Nested Monte-Carlo Search

Tristan Cazenave
LAMSADE
Université Paris-Dauphine
Paris, France
cazenave@lamsade.dauphine.fr

Abstract

Many problems have a huge state space and no good heuristic to order moves so as to guide the search toward the best positions. Random games can be used to score positions and evaluate their interest. Random games can also be improved using random games to choose a move to try at each step of a game. Nested Monte-Carlo Search addresses the problem of guiding the search toward better states when there is no available heuristic. It uses nested levels of random games in order to guide the search. The algorithm is studied theoretically on simple abstract problems and applied successfully to three different games: Morpion Solitaire, SameGame and 16x16 Sudoku.

1 Introduction

When there is no available heuristic, it can be useful to perform random playouts in order to evaluate the interest of developing a position. Moreover, it is important to optimize moves at all stages of a game and not only near the root. Nested Monte-Carlo Search uses random playouts at its base level. A search at a given level uses searches at the lower level to decide which move to play in its game. Since a complete game is performed at each level the moves are optimized at all stages.

The point of this paper is to show that random moves can be successfully used at the base level of a nested search algorithm, and that memorizing the best sequence is very useful in that case. A theoretical analysis of the algorithm as well as its successful application to three different games with huge state spaces are presented.

The outline of this paper is as follows: the next section presents related work, section 3 presents the Nested Monte-Carlo Search algorithm, section 4 analyzes the algorithm on two simple abstract problems, section 5 gives experimental results for three different games.

2 Related work

The simplest Monte-Carlo search algorithm is Iterative Sampling, it consists in playing random games until a solution is found or the search time is elapsed.

Rollouts were successfully used by Tesauro and Galperin to improve their Backgammon program [Tesauro and Galperin, 1996].

Nested rollouts combined with an heuristic to choose the next move at the base level were used by Yan et al. to improve their Klondike solitaire program [Yan *et al.*, 2005]. Nested rollouts have been used with heuristics that change with the stage of the game of Thoughtful Solitaire, a version of Klondike Solitaire in which the locations of all cards is known [Bjarnason *et al.*, 2007]. These algorithms use a base heuristic which is improved with nested rollouts, whereas our algorithm uses random moves at the base level.

A related algorithm is Reflexive Monte-Carlo search [Cazenave, 2007] which has been used to find long sequences at Morpion Solitaire. The idea of Reflexive Monte-Carlo search has some similarity with the nested rollouts idea, it consists in playing random playouts at the base level, and to play a few games at the lower level of a search in order to find the best move at the current level of the search. Games at the meta level give better results than games at the lower level. In Reflexive Monte-Carlo search, there is a fixed number of games played at each level of the search before deciding the move to play. Whereas in Nested Monte-Carlo Search each possible move is tried only once before each lower level search.

The use of Monte-Carlo methods in games has been recently very successful for the game of Go [Gelly and Silver, 2007].

3 The algorithm

Nested Monte-Carlo Search combines nested calls with randomness in the playouts and memorization of the best sequence of moves. In nested rollouts the rollouts are based on a heuristic. It implies that nested rollouts always improves on rollouts and on simply following the heuristic. When the base level does not use a heuristic but random moves, it is possible that a nested search gives worse results than a lower level search. It is then useful to memorize the best sequence found so far in order to follow it when the randomized searches give worse results than the best sequence.

The basic sample function just plays a random game from a given position, we use the function *play(position, move)* which plays the move in the position and returns the resulting position:

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

The Nested Monte-Carlo Search function plays a game, choosing at each step of the game the move that has the highest score of the lower level Nested Monte-Carlo Search. At each step the algorithm tries all possible moves, plays a nested search at the lower level after each move, and memorizes the move associated to the best score of the lower level searches. As the samples are randomized, it is not guaranteed that a nested search will always improve on previous searches or even lower level searches. In order not to lose the best moves of the best sequence found by a previous search, the algorithm memorizes the best sequence. If none of the moves improve on the best sequence, the move of the best sequence is played, otherwise the best sequence is updated with the newly found sequence and the best move is played:

```
int nested (position, level)
  best score = -1
  while not end of game
    if level is 1
      move = argmax_m (sample (
                  play (position, m)))
    else
      move = argmax_m (nested (
        play (position, m), level - 1))
    if score of move > best score
      best score = score of move
      best sequence = seq. after move
    bestMove = move of best sequence
    position = play (position,bestMove)
  return score
```

The algorithm can be made anytime with iterative calls:

```
int iterativeNested (position, level)
  bestScore = -1
  while time left
    score = nested (position, level)
    bestScore = max (bestScore, score)
  return bestScore
```

4 Analysis of the algorithm

In order to explain how Nested Monte-Carlo search behaves, we analyze it on two very simple abstract problems. The search tree of both problems can be represented as a binary tree. In each state there are only two possible moves: going to the left or going to the right.

4.1 The leftmost path problem

The scoring function of the first problem consists in counting the number of moves on the leftmost path of the tree. Let's call this problem the leftmost path problem. The search space of the leftmost path problem for depth 3 is depicted in figure 1. A sample search that chooses moves randomly has a probability 2^{-n} of finding the best score of a depth n problem. A depth-first search that order moves randomly has one chance out of two of choosing the wrong move at the root, so the mean complexity of finding the best score with a depth-first search is at least 2^{n-2}. A level 1 Nested Monte-Carlo Search will always find the best score and its complexity is $n \times (n-1)$. Nested Monte-Carlo Search is appropriate for the leftmost path problem because the scores at the leaves are extremely correlated with the structure of the search tree.

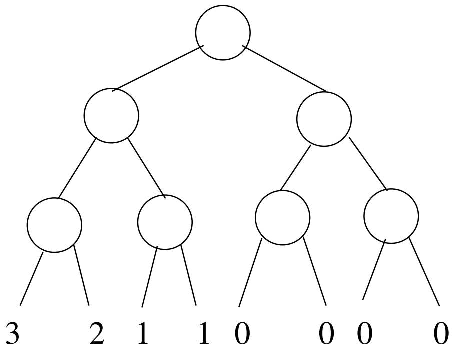

Figure 1: The score of a leaf is the number of moves on the leftmost path

4.2 The left move problem

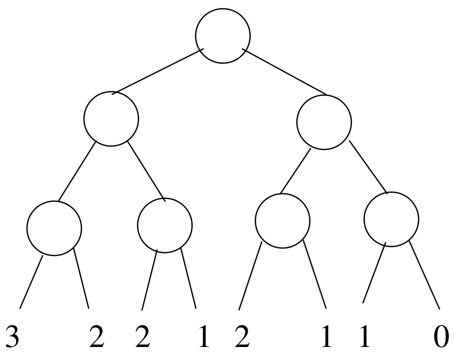

Figure 2: The score of a leaf is the number of moves to the left

However, leaf scores of a problem are not usually as correlated to the structure of the tree as in the leftmost path problem. We define the left move problem as the problem where the score of a leaf is the number of moves to the left that have been made during a game. A depth 3 left move problem search tree is depicted in figure 2. Sample search and depth-first search behave the same as in the leftmost path problem. Nested Monte-Carlo Search is less well informed in this problem. At the root of a depth d tree, the number of leaves that have a given score s is $\binom{s}{d}$, in the left branch this number is $\binom{s-1}{d-1}$, concerning the right branch this number of leaves is $\binom{s}{d-1}$. The probability that a sample search starting with a left move finds the score s is therefore $P_{leftscore}(s,d,0) = \frac{\binom{s-1}{d-1}}{2^{d-1}}$. The probability that a sample search starting with a right move finds the score s is

$P_{rightscore}(s, d, 0) = \frac{\binom{s}{d-1}}{2^{d-1}}$. We now make the assumption that when a search gives equal scores for the left and the right branch, the algorithm chooses randomly between a right move and a left move. We can now deduce the probability for a given score found at the left that it will enable the choice of a left move: $P_{leftmove}(s, d, 0) = \frac{P_{rightscore}(s,d,0)}{2} + \Sigma_{i=0}^{s-1} P_{rightscore}(i, d, 0)$ (the first term comes from the random choice in case of equality). Therefore, the probability of choosing the left move according to the score of a sample search is $P_{left}(d, 0) = \Sigma_{s=0}^{d}(P_{leftscore}(s, d, 0) \times P_{leftmove}(s, d, 0))$.

We can now remark that the distribution of the scores under any node of depth d in the tree is the same, except for the addition of a constant (which is the number of left moves played above the node). Therefore the probability of choosing a left move at a depth d node is not dependent on the location of the node in the tree, and the formula for $P_{left}(d, 0)$ is true for any depth d node in the tree.

Suppose that we already know the formulas that gives the probability $P_{leftscore}(s, d, l)$ that a level l search starting with a left move finds score s at depth d, and the probability $P_{rightscore}(s, d, l)$ that a level l search starting with a right move finds score s at depth d. We can now deduce the probability for a given score found at the left that it will enable the choice of a left move: $P_{leftmove}(s, d, l) = \frac{P_{rightscore}(s,d,l)}{2} + \Sigma_{i=0}^{s-1} P_{rightscore}(i, d, l)$, and the probability of choosing the left move at depth d according to a level l search: $P_{left}(d, l) = \Sigma_{s=0}^{d}(P_{leftscore}(s, d, l) \times P_{leftmove}(s, d, l))$.

The probability that a level l search finds score s at depth d can be written given the probability that a level $l-1$ search chooses a left move and the probabilities at depth $d-1$: $P_{score}(s, d, l) = P_{left}(d, l-1) \times P_{score}(s-1, d-1, l) + (1 - P_{left}(d, l-1)) \times P_{score}(s, d-1, l)$.

Now that we have this formula we can compute the probabilities $P_{leftscore}(s, d, l) = P_{score}(s-1, d-1, l)$ and $P_{rightscore}(s, d, l) = P_{score}(s, d-1, l)$.

We now have a recursive definition of the probability of finding a given score for a given depth and a given level.

A simple and fast recursive program with memo-functions, or a dynamic programming program, can then compute the probabilities tables for all scores, depths and levels using the formulas above. This program has computed all the probabilities for depths ranging from 1 to 100 and for level ranging from 1 to 3.

A Nested Monte-Carlo Search program for the left move problem has also been written. Statistics on finding the best score for different levels and depths were computed using 100,000 searches for each entry of the table.

The previous theoretical formula and statistics presume that the best sequence is not used. In order to evaluate the importance of memorizing the best sequence, statistics on the same levels and depths were computed using 100,000 searches with memorization of the best sequence. Memorizing the best sequence improves the results a lot. For example, a level 3 search of a depth 9 problem has a 0.41 probability of finding the best score while a similar search with memorization of the best sequence increases the probability to 0.80 without any additional cost.

Figure 3 and 4 give the distributions for a depth 60 left move problem. A Nested Monte-Carlo Search that does not memorizes the best sequence improves much less with the level than a search that does memorize it. A level 3 search with memorization has some chances of finding the best score, whereas a search without memorization has very little chances of finding it. The experimental distribution of figure 3 has been compared to the theoretical results given by the dynamic programming algorithm. The experimental results are within 1.02 % of the theoretical distribution.

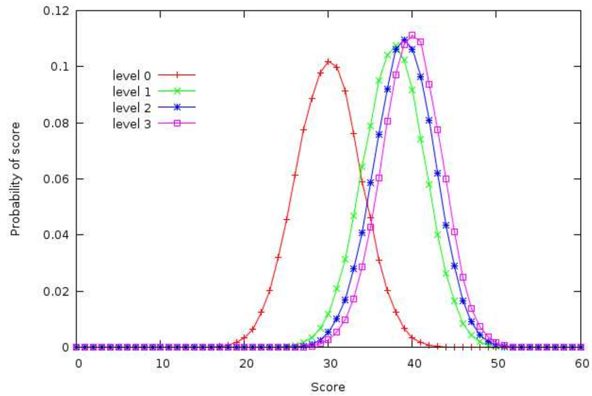

Figure 3: Distributions of the scores for a depth 60 left move problem and different search levels

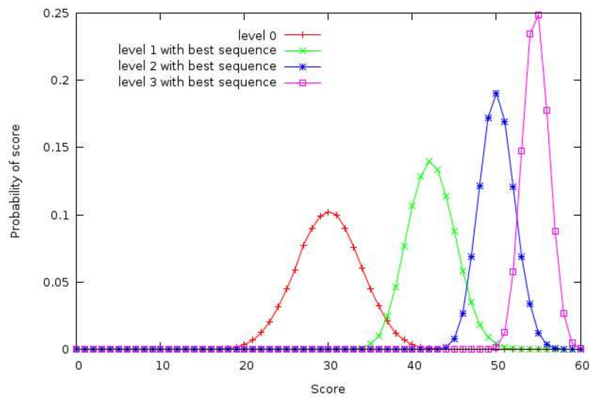

Figure 4: Distributions of the scores for a depth 60 left move problem and different search levels with memorization of the best sequence

4.3 Real-time properties

It is clear that the distribution of the scores improves with the level. However a level $n+1$ search takes more time than a level n search. In order to compare the programs according to the time they take to search, we ran 100 iterative Nested Monte-Carlo Search of 82 seconds for levels 0 to 3. For times starting at 0.01 second and doubling until 81.92 seconds we have the mean score reached with an iterated Nested

Monte-Carlo Search. Figure 5 give the mean scores for different times and levels of a search with memorization of the best sequence. The mean score of a search increases almost linearly with the logarithm of the time. A level 1 search is much better than a level 0 search (6 points), a level 2 search is 2 points better than a level 1 search and a level 3 search is one point better than a level 2 search. In a real-time setting increasing the level of the search is beneficial until level 3 for the left move problem. Figure 6 give similar results for searches without memorization of the best sequence. We see that a level 1 search is better than a level 0 search, however level 2 and 3 are worse than level 1. For the left move problem nested calls are not beneficial at level 2 and 3 if the best sequence is not memorized.

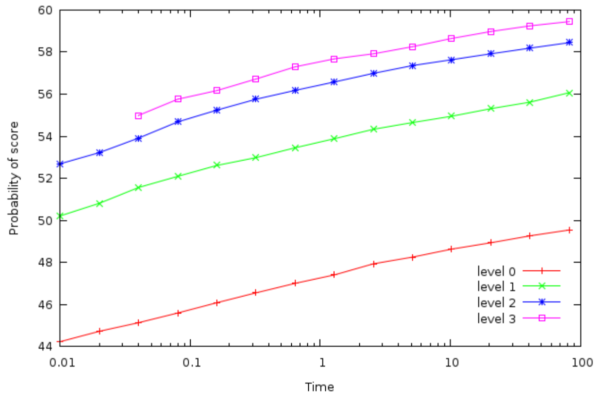

Figure 5: Mean scores of the searches with memorization of the best sequence in a real-time setting.

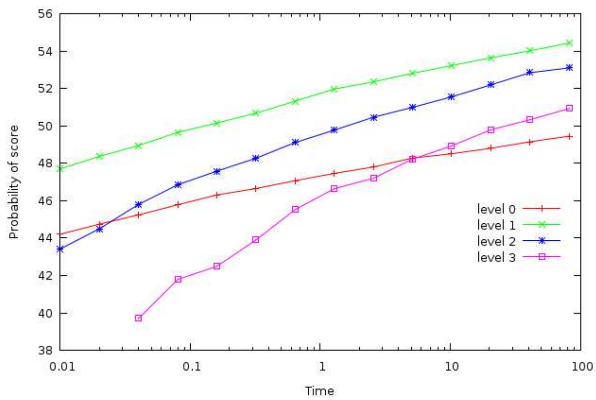

Figure 6: Mean scores of the searches in a real-time setting.

5 Experimental results

Nested Monte-Carlo Search was experimented on three quite different games: Morpion Solitaire, SameGame and 16x16 Sudoku.

5.1 Morpion Solitaire

Morpion Solitaire is an NP-hard puzzle and the high score is inapproximable within $n^{1-\epsilon}$ for any $\epsilon > 0$ unless P = NP [Demaine *et al.*, 2006]. A move consists in adding a circle such that a line containing five circles can be drawn. Lines can either be horizontal, vertical or diagonal. The starting position already contains circles disposed as in figure 7. In the disjoint version a circle cannot be a part of two lines that have the same direction. The best human score at Morpion Solitaire disjoint version is 68 moves [Demaine *et al.*, 2006]. We have tested a level 4 Nested Monte-Carlo Search on Morpion solitaire and obtained an 80 moves grid after 5 hours of computation on a cluster of 32 dual core computers. The grid is given in figure 7. On a single machine a run at level 4 takes approximately 10 days.

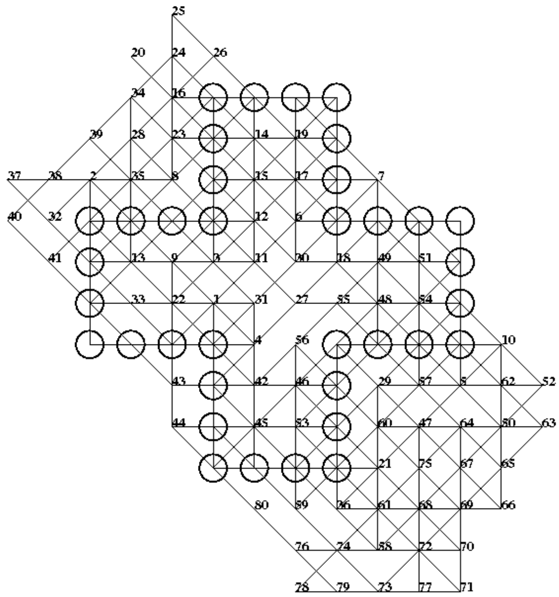

Figure 7: A world record found by Nested Monte-Carlo Search at Morpion Solitaire disjoint version

Figure 8 gives the distributions of the Morpion Solitaire scores for Nested Monte-Carlo Search without memorization of the best sequence. They were computed with 100,000 samples for level 0 and 10,000 searches for level 1 and 2. If we compare it with similar distributions of the left move problem in figure 3 we can observe that Nested Monte-Carlo Search scales better with the level for Morpion Solitaire than for the left move problem.

Figure 9 gives the distributions of the Morpion Solitaire scores for Nested Monte-Carlo Search with memorization of the best sequence. They were also computed with 100,000 samples for level 0, 10,000 searches for level 1 and 2 and 400 searches for level 3. We can see that Nested Monte-Carlo Search scales even better with the level in this case. At level 1 the peak of the distribution is at 61 when it is only at 59 without memorization. At level 2 the peak is at 66 when it is only at 62 without memorization.

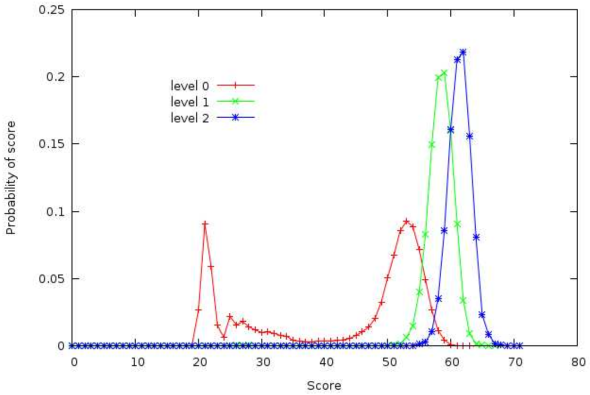

Figure 8: Distributions of the scores for Morpion Solitaire

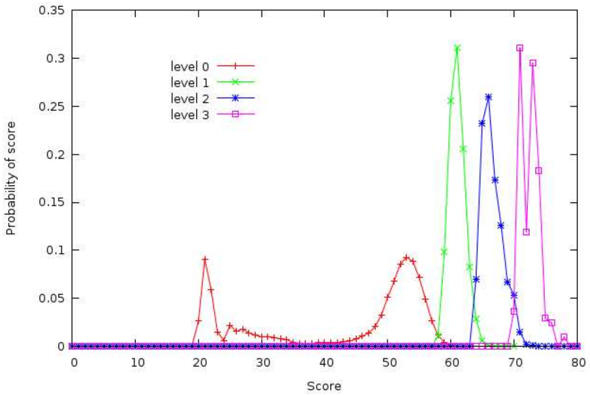

Figure 9: Distributions of the scores for Morpion Solitaire with memorization of the best sequence

In Morpion Solitaire a nested search of level l is 200 times longer than a nested search of level $l-1$. We can guess from the figure 9 that playing 200 games at level $l-1$ is likely to give a worse score that playing one game at level l for $l \leq 4$. In order to test this assumption, we computed the mean score of an iterated search for given times and levels. The results are depicted in figure 10. It is clear that a level 2 search is better than a level 1 search which is better than a level 0 search. Similarly to the left move problem, the increase in score is almost linear with the logarithm of the time. So given a time limit it is advisable to choose the highest level ≤ 4 that can be searched within the time limit, and to use iterative Nested Monte-Carlo Search.

5.2 SameGame

SameGame is an NP-complete puzzle [Kendall *et al.*, 2008]. It consists in a grid composed of cells of different colors. Adjacent cells of the same color can be removed together, scoring $(numberOfCellsRemoved - 2)^2$. When cells are removed, the upper cells fall down, and when a column is empty the columns to the right of the empty column are moved to the left. There is a bonus of 1,000 points for removing all the cells.

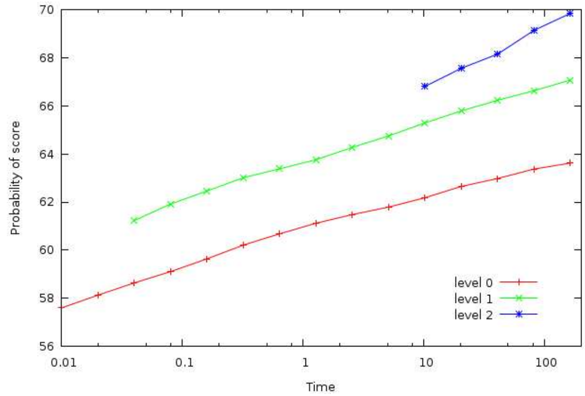

Figure 10: Mean scores of the searches with memorization of the best sequence at Morpion Solitaire in a real-time setting.

In the simulations we used the TabuColorRandom strategy [Schadd *et al.*, 2008]. It means that at the beginning of each playout the color that has the most cells is set as the tabu color. During the playouts, moves of the tabu color are played only if there are no moves of the others colors.

The previous best algorithm at SameGame used SP-MCTS based on restarts of the UCT algorithm [Schadd *et al.*, 2008], it scored 73,998 on a standard test set. With similar time settings IDA* scored a total of 22,354 and Darse Billings program scored 72,816 [Schadd *et al.*, 2008]. Nested Monte-Carlo search is more simple and gives better results at SameGame since it scores 77,934 with a level 3 search with memorization of the best sequence that corresponds roughly to the previous time settings.

Table 1 gives the scores of SP-MCTS and of a level 3 search for the 20 positions of the test set. In order to evaluate the interest of memorizing the best sequence at SameGame a level 2 search was also performed with memorization of the best sequence (level 2m) and without memorization (level 2). Memorizing the best sequence clearly improves the search since its total score is 65,937 when not memorizing only scores 44,731 at level 2.

5.3 16x16 Sudoku

Sudoku is a popular NP-complete puzzle [Kendall *et al.*, 2008] usually played on a 9x9 grid. Some cells are empty and others are filled with a number. The goal is to fill all the empty cells with numbers between 1 and 9 such that all the numbers in a row are different, all the numbers in a column are also different, and all the numbers in predefined 3x3 squares are also different.

Instead of using the usual 9x9 grid, we have used a 16x16 grid in order to have more difficult problems. The principle is the same except that numbers range from 1 to 16 and that squares have size 4x4. We have modeled Sudoku as a constraint satisfaction problem. Each cell is a variable that may contain the sixteen possible values. Each time a cell is associated to a value, all the variables in the same row, column or square are updated and the value is removed from their domain. As soon as a variable has an empty domain the search

Position	SP-MCTS	level 2	level 2m	level 3m
1	2,557	1,442	1,805	3,121
2	3,749	1,543	3,151	3,813
3	3,085	1,817	2,707	3,085
4	3,641	2,077	3,221	3,697
5	3,653	2,471	3,084	4,055
6	3,971	2,394	3,367	4,459
7	2,797	1,802	2,567	2,949
8	3,715	2,480	3,785	3,999
9	4,603	3,598	3,897	4,695
10	3,213	2,040	3,075	3,223
11	3,047	1,486	2,687	3,147
12	3,131	1,995	2,699	3,201
13	3,097	1,265	2,857	3,197
14	2,859	1,163	2,413	2,799
15	3,183	1,871	2,937	3,677
16	4,879	3,468	4,607	4,979
17	4,609	3,263	4,397	4,919
18	4,853	3,015	4,689	5,201
19	4,503	2,822	4,287	4,883
20	4,853	2,719	3,705	4,835
Total	73,998	44,731	65,937	77,934

Table 1: Results for SameGame

is stopped. During the search, the variable that has the smallest domain size is chosen to be tried first. The possible values of a variable are tried in a random order.

The difficulty of 16x16 Sudoku problems is dependent on the number of empty cells. When there are many empty cells, problems are underconstrained and easy to solve. When there are few empty cells, problems are overconstrained and also easy to solve. At the transition between overconstrained and underconstrained problems, lie the hardest problems. The most difficult problems have a percentage of empty cells close to 66%.

In order to apply nested Monte-Carlo search to 16x16 Sudoku, we have used the depth of the sample as its score (i. e. the number of variables that have been assigned before finding a variable with an empty domain). The depth is a very simple measure which biases Nested Monte-Carlo Search toward states that have a high depth, and thus are more likely to be near a solution (which is a state with the maximal depth of 256).

Table 2 gives the times used to solve 100 problems that have 66% of empty cells. The tested algorithms are Forward Checking (i.e. a depth first search), Iterative Sampling and Nested Monte-Carlo Search at level 1 and 2. Concerning Nested Monte-Carlo Search, if the first search does not find a solution, other searches of the same level are performed until a solution is found. Forward Checking (FC) is stopped when the search time for a problem exceeds 20,000 seconds, Forward Checking is unable to solve 21 problems out of 100. Iterative sampling takes much less time than Forward Checking and solves all the problems. Nested Monte-Carlo Search is clearly much better than Forward Checking, and better than Iterative Sampling. Going from a level 1 Nested Monte-Carlo Search to a level 2 search is not beneficial, maybe because

FC	Iterative Sampling	level 1	level 2
>446,771.09 s.	61.83 s.	1.34 s.	1.64 s.

Table 2: Results for 16x16 Sudoku with 66% of empty cells

the problems are already easy for a level 1 search. A level 1 search without memorization of the best sequence takes 7.00 seconds instead of 1.34 seconds. A level 2 search without memorization takes 4.87 seconds instead of 1.64 seconds.

6 Conclusion

Nested Monte-Carlo Search can be used for problems that do not have good heuristics to guide the search. The use of random games implies that nested calls are not guaranteed to improve the search. On simple abstracts problems, a theoretical analysis of nested Monte-Carlo search was presented. It was also shown that memorizing the best sequence improves a lot the mean result of the search. Experiments on three different games gave very good results, finding a new world record of 80 moves at Morpion Solitaire, improving on previous algorithms at SameGame, and being more than 200,000 times faster than depth-first search at 16x16 Sudoku modeled as a Constraint Satisfaction Problem.

References

[Bjarnason *et al.*, 2007] R. Bjarnason, P. Tadepalli, and A. Fern. Searching solitaire in real time. *ICGA Journal*, 30(3):131–142, 2007.

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

[Demaine *et al.*, 2006] E. D. Demaine, M. L. Demaine, A. Langerman, and S. Langerman. Morpion solitaire. *Theory Comput. Syst.*, 39(3):439–453, 2006.

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

[Kendall *et al.*, 2008] G. Kendall, A. Parkes, and K. Spoerer. A survey of NP-complete puzzles. *ICGA Journal*, 31(1):13–34, 2008.

[Schadd *et al.*, 2008] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, G. Chaslot, and J. W. H. M. Uiterwijk. Single-player monte-carlo tree search. In *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2008.

[Tesauro and Galperin, 1996] G. Tesauro and G. Galperin. On-line policy improvement using monte-carlo search. In *Advances in Neural Information Processing Systems 9*, pages 1068–1074, Cambridge, MA, 1996. MIT Press.

[Yan *et al.*, 2005] X. Yan, P. Diaconis, P. Rusmevichientong, and B. Van Roy. Solitaire: Man versus machine. In *Advances in Neural Information Processing Systems 17*, pages 1553–1560, Cambridge, MA, 2005. MIT Press.