Monte-Carlo Beam Search

Tristan Cazenave

Abstract—Monte-Carlo Tree Search is state of the art for multiple games and for solving puzzles such as Morpion Solitaire. Nested Monte-Carlo Search is a Monte-Carlo Tree Search algorithm that works well for solving puzzles. We propose to enhance Nested Monte-Carlo Search with Beam Search. We test the algorithm on Morpion Solitaire. Thanks to beam search, our program has been able to match the record score of 82 moves. Monte-Carlo Beam Search achieves better scores in less time than Nested Monte-Carlo Search alone.

Index Terms—Nested Monte-Carlo Search, Puzzle, Beam Search.

1 Introduction

Monte-Carlo Tree Search [16], [5] is very successful in games such as Go [11], [14], Hex [9], [1] or General Game Playing [13], [2], [18]. Building on these successes in two-player games, Monte-Carlo Tree Search algorithms were also recently used in single-player games [4], [21], [5], [19].

Nested Monte-Carlo search has been successfully applied to hard puzzles [15] such as Morpion Solitaire or SameGame [5]. Beam search is a search algorithm that selects a given number of positions among the children of the current set of positions. It has been succesfully combined with ant colony optimization to solve the travelling salesman problem with time windows for example [17]. We propose to improve Nested Monte-Carlo Search combining it with beam search.

The second section deals with Nested Monte-Carlo Search. The third section describes Monte-Carlo Beam Search. The fourth section explains Parallel Monte-Carlo Beam Search. The fifth section details experimental results.

2 Nested Monte-Carlo Search

The basic idea of Nested Monte-Carlo Search is to play a game choosing each move based on the results of a lower level Nested Monte-Carlo Search [5]. At level 1, the lower level search is simply a playout. A playout is a game where moves are played at random until the game ends.

Figure 1 illustrates a level 1 Nested Monte-Carlo search. Three selections of moves at level 1 are shown. The leftmost tree shows that at the root all possible moves are tried and that for each possible move a playout follows it. Among the three possible moves at the root, the rightmost move has the best result of 20, therefore this is the first move played at level 1. This brings us to the middle tree. After this first move, playouts are performed again for each possible move following the first move. One of the moves has result 30 which is the best playout result among its siblings. So the game continues with this move as shown in the rightmost tree.

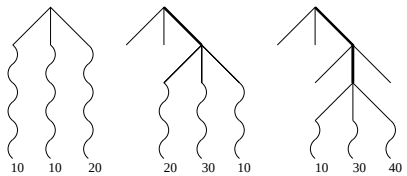


Fig. 1. This figure explains three steps of a level 1 search. At each step of the playout of level 1 (shown here with a bold line), an NMC of level 1 performs a playout (shown with wavy lines) for each available move and selects the best one.

The algorithm for higher levels is Algorithm

• T. Cazenave is at LAMSADE, Université Paris-Dauphine, 75016 Paris, France.
email: cazenave@lamsade.dauphine.fr

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)
```

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 (using the sample function that plays a completely random game). 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 $O(a^n h^{n+1})$.

In Iterative Nested Monte-Carlo Search, 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 parameter tuning. 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.

3 Monte-Carlo Beam Search

In this section we describe how we combine beam search with Nested Monte-Carlo Search. The combination consists of memorising a set of best playouts instead of only one best playout at each level. This set is called a beam and all the positions in the set are developed. The size s_{level} of a beam is fixed for each level. Only the s_{level} best playouts are kept at a given level.

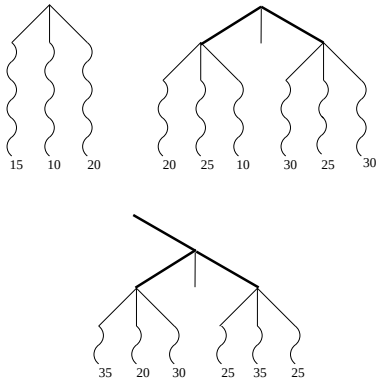


Fig. 2. Illustration of a beam search with a beam of size two. At each move, the best two positions among all children are kept.

Figure 2 intuitively explains how a beam search of level 1 and size 2 works. At the root node, there are three possible moves. For each possible move, the move is played and the resulting position is followed by a playout. The position after the move is associated to the score of the following playout. The best two positions are kept and we are now in the second tree. For each of the two positions in the beam there are three possible moves. For each position in the beam the possible moves are played and followed by a playout. Out of the resulting 6 positions, only the 2 positions that have the best playout results are kept. We are now in the third tree. There are again playouts played after each possible move in the positions of the beam, and the process continues until all positions in the beam are terminal.

Algorithm 2 details Monte-Carlo Beam Search. The variable $beam$ is the set of positions in the beam. There are at most s_{level} positions in the beam. The variable $nextBeam$ represents the best positions of the beam after one move.

Algorithm 2 Monte-Carlo Beam Search

```
beamMonteCarlo (position, level)
beam ← {position}
while true do
  nextBeam ← ∅
  for b in beam do
    p ← b
    if there is a move to play in the best playout of p then
      play (p, move of the best playout)
    end if
    add p to nextBeam
    for move in possible moves of b do
      p ← b
      play (p, move)
      if level = 1 then
        sample (p)
      else
        beamMonteCarlo (p, level − 1)
      end if
      add p to nextBeam
    end for
  end for
  if beam = nextBeam then
    break
  end if
  keep only the best s_level positions in nextBeam
  beam ← nextBeam
end while
```

The variable $nextBeam$ is set to the empty set before filling it with new positions. For all positions in the beam the following position of the best playout associated to the position is put into $nextBeam$ so as to keep the best playout if no better playout is found. Then for all possible moves in the position of the beam, the move is played and a nested search is performed on the resulting position. The best playout resulting from the nested search is associated to the position and the position is inserted in the next beam. After all positions in the beam have been developed, only the s_{level} best positions are kept. A position is better than another one if the associated playout has a better score.

Monte-Carlo Beam Search is a generalization of Nested Monte-Carlo Search. When $s_{level} = 1$ for all levels, Monte-Carlo Beam Search behaves exactly as Nested Monte-Carlo Search.

4 Parallel Monte-Carlo Beam Search

Monte-Carlo Tree Search algorithms close to UCT parallelize quite well until 16 cores [6], [7], [10], [12], while Nested Monte-Carlo Search parallelizes quite well until at least 64 cores [8].

The parallelization of Monte-Carlo Beam Search is even more simple than the parallelization of Nested Monte-Carlo Search. It consists in having a master process that performs the search at the highest level, and some remote processes that perform the search at the lower levels. The master process computes all the positions following the positions in the beam and sends them to the remote processes. The remote processes apply the lower level Monte-Carlo Beam Search to the positions they receive and send back the result to the master process. Once the master process has sent all the following positions, it receives all the searched positions and only keeps the best ones for the beam. The master and the remote processes are given in algorithms 3 and 4.

5 Experimental results

Morpion Solitaire is a single player game. The goal of the game is to play as many moves as possible. A move consists of adding a circle and in drawing a line joining five circles that have not been joined before. In the touching version two lines in the same direction can share a circle, while in the disjoint version they cannot. Before playing, a number of circles are drawn in a cross shape.

Nested Monte-Carlo Search has established world records at Morpion Solitaire [5], [20]. The current records are 82-move for the disjoint version and 178-move for the touching version. They were achieved by Chris Rosin with Nested Monte-Carlo Search and playout policy learning in August 2011.

More information on Morpion Solitaire can be found on the Morpion Solitaire web site [3]. The best human record for the touching

Algorithm 3 Master process

```
beamMonteCarlo (position, level)
beam ← {position}
while true do
  nextBeam ← ∅
  for b in beam do
    p ← b
    if there is a move to play in the best playout of p then
      play (p, move of the best playout)
    end if
    add p to nextBeam
  end for
  remote ← 1
  nbSent ← 0
  for b in beam do
    for move in possible moves of b do
      p ← b
      play (p, move)
      Send (p, remote)
      remote ← remote + 1
      if remote = # of remote CPU then
        remote ← 1
      end if
      nbSent ← nbSent + 1
    end for
  end for
  for i from 1 to nbSent do
    Receive (p)
    add p to nextBeam
  end for
  if beam = nextBeam then
    break
  end if
  keep only the best s_level positions in nextBeam
  beam ← nextBeam
end while
```

Algorithm 4 Remote process

```
remoteProcess (level)
while true do
  Receive(position)
  beamMonteCarlo (position, level − 1)
  Send(position, 0)
end while
```

version is 170 moves. This record by Charles-Henri Bruneau held for 34 years before being beaten by the combination of Nested Monte-Carlo Search and playout policy learning.

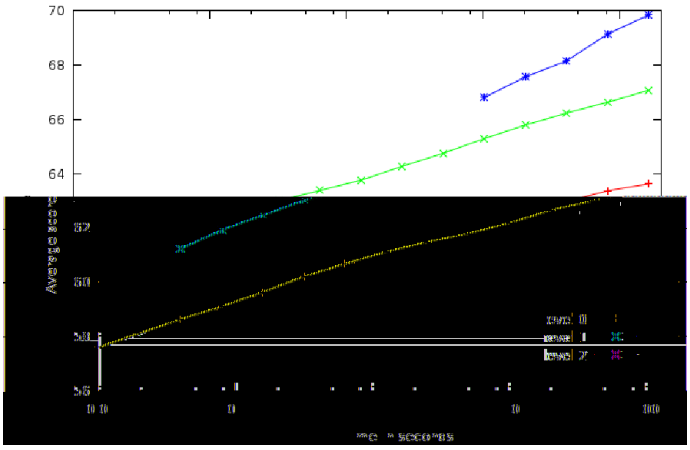


Fig. 3. Evolution of the average score of Iterative Nested Monte-Carlo Search with different levels.

Figure 3 gives the evolution of the average score of a regular Nested Monte-Carlo Search at level 0, 1 and 2. We can observe that at level 1, doubling the computation time improves the average score by half a point. At level 2, doubling the computation time improves the average score by 0.7 points. The average score is only given for time settings of which at least one iteration has been completed.

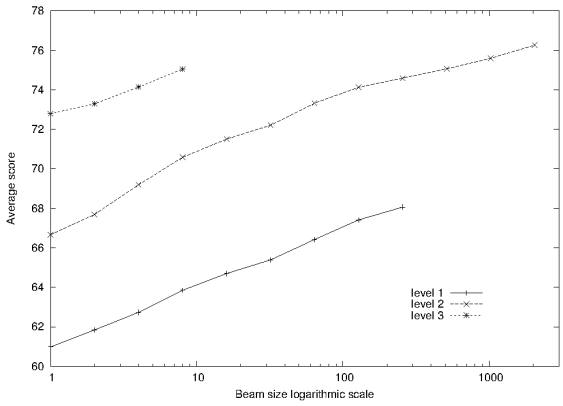


Fig. 4. Evolution of the average score of a search with different levels and beam sizes.

Figure 4 shows the evolution of the average score obtained for different beams and different levels. The sizes of the beams of the underlying

$s_1 \backslash s_2$	1	2	4	8	16	32
1	**66.66**	67.44	69.06	70.48	71.59	72.28
2	**67.84**	68.97	70.17	71.38	72.47	
4	**69.25**	**70.53**	71.47	**72.53**		
8	70.47	71.74	72.48			
16	**71.76**	72.27				
32	72.36					

TABLE 1
Average score of level 2 searches for different combinations of s_1 and s_2

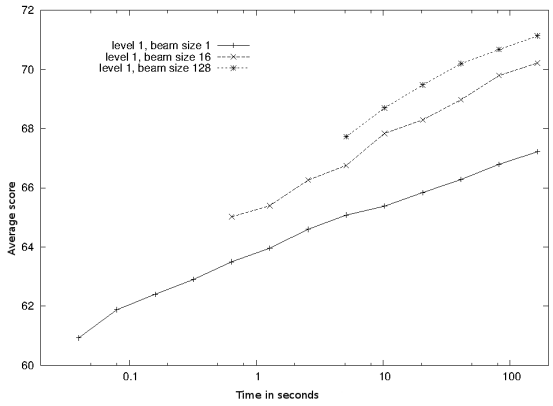
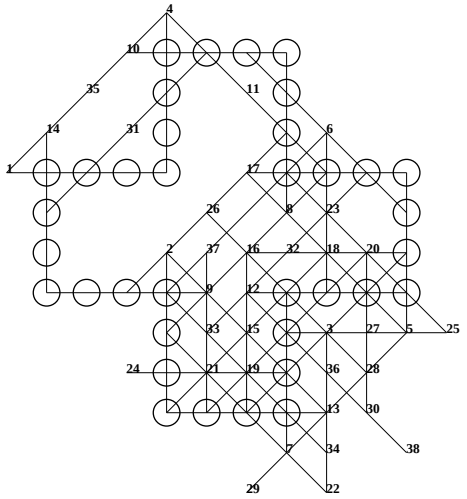


Fig. 5. Evolution of the average score of a search at level 1 for different beam sizes.

levels are set to 1 when testing a value for the beam size of a level. We can see that the behavior is slightly different for the different levels. At level two, doubling the beam size enables the algorithm to get more points than doubling the beam size at levels 1 and 3. The slope of the curve is better at level 2 until a beam of 128: a 1.1 point gain is observed for each doubling, but then it decreases to approximately half a point. Concerning level 1, approximately 0.9 points are gained for each doubling of the beam size. At level 3, approximately 0.7 points are gained for each doubling of the beam size. Doubling the beam approximately doubles the computing time for all levels.

Table 1 gives the average scores obtained for different combinations of the beam sizes at level 1 (s_1) and at level 2 (s_2). The values have been computed over 118 runs of the algorithms, and only for combinations such that $s_1 \times s_2 \leq 32$. As the running time of the algorithm is proportional to $s_1 \times s_2$, we can compare algorithms that are equivalent for the computing time, comparing the values along the antidiagonal. For each antidiagonal, the best value is put in bold. We can observe that values that have the same $s_1 \times s_2$ are quite close to each other.

Figure 5 depicts the evolution of the average score at level 1 for beams of size 1, 16 and 128. The curve for beam size 1 is the curve for the original Nested Monte-Carlo search algorithm. The two other curves are for Monte-Carlo Beam Search. We can see that using a beam much improves the nested Monte-Carlo algorithm. When running for 163 seconds, Monte-Carlo Beam Search with a beam of 16 scores three more points on average than Nested Monte-Carlo search. With a beam of size 128, it scores four more points on average.



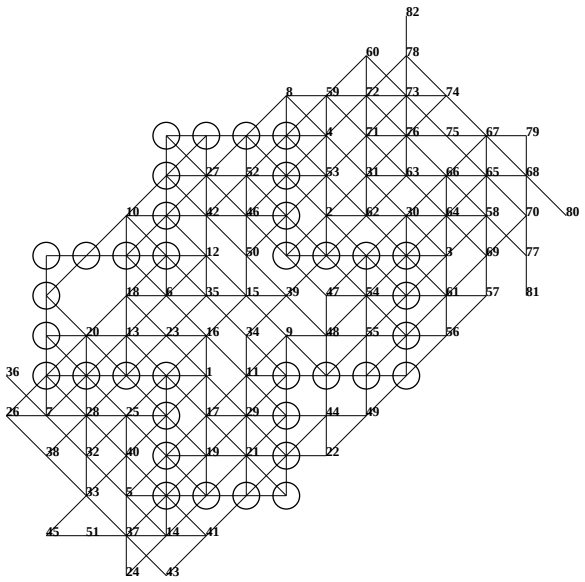


Fig. 7. Another 82-move grid.

a beam size $s_2 = 8$ (as can be observed in figure 4), other tests were done with $s_1 = 1$ and $s_2 = s_3 = 8$. These searches found multiple 80 moves grids and another 82-move record depicted in figure 7.

Previous to these experiments we ran many searches of Nested Monte-Carlo Search at level 4 using a parallel implementation. These searches achieved two 80-move grids out of many attempts. In contrast Monte-Carlo Beam Search achieved many 80-move grids and two 82-move grids with faster searches and much less time. The time to complete a search at level 3 with $s_1 = 1$ and $s_2 = s_3 = 8$ is less than the time to complete a level 4 search and it gives better scores. A Nested Monte-Carlo Search at level $n + 1$ requires 200 more computational time than a search at level n, while the Monte-Carlo Beam Search we have used only requires 64 more computational time than a Nested Monte-Carlo Search of level 3 and gives better results than a search at level 4.

6 Conclusion

We have defined a new Monte-Carlo Tree Search algorithm that combines Nested Monte-Carlo Search with Beam Search. We have shown that the use of a beam improves Nested Monte-Carlo search at Morpion Solitaire. Monte-Carlo Beam Search has matched the current world record of 82 moves at the disjoint version of Morpion Solitaire with a search faster than Nested Monte-Carlo Search at level 4. Moreover Nested Monte-Carlo Search at level 4 only achieved 80-move grids. When compared at level 1, Monte-Carlo Beam Search gets four more points on average than Nested Monte-Carlo Search. In future work, we intend to apply the algorithm to other puzzles.

Acknowledgment

I wish to thank Ed Mertensotto who told me he uses beam search in his evaluation and search based SameGame solver and Nicolas Jouandeau who helped me to run my programs on the mime cluster of University Paris 8. I also thank the anonymous reviewers for helping me improve this paper. This work has been supported by the French National Research Agency (ANR) through the COSINUS program (project EXPLO-RA ANR-08-COSI-004)

References

[1] B. Arneson, R. B. Hayward, and P. Henderson. Monte carlo tree search in hex. IEEE Trans. Comput. Intellig. and AI in Games, 2(4):251–258, 2010.

[2] Y. Björnsson and H. Finnsson. Cadiaplayer: A simulation-based general game player. IEEE Trans. Comput. Intellig. and AI in Games, 1(1):4–15, 2009.

[3] C. Boyer. Morpion solitaire. web page, http://www.morpionsolitaire.com/, 2011.

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

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

[6] T. Cazenave and N. Jouandeau. On the parallelization of UCT. In Computer Games Workshop 2007, pages 93–101, Amsterdam, The Netherlands, June 2007.

[7] T. Cazenave and N. Jouandeau. A parallel monte-carlo tree search algorithm. In Computers and Games, volume 5131 of LNCS, pages 72–80, Beijing, China, 2008. Springer.

[8] T. Cazenave and N. Jouandeau. Parallel nested monte-carlo search. In NIDISC, IPDPS, pages 1–6, 2009.

[9] T. Cazenave and Abdallah Saffidine. Utilisation de la recherche arborescente Monte-Carlo au Hex. Revue d'Intelligence Artificielle, 23(2-3):183–202, 2009.

[10] G. Chaslot, M. H. M. Winands, and H. J. van den Herik. Parallel monte-carlo tree search. In Computers and Games, volume 5131 of Lecture Notes in Computer Science, pages 60–71. Springer, 2008.

[11] 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, 2007. Springer.

[12] M. Enzenberger and M. Müller. A lock-free multithreaded monte-carlo tree search algorithm. In ACG, volume 6048 of Lecture Notes in Computer Science, pages 14–20. Springer, 2010.
[13] H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In AAAI, pages 259–264, 2008.
[14] S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In ICML, pages 273–280, 2007.
[15] G. Kendall, A. J. Parkes, and K. Spoerer. A survey of np-complete puzzles. ICGA Journal, 31(1):13–34, 2008.
[16] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In ECML, volume 4212 of Lecture Notes in Computer Science, pages 282–293. Springer, 2006.
[17] M. López-Ibáñez and C. Blum. Beam-aco for the travelling salesman problem with time windows. Computers & OR, 37(9):1570–1583, 2010.
[18] J. Méhat and T. Cazenave. Ary, a general game playing program. In Board Games Studies Colloquium, Paris, 2010.
[19] 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.
[20] C. D. Rosin. Nested rollout policy adaptation for monte carlo tree search. In IJCAI, pages 649–654, 2011.
[21] 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.