

Monte-Carlo Tree Search for the "Mr Jack" Board Game

Ahmad Mazyad, Fabien Teytaud, Cyril Fonlupt

▶ **To cite this version:**

Ahmad Mazyad, Fabien Teytaud, Cyril Fonlupt. Monte-Carlo Tree Search for the "Mr Jack" Board Game. International Journal on Soft Computing, Artificial Intelligence and Applications (IJSCAI), 2015. hal-01406506

HAL Id: hal-01406506
https://inria.hal.science/hal-01406506

Submitted on 1 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Monte-Carlo Tree Search for the "Mr Jack" Board Game

A. Mazyad, F. Teytaud, and C. Fonlupt

LISIC, ULCO, Univ Lille–Nord de France, FRANCE

Abstract. Recently the use of the Monte-Carlo Tree Search algorithm, and in particular its most famous implementation, the Upper Confidence Tree can be seen has a key moment for artificial intelligence in games. This family of algorithms provides huge improvements in numerous games, such as Go, Havannah, Hex or Amazon. In this paper we study the use of this algorithm on the game of *Mr Jack* and in particular how to deal with a specific decision-making process. *Mr Jack* is a 2-player game, from the family of board games. We will present the difficulties of designing an artificial intelligence for this kind of games, and we show that Monte-Carlo Tree Search is robust enough to be competitive in this game with a smart approach.

1 Introduction

Monte-Carlo Tree Search (MCTS) [9, 6, 5] is a recent algorithm for decision-making, in a discrete, observable and uncertain environment, with finite horizon. It can roughly be described as a reinforcement learning algorithm. This algorithm has been particularly successful in performing abstract games [8, 10, 4, 1, 14], but has moderate results in complex games due to implementations caveats. Previous works have been done on complex games, but authors often work on a limited version of the game, generally by using only a subset of the rules. For instance, in *the Settlers of Catan* board game [13] no trade interaction between players was specified, or in *Dominion* [15] only a subset of the possible cards was used. However, more recently a competitive MCTS player has been designed for the game of 7wonders [11]. In this paper, we are interested in having an artificial intelligence able to play a real game, at an amateur level, but with the full rules even when the search space becomes huge.

In this paper, we first present the game of *Mr Jack* in Section 2, before presenting the MCTS algorithm in Section 3. Then, in Section 4, we present the difficulties one can have for designing an artificial intelligence for the game of *Mr Jack*. In Section 5, we present the experiments and the results we get for the creation of an MCTS player. Conclusion and future works are then discussed.

2 The game of "Mr Jack"

In this section, we briefly sum up the rules of the *Mr Jack* board game. *Mr Jack* is a 2-player board game which aims at recreating the Jack the Ripper story. 8 characters are used throughout the game and one is Jack.

One of the 2 players is Jack the Ripper while his opponent plays the investigator. Jack's goal is to escape the investigators taking profit of the darkness while his opponent has to catch him. Jack is **the only one** to know who he is among the 8 starting characters.

The game-board features the Whitechapel district and is divided into several hexagonal areas called hexes. Some of the gaslights are lit and all the hexes next to one of these are considered as illuminated, and if a character stops by a lit area he/she is considered as a *seen* character. This rule also applies if two characters are on two adjoining hexes. The other characters (*i.e.* that are not on an illuminated area or are not close to another character) are considered to be in the darkness and so to be unseen.

Each player will embody several characters. The game is divided into several turns where each player embodies one of the 8 characters. At each turn, fours characters are randomly drawn and used (two by the investigators and two by Jack). Each character can make a move from 1 to 3 hexes (see Fig. 1 for a graphical view of the board). Each character has the ability to move across the street hexes (grey hexes), the other hexes (building) cannot be crossed. When a character enters a manhole hex that is open, he can use 1 movement point to go to any other hex that has an open manhole.

At the next turn, the four remaining characters are played. Note that unlike many 2-player games, each player does not play alternatively. The first player (A) chooses a character and plays it, then player (B) chooses two characters and plays them. Finally, A plays the last remaining character. For the next turn, the play order is B A A B.



Fig. 1. Snapshot of the *MrJack* computer game. On the left, four possible characters are presented and the current player must select one of them.

Figure 1 gives an overview of the board as well as an example of seen and unseen characters. Four characters can be seen (standing next to a gas-lit) while the four others remain unseen.

At the end of every other turn, there is a "call for witness". It means that the Jack player must announce whether or not the Jack character (*i.e.* remember that he is the only one to know who Jack is) is visible or not. Once the call for witness has been made, the investigators' side can possibly clear some of the characters.

The game goes on for eight turns. When the detective puts a character on the same hex as another investigator/character and rightfully accuses him, the detective player wins. When a character is accused falsely, or Jack escapes Whitechapel or he manages to stay out of the hands of the police during the 8 turns, the Jack player wins.

So, the game can roughly be summed up as a deduction game. The investigator will try to narrow down his search by splitting the innocent or guilty characters into smaller and smaller sets while Jack will try to get the investigator confused by preventing him from building these sets.

However to make things much more complex each character has a special ability that modifies the rules and/allow special moves.

For instance,

- Sherlock Holmes draws a card to clear one of the eight characters.
- John Watson must shine his lantern in a certain direction and illuminate all the hexes in that direction.
- Inspector Lestrade must move a police cordon that prohibits some exits on the board.
- Sergeant Goodley must whistle and force some characters to move one or more hexes closer to him.
- ...

The interested reader may refer to the full rules of *Mr Jack* at this location `http://hurricangames.com/datapdf/device/mr-jack-london_rules_en.pdf`.

Unlike many board games, *Mr Jack* is quite an easy game to learn. Both players use the same tokens throughout the game but have different goals. And besides a simple aspect, *Mr Jack* is a deep and complex game that remains a challenge for IA applications. One of the challenges is to consider how a move of one character will affect the possible moves of many other pieces over the next turns and how to prevent the opponent to guess his future moves.

3 Monte-Carlo Tree Search

In this section we first present a description of the Monte-Carlo Tree Search algorithm, before detailing more specifically the Upper Confidence Bound implementation.

3.1 MCTS description

The Monte-Carlo Tree Search algorithm has been recently proposed for decision-making problems [9, 6, 5]. MCTS has been applied to numerous applications, and in particular in games [8, 10, 4, 1, 14].

The strength of MCTS is that this family of algorithms performs a lot of random simulations in order to evaluate a situation. Then, their use is especially relevant for applications for which it is hard to design an evaluation function. An evaluation function is a function which gives a score from a situation [12]. Alpha-beta techniques (with the different improvements) are state-of-the-art algorithms when such a function exists. In other games, MCTS is now considered as a standard. The most known implementation of MCTS is Upper Confidence Tree (UCT). We now present this particular algorithm.

3.2 UCT description

Let us first define two functions:

- $s' =$`mc(`s`)` which plays a uniform random move m from the situation s and returns the new position s' reached from s when m is played.
- $score =$ `result(`s`)` which returns the final result of the situation s. s must be a final situation (no decision can be made from s). In games, such a reward is often either a loss (0) or a win (1).

The idea is to build an imbalanced partial subtree by performing many random simulations from the current state, and to bias theses simulations toward those giving good results. The construction is then done incrementally and consists of four different parts : *selection*, *expansion*, *simulation* and *backpropagation* as illustrated in Fig. 3. Sometimes in the literature, MCTS algorithm is presented in three different parts: the difference is only that the backpropagation part is presented with the expansion part.

The *descent* part is done by using a bandit formula, i.e. by choosing the node j among all possible nodes C^s which gives the best reward according to the formula :

$$s' \leftarrow \underset{j \in C^s}{\arg\max} \left[\bar{x}_j + K_{UCT} \sqrt{\frac{ln(n_s)}{n_j}} \right],$$

with $\bar{x}_j$ the average reward for the node j, it is the ratio of the number of wins over the number of simulations, n_j the number of simulations for the node j, n_s is the number of simulations in s, and $n_s = \sum_j n_j$.

K_{UCT} is called the exploration parameter and is used to tune the trade-off between exploitation and exploration. At the end of the *descent* part, a node, which is outside the subtree is reached. First, this new node is added to the subtree; this is the *expansion* part. Because this is a new node, there is no information yet, then, in order to evaluate it (as a reward is needed), a *playout* is performed. A playout is a random game: for both players, each decision is taken randomly until the end of the game is reached. This corresponds to the *simulation* part. Eventually, the reward is updated in all the nodes in the subtree which have been crossed during the selection part. This last step is called *backpropagation*.

The algorithm is presented in Alg.1.

Algorithm 1 MCTS

argument node s, MCTS subtree $\hat{T}$
while there is some time left **do**
  $s' \leftarrow s$
  Initialization: $game \leftarrow \emptyset$
  // *DESCENT*
  while s' in $\hat{T}$ and s' not terminal **do**
    $s' \leftarrow \arg\max_{j \in C^{s'}} [\bar{x}_j + K_{UCT}\sqrt{\frac{ln(n_{s'})}{n_j}}]$
    $game \leftarrow game + s'$
  $S \leftarrow s'$
  // *EVALUATION*
  while s' is not terminal **do**
    $s' \leftarrow mc(s')$
  $r = \text{result}(s')$
  // *GROWTH*
  $\hat{T} \leftarrow \hat{T} + S$
  for each s in $game$ **do**
    $n_s \leftarrow n_s + 1$
    $\bar{x_s} \leftarrow \frac{(\bar{x_s} * (n_s - 1) + r)}{n_s}$

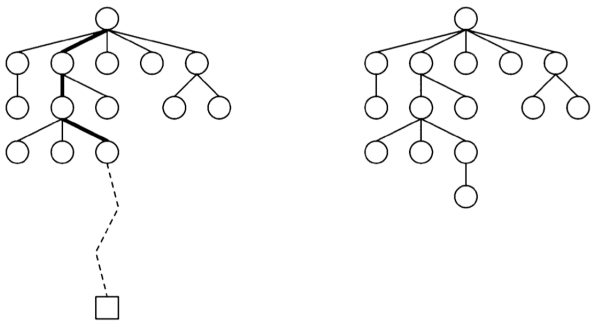


Fig. 2. Illustration of the MCTS algorithm from [3]. Circles represent nodes and the square represents the final node. On the left subtree, the descent part (guided by the bandit formula in the UCT implementation) is illustrated in bold. The evaluation part is illustrated with the dotted line. The subtree after the growth stage is illustrated on the right: the new node has been added.

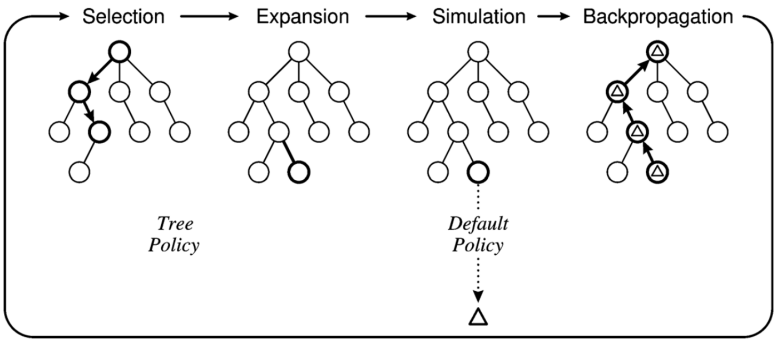


Fig. 3. One iteration of the general MCTS approach from [3]

.

4 MCTS and Mr Jack

The main challenge for applying MCTS to the *Mr Jack* board game lies in several aspects. First, the number of states may become really huge. Second, as explained in Section 2, one decision consists, in fact, of three steps:

- Choosing one card (corresponding to one character).
- Moving the character to a reachable location on the board.
- Applying the character's power.

Eventually, in order to get a decent artificial intelligence, game studies of MCTS have shown that the inclusion of domain knowledge can significantly improve the performance of the algorithm [5, 2].

These main difficulties and the possible add of expert knowledge are now discussed.

4.1 The tree size and the branching factor

The combinatorial complexity of the *Mr Jack* game can be roughly estimated with the game tree size. The game tree size can subsequently be estimated by B^L, where B is the average branching factor of the game, and L is the game length.

As introduced in section 2, a move in the game consists in each character making a move from one to three hexes. It is a bit difficult to evaluate how many hexes can be reached from a given position as a character may interfere with another character (by blocking a hex), a wall or a sewer can be opened or closed (thus opening or closing new short-cuts). We ran many simulations to evaluate the number of hexes that can be reached by one move. On average, a character can reach about 12 new hexes.

There are eight turns in the game and during each turn, four characters (among eight) are played. For the next turn, the four remaining characters are played. In our implementation a node is a possible move for each character, it means that for the first character in the Monte-Carlo tree, 48 nodes are created (4 possible characters $\times$ 12

possible moves), for the second among three remaining characters about 36 nodes are created (3 characters $\times$ 12 possible moves) and so forth until the second turn is reached. At this step, the last four characters are played leading to a tree of size $4 \times 12 \times 3 \times 12 \times 2 \times 12 \times 12 = 497664 \sim 500000$.

When two new turns are to be examined, we face a new situation where the 4 characters have not be selected yet. So all 8 characters must be considered leading to a number of possible nodes of $8 \times 12 \times 7 \times 12 \times 6 \times 12 \times 5 \ldots \times 12 = !8 \times 12^8 = 1.7^{13}$

An estimation of the game tree size for the 8 turns can be estimated by the following formula:

$$500000 \times 1.7^{13^3} = 2.5 \times 10^{44} \quad (1)$$

Table 1(from [2]) provides the value of B^L for some well known games including the game of Go.

Game	Checkers	Othello	Chess	Go
B^L	10^{32}	10^{58}	10^{123}	10^{360}

Table 1. B^L estimation

A correlation between the size of the game tree and the difficulty to build a decent AI is known. However, equation 1 is only **a very low** lower bound. As explained in section 2 each character may/must use his power and this power may interfere with all characters.

Let us have a look at two different characters and see how their power can affect the game tree size. If we look closely at the *Miss Stealthy* character she can make a move between 1 to 4 hexes and she can cross during her move any hex including building hex. Actually, from our simulations she usually can reach almost any hex of the board. The board is actually made of 54 hexes (let us say 50 hexes to make the computations easier). It means that each time this character is used the branching factor is around 50 instead of 12.

If we look closely at the *Sergeant Goodley* character, the branching factor reaches new heights. This character has some kind of whistle and he must use it to call for help. Then, you get 3 movement points to use as you may slot into one or more characters in order to bring them closer to *Sergeant Goodley*. Moreover, this ability can be used before or after the movement. It means in the MCTS tree, each time *Sergeant Goodley* is used the number of nodes we have to create explode. Each character gets between 1 and 3 movement points, that means that on average some 12 hexes can be reached. Overall for this given character $(12^7) \times 2 = 71663616$ new nodes are to be created (12 possible hexes, 7 characters but *Sergeant Goodley* and before or after the movement).

It is quite obvious that we face a difficult problem and that even the MCTS approach must be used with caution to cope with this huge search space.

4.2 Adding expert knowledge

In *Mr Jack* the domain knowledge is mostly driven by the powers that any character may/must use during his turn. This is quite sticky as the use of powers can change the whole set up of the game and interfere with many if not all characters. So getting a decent knowledge domain in *MrJack* implies an efficient use of each power.

In our implementation, we investigated several ways of including domain knowledge: an *implicit* approach where each power increases the branching factor of the nodes when it is used and a *specific* domain knowledge. We call this approach **pure** Monte-Carlo as the specific domain knowledge is fully embedded in the MC tree.

4.3 How to deal with the particular decisions encountered in this game ?

As we have previously seen it, in *MrJack* a decision consists of several parts. Precisely we have to take into account

- The choice of one character among the possible cards.
- Once the character is chosen, we have to move it to a reachable position on the board.
- Finally, choose to apply its corresponding power (which is different for each character).

How to deal with this kind of decisions is not an easy task. Of course, the most accurate way should be to treat all decisions separately: one decision is one character to one location with one power (the **pure** MCTS approach). The problem with this representation is that the branching factor becomes by far too large (see section 4.1).

Another possibility is to represent a decision as a character and one location on the board and to choose the power randomly; hoping that the impact of the position of the character on the board is more important than the impact of the power. This is a rather "crude" but "classical" way of dealing with the branching factor.

A third way of coping with the specific domain knowledge is to get some artificial intelligence for using the character ability. A simple heuristic may be used each time a character has to use his power, and a "best" move is returned. In order to keep the overall computation time to a reasonable time the heuristic for each character must be rather fast. For instance, a list of guilty and innocent characters is built as well as a list of visible and non-visible characters. Each time a character is cleared it moves from the guilty list into the innocent list. The use of the heuristic is based on both these lists. It only aims at splitting the characters of the guilty/innocent lists towards a visible or non-visible.

For example, the heuristic for *Sergeant Goodley* brings the characters from one of these two lists closer to a gaslight to try to make them visible or non-visible.

These different representations have been experimented in Section 5.

5 Experiments and results

5.1 Experimental setting

We chose to experiment three versions of MCTS according to specifications introduced in the previous section.

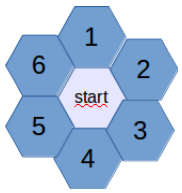


Fig. 4. Graphical view of 6 possible moves for a given character.

– **V1** would be roughly considered as a pure MCTS scheme. The tree is fully developed and explored according to the standard MCTS rules.
A node is created for every possible position of every character **and** possibly and the possible results of each power. Let's take an example to make the things clear. If the *Sir William Gull* character is to be played, (Mr Gull may exchange his location with the location of any character), and for instance, 6 moves are possible (see Fig 4). **V1** will exhaustively create nodes for all 6 moves and for each result of *Sir William Gull's* special ability.
In fig 5 you can see that there a first node is created for move 1 and for switching *Gull* with character number 2, the second node is for the same move and for switching *Gull* with character number 3, and so on.

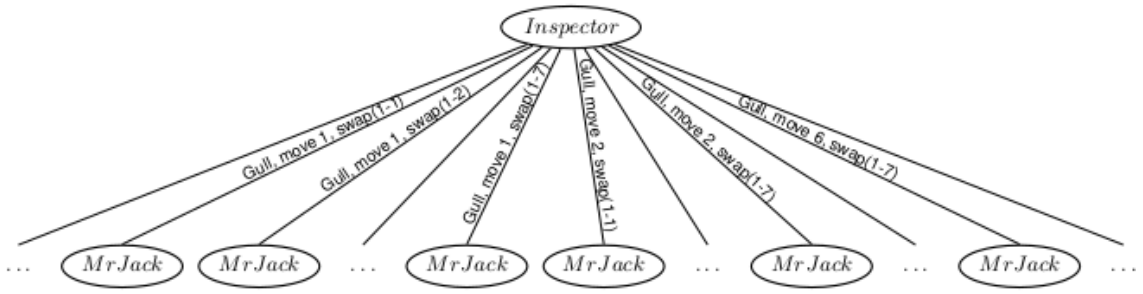


Fig. 5. Graphical view of the **V1** scheme for *Sir William Gull*.

In this case when dealing with the *Sir William Gull* character for these 6 possible moves, 6×8 nodes need to be created. (8 characters are available as it is possible for *Gull* not to use his ability, that means that he switches his position with himself).

– **V2** would be considered as a slightly improved version of **V1**. When a node has to be created inside the MCTS tree and that at this node a special ability can/must be played, only one random node is generated for the ability. In a word only the movement is taken into account. It means that, for instance if the *Sir William Gull* character is to be played, one of the eight possible characters (including Gull) is randomly selected to switch his position with Gull.
In fig 6 you can see that only 6 nodes for *Gull* are created, one node for each move. The special ability is randomly chosen. It is quite obvious that such a random choice for the ability may seriously impede the AI overall level but it allows to greatly reduce the search space.

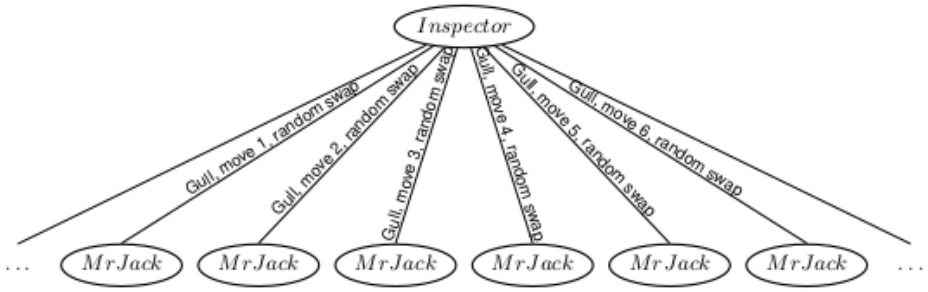


Fig. 6. Graphical view of the **V2** scheme for *Sir William Gull*.

In this case when dealing with the *Sir William Gull* character for these 6 possible moves, only 6 nodes need to be created. A random character is randomly selected to switch his position with *Gull's*.

– **V3** is the most advanced *MrJack* version. Instead of randomly applying the character's ability, a set of rules is used to, hopefully, select one of the best ability of the character. These rules are *a priori* rules that we built and extracted.
These rules have been coded using the *LUA* language. *LUA* is a lightweight programming language designed as a scripting language. LUA is quite flexible and many interpreters for the *LUA* language exist.
We chose the *LUA* language as the rules for our expert knowledge program can be tuned off-line. They are easy to understand and can possibly be refined without being compiled. Furthermore it is not necessary to know the insight of the Mrjack code to modify the rules.
There is a *LUA* script for each character. In most scripts, the program iterates through the different characters to evaluate how far they are from the calling character. The scripts then make a decision based on these information.

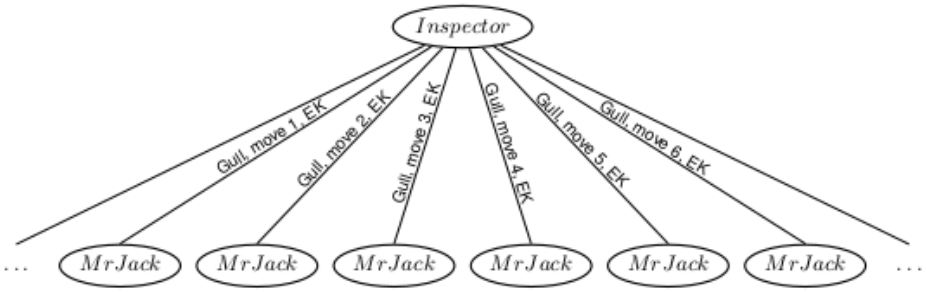


Fig. 7. Graphical view of the **V3** scheme for *Sir William Gull*.

Based on our experiments, these scripts do not add a lot of extra computation time. For instance, for *Sergeant Goodley*, the script tries to bring some characters closer or away to lit hexes so that they can possibly be cleared or not at the end of the call. The number of nodes for the *Gull* character remains as low as the number of nodes needed in the **V2** version.

5.2 Results and Comparisons

First, we experiment the three versions of making decisions. Results are summarized in Table 2. As we expected, the third version (V3, with knowledge for choosing the power) is the best one. The second version (V2) is roughly at the same level as the first one (V1). This shows that both power and move are an important step in making a decision.

For this experiment, we have chosen to compare versions with fixed times (1 second and 10 seconds) for each version (instead of a fixed number of simulations) because the heuristic is more time costly than the random version.

As we have seen that the third version of the MCTS algorithm is the best one, we use it for all the remaining experiments.

	Wins of V3 vs V1	wins of V3 vs V2
1s	$83\% \pm 3.76\%$	$82\% \pm 3.84$
10s	$78\% \pm 4.14\%$	$83\% \pm 3.76\%$

Table 2. Percentage of wins of V3 against V1 and V2. Each experience is the average of 100 games (splited in both Jack and the inspector). As we expected the third version (V3) is by far the best version.

5.3 Tuning of the K_{UCT} parameter

In this experiment, we have to tune the critical parameter K_{UCT}. We have to take into account that this parameter could be different according to the player (Jack or the inspector). Each experiment is done with 10000 simulations and 200 games are performed in order to have a small standard deviation. Table 3 corresponds to the results from the point of view of the inspector, Table 4 corresponds to the results from the point of view of Mr Jack K_{UCT} around 0.6 seems to be a good value for both Mr Jack and the inspector points of view.

K_{UCT} (Inspector)	K_{UCT} (Jack)	Inspector wins
0.1	0.2	$58.0\% \pm 3.49\%$
0.1	0.3	$58.0\% \pm 3.49\%$
0.1	0.4	$53.5\% \pm 3.53\%$
0.4	0.5	$58.5\% \pm 3.48\%$
0.5	0.6	$62.5\% \pm 3.42\%$
0.6	0.7	$69.5\% \pm 3.26\%$
0.6	0.8	$68.0\% \pm 3.30\%$
0.6	0.9	$72.0\% \pm 3.17\%$

Table 3. Tuning of K_{UCT}. 10 000 simulations per move and V3 algorithm.

K_{UCT} (Inspector)	K_{UCT} (Jack)	Jack wins
0.2	0.1	$57.0\% \pm 3.50\%$
0.3	0.1	$54.0\% \pm 3.52\%$
0.4	0.1	$62.5\% \pm 3.42\%$
0.5	0.4	$64.0\% \pm 3.39\%$
0.6	0.5	$65.0\% \pm 3.37\%$
0.7	0.6	$65.0\% \pm 3.37\%$
0.8	0.6	$62.0\% \pm 3.43\%$
0.9	0.6	$62.5\% \pm 3.42\%$

Table 4. Tuning of K_{UCT}. 10 000 simulations per move and V3 algorithm.

5.4 Scaling on the algorithm

MCTS algorithms are known for having particular scaling properties [7, 14]. When the number of simulations increases, their impact becomes less important. The most common explanation is that there are some issues that the algorithm is not able to deal with whatever the number of simulations is. Then the algorithm reaches a plateau and increasing the number of simulations is useless. Tables 5, 6 and 7 show that this behaviour holds true for this game.

# simulations for Inspector	# simulations for Jack	Wins
50	50	$87.0\% \pm 2.38\%$
100	100	$75.0\% \pm 3.06\%$
200	200	$78.0\% \pm 2.92\%$
400	400	$74.5\% \pm 3.08\%$
800	800	$69.5\% \pm 3.26\%$
1600	1600	$66.5\% \pm 3.34\%$
3200	3200	$64.5\% \pm 3.38\%$

Table 5. Scaling of the V3 algorithm. Point of view of the inspector. We can see that playing the inspector is easier. When the level of the algorithm is higher the game becomes more and more balanced.

# simulations for Jack	# simulations for Inspector	Wins
100	50	$27.5\% \pm 3.15\%$
200	100	$26.0\% \pm 3.11\%$
400	200	$28.0\% \pm 3.18\%$
800	400	$34.5\% \pm 3.36\%$
1600	800	$31.5\% \pm 3.29\%$
3200	1600	$42.5\% \pm 3.50\%$
6400	3200	$48.5\% \pm 3.53\%$
12800	6400	$40.0\% \pm 3.46\%$

Table 6. Scaling of the V3 algorithm. Point of view of Mr Jack. 2N simulations against N. As expected, the advantage of having more simulations decreases with the number of simulations.

# simulations for Inspector	# simulations for Jack	Wins
100	50	$88.5\% \pm 2.26\%$
200	100	$81.0\% \pm 2.77\%$
400	200	$84.5\% \pm 2.56\%$
800	400	$83.5\% \pm 2.62\%$
1600	800	$71.0\% \pm 3.21\%$
3200	1600	$72.0\% \pm 3.18\%$
6400	3200	$64.5\% \pm 3.38\%$
12800	6400	$60.5\% \pm 3.46\%$

Table 7. Scaling of the V3 algorithm. Point of view of the inspector. 2N simulations against N. Results are similar as for the point of view of Mr Jack.

6 Conclusion and future works

We can validate in the case of the game of Mr Jack the efficiency of some well-known techniques coming from abstract games. This is another demonstration of the genericity of the Monte-Carlo Tree Search approach and we can note that some properties of the algorithm, such as the scaling perform the same way. It could really be interesting to test other known improvements such as progressive widening and Bersntein rule. The use of the famous Rapid Action Value Estimate (RAVE) improvement is not possible in this game, as there is no possibility of having a permutation of moves.

An important point should be to deal with the addition of an evaluation function. We have used expert knowledge for the decision-making process, but it could be also good to use this heuristic to evaluate a situation. In the *simulation* step, instead of doing a random simulation, one can only play 5 or 6 random moves, and then use the evaluation function [10]. Using determinization to simulate possible future cards in the game could also be interesting.

References

1. Arneson, B., Hayward, R.B., Henderson, P.: Monte-Carlo tree search in Hex. Computational Intelligence and AI in Games, IEEE Transactions on 2(4), 251–258 (2010)
2. Bouzy, B., Chaslot, G.: Monte-carlo go reinforcement learning experiments. In: IEEE Symposium on Computational Intelligence and Games. pp. 187–194 (2006)
3. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte-Carlo tree search methods. Computational Intelligence and AI in Games, IEEE Transactions on 4(1), 1–43 (2012)
4. Cazenave, T.: Monte-Carlo Kakuro. In: van den Herik, H.J., Spronck, P. (eds.) ACG. Lecture Notes in Computer Science, vol. 6048, pp. 45–54. Springer (2009), `http://dblp.uni-trier.de/db/conf/acg/acg2009.html#Cazenave09`
5. Chaslot, G., Saito, J.T., Bouzy, B., Uiterwijk, J., Van Den Herik, H.J.: Monte-Carlo strategies for computer Go. In: Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium. pp. 83–91 (2006)
6. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: Computers and games, pp. 72–83. Springer (2007)
7. Gelly, S., Hoock, J.B., Rimmel, A., Teytaud, O., Kalemkarian, Y., et al.: On the parallelization of Monte-Carlo planning. In: ICINCO (2008)
8. Gelly, S., Silver, D.: Combining online and offline knowledge in uct. In: Proceedings of the 24th international conference on Machine learning. pp. 273–280. ACM (2007)
9. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Machine Learning: ECML 2006, pp. 282–293. Springer (2006)
10. Lorentz, R.J.: Amazons discover Monte-Carlo. In: Computers and games, pp. 13–24. Springer (2008)
11. Robilliard, D., Fonlupt, C., Teytaud, F.: Monte-carlo tree search for the game of "7 wonders". Computer Game workshop of European Conference on Artificial Intelligence (ECAI) (2010)
12. Shannon, C.E.: Xxii. programming a computer for playing chess. Philosophical magazine 41(314), 256–275 (1950)
13. Szita, I., Chaslot, G., Spronck, P.: Monte-Carlo tree search in Settlers of Catan. In: Advances in Computer Games, pp. 21–32. Springer Berlin Heidelberg (2010)

14. Teytaud, F., Teytaud, O.: Creating an upper-confidence-tree program for Havannah. Advances in Computer Games pp. 65–74 (2010), http://hal.inria.fr/inria-00380539/en/
15. Winder, R.K.: Methods for approximating value functions for the Dominion card game. Evolutionary Intelligence (2013)