

# Monte-Carlo Tree Search Project Report

Rémi Chan-Renous, Ba Tuan Thai

April 15th, 2022

## 1 Introduction

Monte-Carlo Tree Search algorithms have successfully been applied to games [1][2]. Such algorithms consist in playing random games to choose the best move to play. Monte-Carlo methods are particularly useful when the set of possible states of the game is large and the tree of the possible states of the game cannot be fully explored.

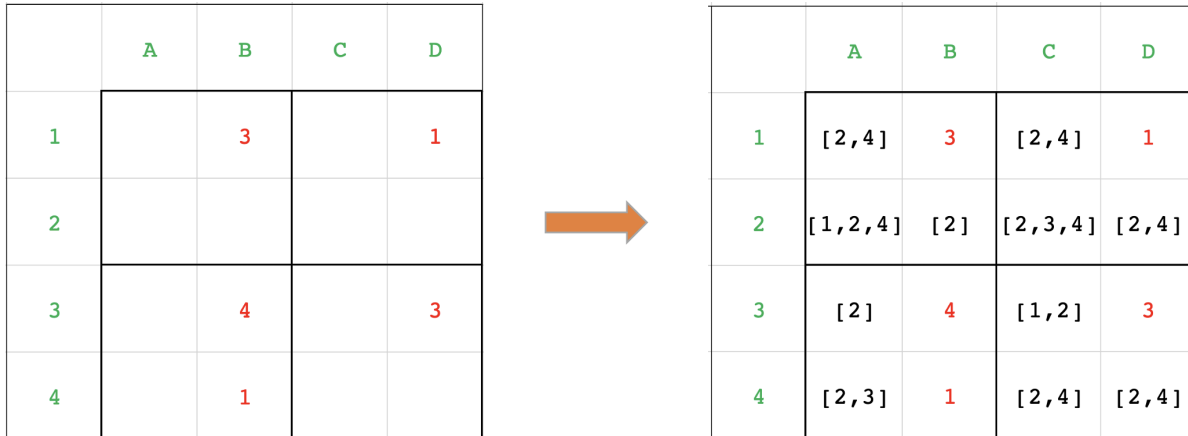
We apply in this project Monte-Carlo Tree Search algorithms to the game of Sudoku. We compare different algorithms on Sudoku 16x16 problems with 2 levels of difficulty.

## 2 Monte-Carlo algorithms for Sudoku

The game of Sudoku can be modeled as a constraint satisfaction problem and solved using Monte-Carlo algorithms [3]. In particular in the latter reference the author shows that *nested* Monte-Carlo have the best results on Sudoku 16x16 problems initialized with 66% of missing cells. They use a playing strategy that consists to play the moves with the smallest possible-values domain size.

Figure 1 shows a Sudoku 4x4 problem modeled as a constraint satisfaction problem.

We propose to compare the results of the algorithms described in 2.1. We also present different playing strategies in 2.2 and different random move selection functions in 2.3.



	A	B	C	D
1		3		1
2				
3		4		3
4		1		

	A	B	C	D
1	[2,4]	3	[2,4]	1
2	[1,2,4]	[2]	[2,3,4]	[2,4]
3	[2]	4	[1,2]	3
4	[2,3]	1	[2,4]	[2,4]

Figure 1: An example of Sudoku 4x4 problem (left figure). In the right figure you can see the domains associated to the empty cells (right figure).

## 2.1 Algorithms

We propose to compare the performances of the flat Monte-Carlo [1], UCT [4] [5] and nested Monte-Carlo [3] algorithms.

Flat Monte-Carlo is a naive algorithm which consists in picking the move that performs best when playing an equal number of random games for each available move.

UCT is an algorithm that uses a transposition table to map statistics of the outcomes of playing each available move in a given state of the game. At each state it then chooses the move with the highest *upper confidence bound* on its score, defined by

$$u := \frac{w}{n} + c\sqrt{\frac{\ln(t)}{n}}$$

where  $w$  is the sum of the (normalized) scores of the outcomes after playing that move in that state,  $n$  is the number of times the move has been played in that state,  $c$  is an exploration parameter and  $t$  is the number of games that have been played and have encountered that state so far. A bigger exploration parameter yields a more optimistic bound, implying more exploration of the set of moves. We used the Zobrist hashing function [6] to implement the transposition table.

Finally nested Monte-Carlo consists in guiding the move selection process without employing any heuristic by applying nested levels of random games.

## 2.2 Playing strategy


When playing a move we revise the domains of the cells that share a constraint with the move we just played (see figure 2 for an example of this). As a result it may occur that an empty cell has multiple, a unique or no value left in its domain. We believe that playing the unique value in the domain of the empty cells can provide additional information on the state of the game compared to a naive playing strategy that does not account for domain sizes. Indeed it may happen that a game is lost with one or multiple empty cells with unique values which could have been played and would have resulted in a different score for the game. Playing all the cells with unique values in their domain allows to unify the final result of a game.

	A	B	C	D
1	[4]	3	2	1
2	[1, 2, 4]	[2]	[3, 4]	[4]
3	[2]	4	[1]	3
4	[2, 3]	1	[4]	[2, 4]

Figure 2: Starting from the Sudoku in figure 1 and playing 2 in the cell C1 we obtain this updated version of the Sudoku where the domains of the cells have been revised.

*Maximum inference playing* refers to playing a move and all the empty cells with a unique candidate value that can be inferred from playing the initial move. Maximum inference is performed on the inferred moves to be played until there is no inferred move left to play anymore. Figure 3 shows the outcome of playing a move using the maximum inference playing strategy.

	A	B	C	D
1	[2,4]	3	[2,4]	1
2	[1,2,4]	[2]	[2,3,4]	[2,4]
3	[2]	4	[1,2]	3
4	[2,3]	1	[2,4]	[2,4]



	A	B	C	D
1	4	3	2	1
2	1	2	3	4
3	2	4	1	3
4	3	1	4	2

Figure 3: Suppose we start from the Sudoku 4x4 on the left and play the value 2 in the cell A3. The maximum inference playing results in solving the problem by playing the cells in blue in the right figure by inference from playing 2 in the cell A3.

The playing strategy employed in [3] which prioritizes the cells with the smallest domain size already uses this playing rule implicitly. The maximum inference strategy offers more liberty on the choice of the moves. This can be interpreted as a playing strategy with more exploration. Thus we propose to compare the naive playing strategy which consists in only playing a move that is randomly selected (see section 2.3 for more details on random move selection), the maximum inference strategy that we described above, and the playing strategy from [3].

## 2.3 Random move selection

We propose to experiment on the random choice of the moves in the random games.

### Randomization on the domain values

We consider each possible value in the domains of the empty cells over the cells themselves, and randomly pick one value associated to an empty cell (with uniform probability on all the domain values). As a result say there are 2 empty cells with respectively 2 and 4 candidate values. Then the latter cell has twice the probability to be played than the former cell, but each of the 6 values have the same probabilities to be played.

### Randomization on the empty cells

In this randomization technique we first start by randomly picking an empty cell then pick randomly a value from the domain. Taking the example with 2 empty cells with respectively 2 and 4 candidates values in their domain, now both cells have the same probability to be played but the values from the domain of the latter cell have twice the probability to be played of the values from the domain of the former cell.

## Randomization on the cells with the smallest domains

This randomization technique consists in randomly selecting a move from the domain of the cell with the smallest domain size. It is the one that is used in [3].

Note that given that we keep track of the empty cells and their possible values the implementation of the randomization on the empty cells is rather straightforward. Regarding the randomization method that prioritizes the moves of the cells with the smallest domain size, we keep track of the domain size of the empty cells in our implementation, facilitating the implementation of this randomization method. Nevertheless it still requires to sort the empty cells by the size of their domain which can be costly. The implementation of the randomization on the domain values requires to explicit all the possible moves to select a move. This operation seems to be costly as well.

## 3 Experiments

We compare the performance of the flat Monte-Carlo, UCT and nested Monte-Carlo algorithms on the game of Sudoku 16x16. We compare the different versions of the algorithms that we described in section 2 and different sets of parameters related to the algorithms. The algorithms are tested on Sudoku 16x16 problems generated randomly with 66% and 50% of empty cells. The tests consist in solving a certain number of problems. The algorithms are run and restarted until the current problem is solved or a time limit is exceeded. The full results and details of the experiments are available in appendix A<sup>1</sup>.

No matter the number of random games used to choose the move to play in the flat Monte-Carlo algorithm, the playing strategy that consists to play the moves from the cells with the smallest domain size leads to impressive results on the easy problems (50% of empty cells) with an average time of almost 0.2 second per problem. This playing strategy also performs the best on the harder problems (66% of empty cells) by solving 14 out of the 15 problem, the best average solving time being close to 23 seconds. The naive playing strategy fails to solve any of the problems while the maximum inference playing succeeds to solve most of the easy problems, the best average solving time being around 48 seconds. Nevertheless the maximum inference strategy barely solves any of the hard problems.

Regarding the UCT algorithm, the naive playing strategy fails to solve any problem within the 10 minutes interval. Here the playing strategy that prioritizes the moves from the cells with smallest domain size performs poorly on the easy problems (at most 7 problems out of 15 are solved using this strategy) and fails to solve any hard problem. For each set of parameters the maximum inference strategy solved at least 7 problems out of 15, with one configuration that solves all the easy problems with an average time under 6 seconds. The maximum playing strategy barely solves any hard problem in less than 10 minutes.

Finally the best results on both the easy and hard problems of the level 1 nested Monte-Carlo algorithm are comparable to those of the Flat Monte-Carlo algorithm. Level 1 nested Monte-Carlo achieves an average solving time of 0.25 seconds on the 50 easy problems using the move selection that prioritizes the cells with the smallest domain size. The level 2 nesting shows satisfying results on the easy problems with an average time per solved problem of about 23 seconds. Again the naive strategy fails to solve any problem. The maximum inference strategy solves most of the easy problems no matter the random move selection method but the total time on the problems is way higher than when using the playing strategy that prioritizes the moves from the cells

---

<sup>1</sup>the code and notebooks are available at [https://github.com/rchanrenous/sudoku\\_mcts](https://github.com/rchanrenous/sudoku_mcts)

with the smallest domain values. Regarding the harder problems the playing strategy from [3] associated to the nested Monte-Carlo algorithm solves more than 80% of the 50 problems with an average solving time of about 34 seconds for level 1 nesting and 1 min 30 seconds for level 2 nesting.

In a nutshell the best results - both on the easy and hard problems - are achieved by the level 1 nested Monte-Carlo algorithm and flat Monte-Carlo algorithm when using the playing strategy from [3]. The UCT algorithm does not seem to be a good heuristic to solve the Sudoku 16x16 problems. Overall, the maximum inference playing strategy shows better results than the naive playing strategy. Nevertheless the playing strategy that selects the moves with the smallest domain size shows better results than these two other playing strategies. These results seem to be explained by the consistency of the search that this playing strategy offers. It is more consistent than the maximum inference playing which is in turn more consistent than the naive playing strategy. Note that the randomization over the cells offers slightly better results than the randomization over the values. The latter tends to favor the selection of cells with larger domain sizes which is the opposite of the playing strategy that selects the cells with the smallest domain size first.

## 4 Conclusion

We proposed in this project to compare different Sudoku playing methodologies for the flat Monte-Carlo, UCT and nested Monte-Carlo algorithms. In particular we introduced the maximal inference playing strategy which is in between the naive playing strategy and the playing strategy from [3] in terms of consistency of the search. When used along the flat and nested Monte-Carlo algorithms, the maximum inference playing succeeds to solve easy problems but struggles to solve any hard problem. We observed empirically that applying this random strategy to the flat Monte-Carlo algorithm shows results on par with the level 1 nested Monte-Carlo algorithm with the same playing strategy, which are the best results on both the easy and hard Sudoku 16x16 problems in our experiments. The UCT algorithm proved inefficient in our experiments, both on easy and hard problems.

## References

- [1] Cameron Browne et al. “A Survey of Monte Carlo Tree Search Methods”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4 (2012).
- [2] Tristan Cazenave. *Intelligence Artificielle une Approche Ludique*. 2011, p. 256.
- [3] Tristan Cazenave. “Nested Monte-Carlo Search”. In: *IJCAI* (2009), pp. 456–461.
- [4] Levente Kocsis and Csaba Szepesvári. “Bandit based monte-carlo planning”. In: *European conference on machine learning* (2006), pp. 282–293.
- [5] Sylvain Gelly and David Silver. “Monte-Carlo tree search and rapid action value estimation in computer Go”. In: *Artificial Intelligence* 175 (2011), pp. 1856–1875.
- [6] Albert L. Zobrist. “A New Hashing Method With Application for Game Playing”. In: *CS Technical Reports* (1970).

## A Results

15 Sudoku 16x16 problems with 50% of empty cells					
Algorithm	Maximum Inference	Random move function	Number of random games	Number of problems solved	Total time (seconds)
Flat	yes	cells	5	13	1652.144
Flat	no	cells	5	0	9000
Flat	yes	values	5	14	1773.744
Flat	no	values	5	0	9000
<b>Flat</b>	<b>no</b>	<b>priority</b>	<b>5</b>	<b>15</b>	<b>3.151</b>
Flat	yes	cells	10	15	727.346
Flat	no	cells	10	0	9000
Flat	yes	values	10	15	1250.117
Flat	no	values	10	0	9000
<b>Flat</b>	<b>no</b>	<b>priority</b>	<b>10</b>	<b>15</b>	<b>3.350</b>
Flat	yes	cells	15	14	1548.630
Flat	no	cells	15	0	9000
Flat	yes	values	15	12	2464.996
Flat	no	values	15	0	9000
<b>Flat</b>	<b>no</b>	<b>priority</b>	<b>15</b>	<b>15</b>	<b>3.304</b>
15 Sudoku 16x16 problems with 66% of empty cells					
Algorithm	Maximum Inference	Random move function	Number of random games	Number of problems solved	Total time (seconds)
Flat	yes	cells	5	1	8493.078
Flat	no	cells	5	0	9000
Flat	yes	values	5	0	9000
Flat	no	values	5	0	9000
<b>Flat</b>	<b>no</b>	<b>priority</b>	<b>5</b>	<b>14</b>	<b>1849.749</b>
Flat	yes	cells	10	1	8513.095
Flat	no	cells	10	0	9000
Flat	yes	values	10	0	9000
Flat	no	values	10	0	9000
<b>Flat</b>	<b>no</b>	<b>priority</b>	<b>10</b>	<b>14</b>	<b>955.921</b>
Flat	yes	cells	15	0	9000
Flat	no	cells	15	0	9000
Flat	yes	values	15	0	9000
Flat	no	values	15	0	9000
<b>Flat</b>	<b>no</b>	<b>priority</b>	<b>15</b>	<b>14</b>	<b>1339.786</b>

Table 1: Experiments on the Flat MC algorithm to solve 15 Sudoku 16x16 problems with 50% and 66% of empty cells (with a 10 minutes limit to solve each problem). The number of random games represents the number of random games played for each move available at each each to determine the next move to play.

15 Sudoku 16x16 problems with 50% of empty cells						
Algorithm	Exploration parameter	Maximum Inference	Random move function	Number of random games	Number of problems solved	Total time (seconds)
<b>UCT</b>	<b>0.2</b>	<b>yes</b>	<b>cells</b>	<b>2000</b>	<b>13</b>	<b>2750.998</b>
UCT	0.2	no	cells	2000	0	9000
UCT	0.2	yes	values	2000	8	4774.089
UCT	0.2	no	values	2000	0	9000
UCT	0.2	no	priority	2000	6	5539.764
UCT	0.2	yes	cells	4000	9	3632.823
UCT	0.2	no	cells	4000	0	9000
<b>UCT</b>	<b>0.2</b>	<b>yes</b>	<b>values</b>	<b>4000</b>	<b>10</b>	<b>3076.683</b>
UCT	0.2	no	values	4000	0	9000
UCT	0.2	no	priority	4000	5	6032.590
<b>UCT</b>	<b>0.2</b>	<b>yes</b>	<b>cells</b>	<b>6000</b>	<b>15</b>	<b>89.171</b>
UCT	0.2	no	cells	6000	0	9000
UCT	0.2	yes	values	6000	8	4248.537
UCT	0.2	no	values	6000	0	9000
UCT	0.2	no	priority	6000	6	5481.752
UCT	0.4	yes	cells	2000	10	5466.813
UCT	0.4	no	cells	2000	0	9000
<b>UCT</b>	<b>0.4</b>	<b>yes</b>	<b>values</b>	<b>2000</b>	<b>9</b>	<b>4588.473</b>
UCT	0.4	no	values	2000	0	9000
UCT	0.4	no	priority	2000	7	5119.826
UCT	0.4	yes	cells	4000	8	4230.168
UCT	0.4	no	cells	4000	0	9000
<b>UCT</b>	<b>0.4</b>	<b>yes</b>	<b>values</b>	<b>4000</b>	<b>10</b>	<b>3049.847</b>
UCT	0.4	no	values	4000	0	9000
UCT	0.4	no	priority	4000	5	6615.857
<b>UCT</b>	<b>0.4</b>	<b>yes</b>	<b>cells</b>	<b>6000</b>	<b>10</b>	<b>3042.469</b>
UCT	0.4	no	cells	6000	0	9000
UCT	0.4	yes	values	6000	7	4838.965
UCT	0.4	no	values	6000	0	9000
UCT	0.4	no	priority	6000	5	6080.865

15 Sudoku 16x16 problems with 66% of empty cells						
Algorithm	Exploration parameter	Maximum Inference	Random move function	Number of random games	Number of problems solved	Total time (seconds)
UCT	0.2	yes	cells	2000	0	9000
UCT	0.2	no	cells	2000	0	9000
<b>UCT</b>	<b>0.2</b>	<b>yes</b>	<b>values</b>	<b>2000</b>	<b>2</b>	<b>8175.517</b>
UCT	0.2	no	values	2000	0	9000
UCT	0.2	no	priority	2000	0	9000
<b>UCT</b>	<b>0.2</b>	<b>yes</b>	<b>cells</b>	<b>4000</b>	<b>1</b>	<b>8787.417</b>
UCT	0.2	no	cells	4000	0	9000
UCT	0.2	yes	values	4000	0	9000
UCT	0.2	no	values	4000	0	9000
UCT	0.2	no	priority	4000	0	9000
<b>UCT</b>	<b>0.2</b>	<b>yes</b>	<b>cells</b>	<b>6000</b>	<b>1</b>	<b>8574.005</b>
UCT	0.2	no	cells	6000	0	9000
UCT	0.2	yes	values	6000	1	8668.906
UCT	0.2	no	values	6000	0	9000
UCT	0.2	no	priority	6000	0	9000
<b>UCT</b>	<b>0.4</b>	<b>yes</b>	<b>cells</b>	<b>2000</b>	<b>1</b>	<b>8547.167</b>
UCT	0.4	no	cells	2000	0	9000
UCT	0.4	yes	values	2000	0	9000
UCT	0.4	no	values	2000	0	9000
UCT	0.4	no	priority	2000	0	9000
UCT	0.4	yes	cells	4000	0	9000
UCT	0.4	no	cells	4000	0	9000
UCT	0.4	yes	values	4000	0	9000
UCT	0.4	no	values	4000	0	9000
UCT	0.4	no	priority	4000	0	9000
UCT	0.4	yes	cells	6000	0	9000
UCT	0.4	no	cells	6000	0	9000
UCT	0.4	yes	values	6000	0	9000
UCT	0.4	no	values	6000	0	9000
UCT	0.4	no	priority	6000	0	9000

Table 2: Experiments on the UCT algorithm to solve 15 Sudoku 16x16 problems with 50% and 66% of empty cells (with a 10 minutes limit to solve each problem). The number of random games represents the total number of random games that are played at each state to choose the next move to play.



50 Sudoku 16x16 problems with 50% of empty cells					
Algorithm	Level	Maximum Inference	Random move function	Number of problems solved	Total time (seconds)
Nested	1	yes	cells	44	1989.084
Nested	1	no	cells	0	9000
Nested	1	yes	values	42	2639.471
Nested	1	no	values	0	9000
<b>Nested</b>	<b>1</b>	<b>no</b>	<b>priority</b>	<b>50</b>	<b>12.549</b>
Nested	2	yes	cells	45	4736.538
Nested	2	no	cells	0	9000
Nested	2	yes	values	46	4418.875
Nested	2	no	values	0	9000
<b>Nested</b>	<b>2</b>	<b>no</b>	<b>priority</b>	<b>49</b>	<b>1321.077</b>
50 Sudoku 16x16 problems with 66% of empty cells					
Algorithm	Level	Maximum Inference	Random move function	Number of problems solved	Total time (seconds)
Nested	1	yes	cells	2	8940.495
Nested	1	no	cells	0	9000
Nested	1	yes	values	0	9000
Nested	1	no	values	0	9000
<b>Nested</b>	<b>1</b>	<b>no</b>	<b>priority</b>	<b>40</b>	<b>3165.262</b>
Nested	2	yes	cells	4	9000
Nested	2	no	cells	0	9000
Nested	2	yes	values	0	9000
Nested	2	no	values	0	9000
<b>Nested</b>	<b>2</b>	<b>no</b>	<b>priority</b>	<b>41</b>	<b>5255.274</b>

Table 3: Experiments on the nested Monte-Carlo algorithm to solve 50 Sudoku 16x16 problems with 50% and 66% of empty cells (with a 3 minutes limit to solve each problem).