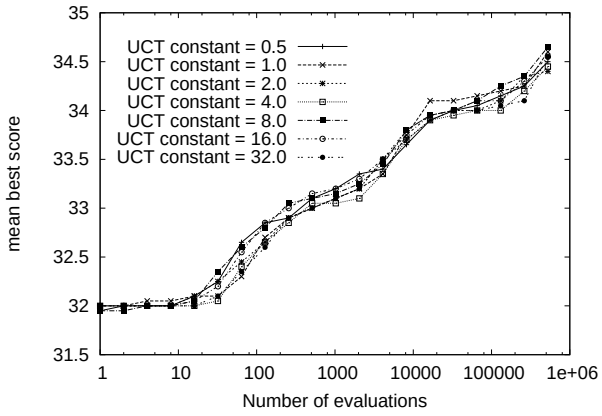

Fig. 12. Average best score with time for UCT applied to the parity six problem.

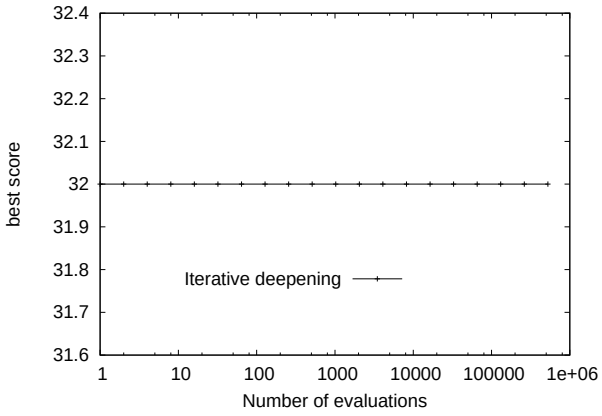

Fig. 13. Average best score with time for Iterative Deepening applied to the parity six problem.

6.5. *The N-prisoners puzzle*

The N-prisoners puzzle was made popular by Ebert in his PhD thesis [7] and is attributed to Walter Wesley Winters (1905-1973). In this problem N prisoners are assigned with either a 0 or a 1. A prisoner can see the number assigned to the other prisoners but cannot see his own number. Each prisoner is asked independently to guess if he is 0 or 1 or to pass. The prisoners can formulate a strategy before beginning the game. All the prisoners are free if at least one guesses correctly and none guess incorrectly. A possible strategy is for example that one of the prisoners says 1 and the others pass, this strategy has fifty percent chances of winning.

Genetic Programming has been applied to the N-prisoners puzzle [2]. If there are $2^k - 1$ prisoners, using a Hamming code enables to win the game with probability $\frac{2^k-1}{2^k}$ [2]. We

evolved expressions for values of N different from $2^k - 1$. The static components of an expression are the integers from 0 to 10, the operations +, -, /, *, the comparators $>$, $<$, = and the functions %, sqrt, log, exp, pow, and, or, xor, not. The variables are the bits of the other players (b_1 to b_{N-1}), me which is the number of the prisoner that is to choose, $0's$ which is the number of 0 among the other prisoners and $1's$ which is the number of 1 among the other prisoners.

For the six-prisoners puzzle an example of a strategy found by Nested Monte-Carlo Search is: - (log O's , / (xor (% (b2 , me) , b1) , 2)). It saves the prisoners 45 times out of the 64 different possible configurations.

The average score of the best expressions found by Nested Monte-Carlo search is given in figure 14. A level one search improves on a level zero search but searching at higher levels does not help for the six-prisoners problem.

Figure 15 gives the average score for UCT. It is worse than Nested Monte-Carlo Search. Figure 16 gives the scores for Iterative Deepening. For this problem Iterative Deepening has results comparable to Nested Monte-Carlo Search.

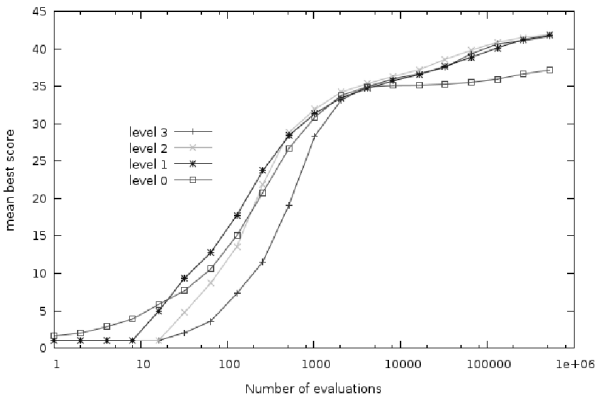

Fig. 14. Average best score with time for Nested Monte-Carlo search applied to the six-prisoners problem.

6.6. *The MAX problem*

The MAX problem [13] consists in finding an expression that results in the maximum possible number given some limit on the size of the expression. In [13] the limit was on the depth of the corresponding tree and the available atoms were +, * and 0.5. In our experiments we fixed a limit on the number of atoms of the generated expression, not on the depth of the tree.

Figure 17 gives the average best score for levels of Nested Monte-Carlo search ranging from zero to three. In these experiments the maximum number of atoms of an expression is set to forty one and the available atoms are +, * and 0.5.

We see that a level one search improves on a level zero search and that a level two

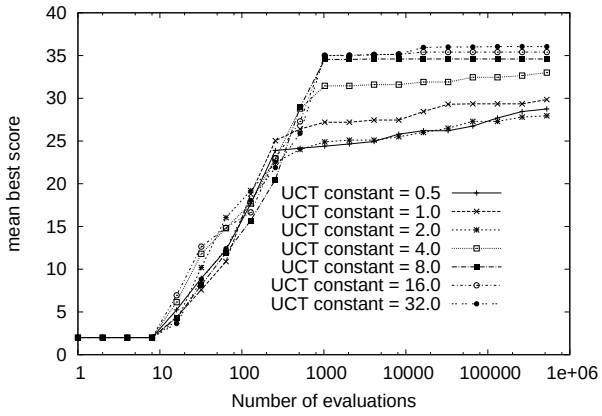

Fig. 15. Average best score with time for UCT applied to the six-prisoners problem.

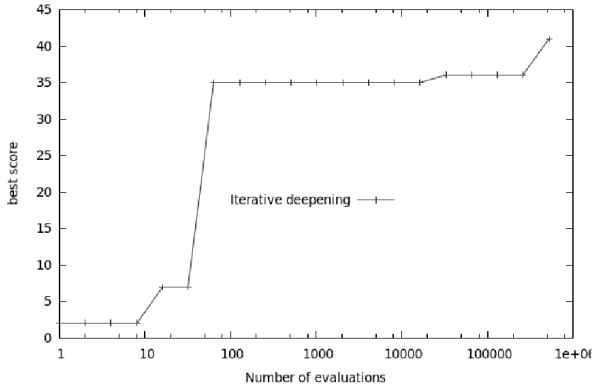

Fig. 16. Average best score with time for Iterative Deepening applied to the six-prisoners problem.

search is even better. The level three search is equivalent to the level two search.

Figure 18 and 19 give the scores for UCT and Iterative Deepening. They are worse than Nested Monte-Carlo search.

The maximum score found at level one, two and three but not at level zero is 46.875 and one of the corresponding expression is:

* (+ (+ (+ (+ (0.5 , 0.5) , 0.5) , 0.5) , 0.5) , * (* (+ (0.5 , + (+ (0.5 , + (0.5 , 0.5)) , 0.5)) , + (+ (+ (0.5 , 0.5) , + (0.5 , 0.5)) , 0.5)) , + (+ (0.5 , + (+ (0.5 , + (0.5 , 0.5)) , 0.5)) , 0.5)))

6.7. *The target number problem*

The target number problem consists in finding an expression that contains a set of predefined numbers and the atoms +, -, *, /. Each number has to be present once and only once

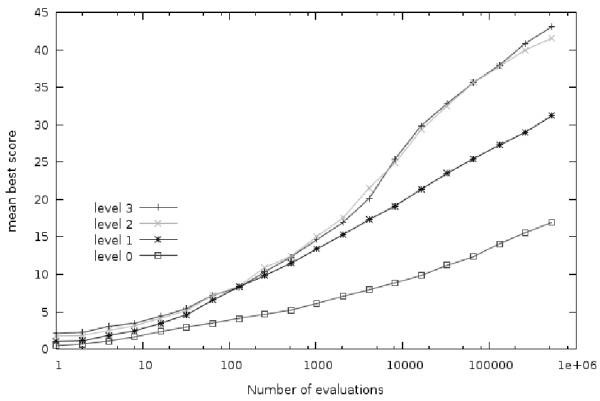

Fig. 17. Average best score with time for Nested Monte-Carlo search applied to the MAX problem.

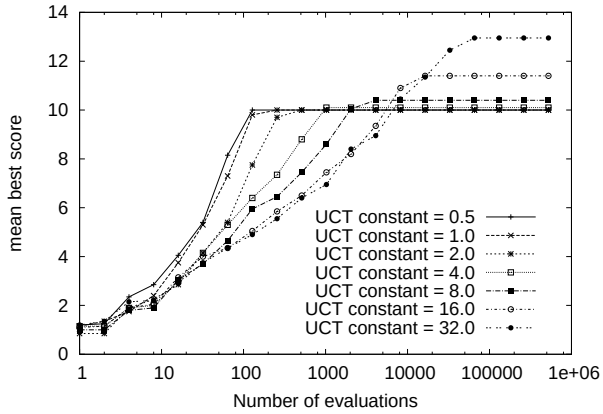

Fig. 18. Average best score with time for UCT applied to the MAX problem.

and the score of an expression is a large number (i.e. 1000) minus the absolute value of the difference to a target number. If one number is missing or present more than once the score of the expression is the number of numbers present only once.

We tested the expression problem with the set of numbers being all the numbers from 1 to 10, and the target number being 737.

The behavior of nested search is given in figure 20. A level one search improves much on a level zero search, a level two search is still a large improvement, level three and level four searches are close to level two. An expression solving the problem with a score of one thousand was found with nested search: + (+ (10 , 7) , * (* (* (1 , + (4 , 2)) , * (6 , - (8 , 3))) , - (9 , 5)))

Figure 21 gives the average score for UCT. It is equivalent to random sampling. Figure 22 gives it for Iterative Deepening. It is worse than random sampling.

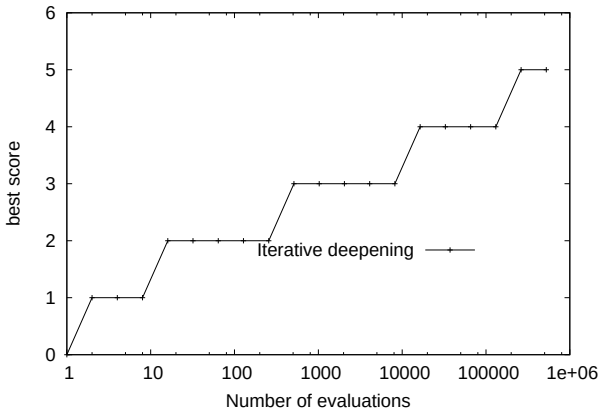

Fig. 19. Average best score with time for Iterative Deepening applied to the MAX problem.

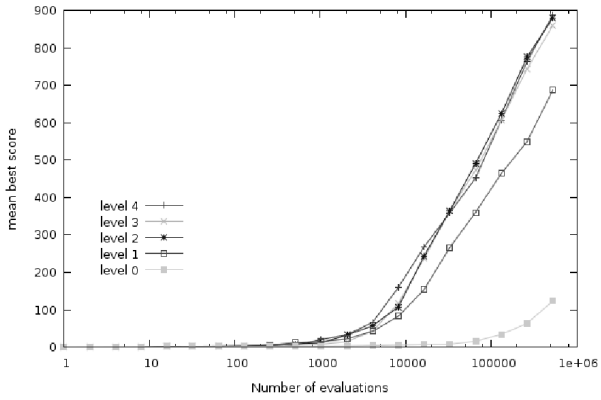

Fig. 20. Average best score with time for Nested Monte-Carlo search applied to the target number problem.

7. Conclusion

Nested Monte-Carlo search improves much on random search for difficult expression discovery problems. It is competitive with state of the art Genetic Programming since it produces shorter expressions that have equal or better scores with less evaluations for some problems such as the finite algebra problem or the prime generating polynomial problem. Moreover it is a simple and easy to program algorithm that keeps a good balance between exploration of new expressions and exploitation of already found ones. It has a natural restart strategy that ensures diversification and it avoids bloat. It is also very easy to parallelize it massively, simply running the same algorithm in parallel on many computers with a different random seed.

Nested Monte-Carlo search gives better results than UCT and that Iterative Deepening search for all the test domains examined. A reason for this behavior is that Nested Monte-

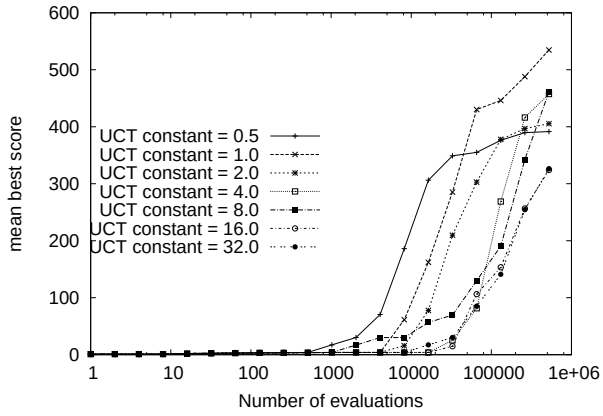

Fig. 21. Average best score with time for UCT applied to the target number problem.

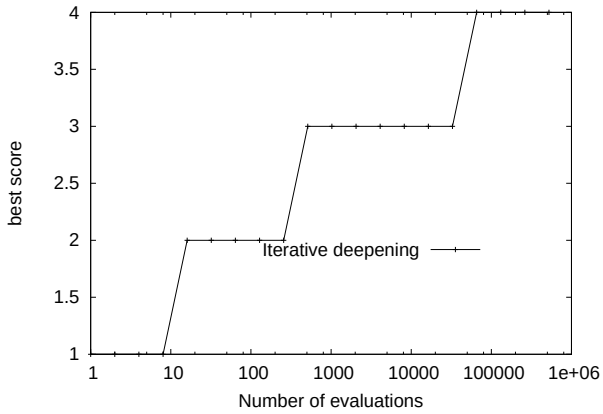

Fig. 22. Average best score with time for Iterative Deepening applied to the target number problem.

Carlo search optimizes an expression at all steps of its building while UCT and Iterative Deepening only optimize the beginning of an expression.

Concerning future works, it would certainly improve nested searches to memorize the results of previous attempts in order to direct its search. For example using a tree of previously evaluated expressions would at least make the search faster because it would not evaluate again a previously encountered expression, moreover knowing the results of previous attempts contained in the tree could help the algorithm direct its search. It would also be interesting to understand the properties of a problem that makes nested Monte-Carlo Search relevant for it.

References

1. Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Monte carlo tree search in hex. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):251–258, 2010.
2. Edmund Burke, Steven Gustafson, and Graham Kendall. A puzzle to challenge genetic programming. In *Genetic Programming*, Volume 2278 of LNCS, pages 136–147. Springer, 2002.
3. T. Cazenave. Nested Monte-Carlo expression discovery. In *ECAI*, pages 1057–1058, Lisbon, 2010.
4. T. Cazenave and Abdallah Saffidine. Utilisation de la recherche arborescente Monte-Carlo au Hex. *Revue d'Intelligence Artificielle*, 23(2-3):183–202, 2009.
5. Tristan Cazenave. Nested Monte-Carlo search. In *IJCAI*, pages 456–461, 2009.
6. 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.
7. T. Ebert. Applications of recursive operators to randomness and complexity. Phd thesis, University of California at Santa Barbara, 1998.
8. Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI*, pages 259–264, 2008.
9. Sylvain Gelly and David Silver. Achieving master level play in 9 x 9 computer go. In *AAAI*, pages 1537–1540, 2008.
10. 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.
11. R. E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
12. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, 1992.
13. W. B. Langdon and R. Poli. An analysis of the max problem in genetic programming. In J. R. Koza et al., editor, *Genetic Programming*, pages 222–230. Morgan Kaufmann, 1997.
14. W. B. Langdon and R. Poli. Why ants are hard. In J. R. Koza et al., editor, *Genetic Programming*, pages 193–201. Morgan Kaufmann, 1998.
15. J. Méhat and T. Cazenave. Ary, a general game playing program. In *Board Games Studies Colloquium*, Paris, 2010.
16. J. Méhat and T. Cazenave. Combining UCT and nested monte-carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):271–277, 2010.
17. J. Méhat and T. Cazenave. A parallel general game player. *KI journal*, 25(1):43–47, 2011.
18. Christopher D. Rosin. Nested rollout policy adaptation for monte carlo tree search. In *IJCAI*, pages 649–654, 2011.
19. Lee Spector, David M. Clark, Ian Lindsay, Bradford Barr, and Jon Klein. Genetic programming for finite algebras. In *Genetic And Evolutionary Computation Conference*, pages 1291–1298. ACM New York, NY, USA, 2008.
20. James Alfred Walker and Julian Francis Miller. Predicting prime numbers using cartesian genetic programming. In *Genetic Programming*, Volume 4445 of LNCS, pages 205–216. Springer, 2007.
21. H. Werner. *Discriminator-Algebras: Algebraic Representation and Model Theoretic Properties*. Akademie Verlag, 1978.