# Monte-Carlo Tree Search in Backgammon

François Van Lishout[1], Guillaume Chaslot[2], and Jos W.H.M. Uiterwijk[2]

[1] University of Liège, Montefiore Institute, B28, 4000 Liège, Belgium,
vanlishout@montefiore.ulg.ac.be
[2] Universiteit Maastricht, Maastricht ICT Competence Center (MICC), Maastricht, The Netherlands, {G.Chaslot, uiterwijk}@micc.unimaas.nl

**Abstract.** Monte-Carlo Tree Search is a new method which has been applied successfully to many games. However, it has never been tested on two-player perfect-information games with a chance factor. Backgammon is the reference game of this category. Today's best Backgammon programs are based on reinforcement learning and are stronger than the best human players. These programs have played millions of offline games to learn to evaluate a position. Our approach consists rather in playing online simulated games to learn how to play correctly in the current position.

## 1 Introduction

Monte-Carlo Tree Search has among others been applied successfully to the game of Go [5, 7], the Production Management Problem [4], the game of Clobber [9], Amazons [10], and the Sailing Domain [9]. These games are either one-player games with a chance factor or two-player games without a chance factor. It seems thus natural to try this approach on a two-player game with a chance factor. The game of Backgammon is the most famous representative of this category and is therefore a good testbed. Furthermore, thanks to the works of Tesauro, it has already been proved that a very strong AI can be written for this game [11]. The actual approach is to train a neural network by playing millions of offline games. In this paper we present a contrasting approach consisting of using only online simulated games from the current position.

## 2 The game of Backgammon

Backgammon is a board game for two players in which checkers are moved according to the roll of two dice. The objective of the game is to remove all of one's own checkers from the board. Each side of the board is composed of 12 triangles, called points. Figure 1 gives the initial position from Black's point of view. The points are numbered from 1 to 24 and connected across the left edge of the board. The white checkers move in the clockwise direction and the black checkers in the opposite one. The points 1 to 6 are called Black's *home board* and the points 19 to 24 White's home board.

### 2.1 Rules of the game

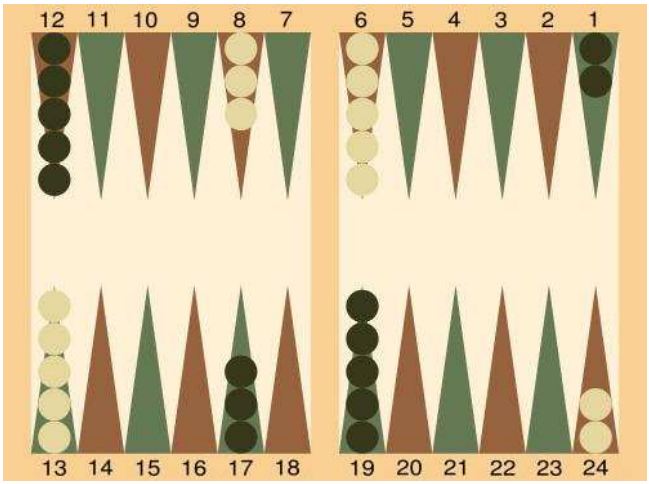


**Fig. 1.** The initial position of a Backgammon game.

Initially, each player rolls one die and the player with the higher number moves first using both dice. The players then alternately move, rolling two dice at the beginning of each turn. Let's call $x$ and $y$ the number of pips shown on each dice. If possible, the player must move one checker $x$ points forward and one checker $y$ points forward. The same checker may be used twice. If it is impossible to use the dice, the player passes its turn. If $x = y$ (doubles), each die must be used twice.

It is forbidden to move a checker to a point occupied by more than one opponent's checker (the point is *blocked*). If a checker lands on a point occupied by exactly one opponent's checker, the latter is taken and placed on the middle bar of the board. As long as a player has checkers on the bar, he cannot play with his other checkers. To free them, he must play a die which allows him to reach a point of the opponent's home board that is not blocked (e.g., a die of 1 permits to reach the 1-point/24-point and a die of 6 the 6-point/19-point).

When all the checkers of a player have reached the home board, he can start removing them from the board. This is called *bearing off*. A checker can only go out if the die number is the exact number of points to go out. However, there is an exception: a die may be used to bear off checkers using less points if no other checkers can be moved with the exact number of points. If a player has not borne off any checker by the time his opponent has borne off all, he has lost a gammon which counts for double a normal loss. If the losing player still has checkers on the bar or in its opponent's home board, he has lost a backgammon which counts for triple a normal loss [2].

Before rolling the dice on his turn, a player may demand that the game be played for twice the current stakes. The opponent must either accept the new stakes or resign the game immediately. Thereafter, the right to redouble belongs exclusively to the player who last accepted a double.

### 2.2 Efficient Representation

The choice of an efficient representation for a backgammon configuration is not as trivial as it seems. It must be efficient to do the following operations:

1. determine the content of a particular point,
2. compute where the white/black checkers are,
3. decide if a checker can move from point $A$ to point $B$,
4. compute if all the white/black checkers are in the home board, and
5. actualize the representation after a move from point $A$ to point $B$.

The straight-forward idea of using simply an array which gives the number of checkers present in each point is not sufficient. This representation has in fact the drawback that the second and fourth operations require to traverse all the array. Even if the latter has only a size of 24, those operations will be performed millions of time and should go faster. For this reason, we use:

- An array $PointsContent[25]$, where the $i^{th}$ cell indicates how many checkers are on the $i^{th}$ point. $+x$ indicates that $x$ black checkers are present, $-x$ that $x$ white checkers are present, and 0 that the point is empty.
- A vector $BlackCheckers$ whose first element is the number of points containing black checkers and the other elements are the corresponding point number in **increasing** order.
- A vector $WhiteCheckers$ whose first element is the number of points containing white checkers and the other elements are the corresponding point number in **decreasing** order.
- Two integers $BlackCheckersTaken$ and $WhiteCheckersTaken$ which gives the amount of black/white checkers taken.

The first operation can be performed immediately as $PointsContent[i]$ gives the content of point $i$ directly. The second operation takes a time proportional to the length of $BlackCheckers/WhiteCheckers$. The third operation is trivial as it depends only on whether $PointsContent[B]$ is blocked or not. The fourth operation is also immediate: all the black checkers are in the home board if and only if $BlackCheckers[1]$ [3] is greater than 18, and all the white checkers are in the home board if and only if $WhiteCheckers[1]$ is smaller than 7.

The fifth operation requires to adapt $PointsContent[A]$ and $PointsContent[B]$ to their new content which is immediate. The point $A$ must be removed from $Blackcheckers/WhiteCheckers$ if $A$ contains only one own checker. The point $B$ must be added if it was empty or contained one opponent's checker. In this latter case, it must be removed from $WhiteCheckers/BlackCheckers$.

[3] Since $BlackCheckers$ is sorted in increasing order from the second element on, the element indexed 1 is the lowest point.

## 3 Monte-Carlo Tree Search

In this section, we describe the structure of an algorithm based on Monte-Carlo tree search. This is a standard Monte-Carlo Tree Search [7] adapted to include chance nodes.

### 3.1 Structure of MCTS

In MCTS, a node $i$ contains at least the following two variables: (1) the value $v_i$ (usually the average of the results of the simulated games that visited this node), and (2) the visit count $n_i$. MCTS usually starts with a tree containing only the root node. We distinguish two kinds of nodes. *Choice nodes* correspond to positions in which one of the players has to make a choice. *Chance nodes* correspond to positions in which dice are rolled.

Monte-Carlo Tree Search consists of four steps, repeated as long as there is time left. (1) The tree is traversed from the root node to a leaf node ($L$), using a *selection strategy*. (2) An *expansion strategy* is called to store one (or more) children of $L$ in the tree. (3) A *simulation strategy* plays moves in self-play until the end of the game is reached. (4) The result $R$ of this "simulated" game is $+1$ in case of a win for Black, and $-1$ in case of a win for White. $R$ is backpropagated in the tree according to a *backpropagation strategy*. This mechanism is outlined in Figure 2. The four steps of MCTS are explained in more details below.

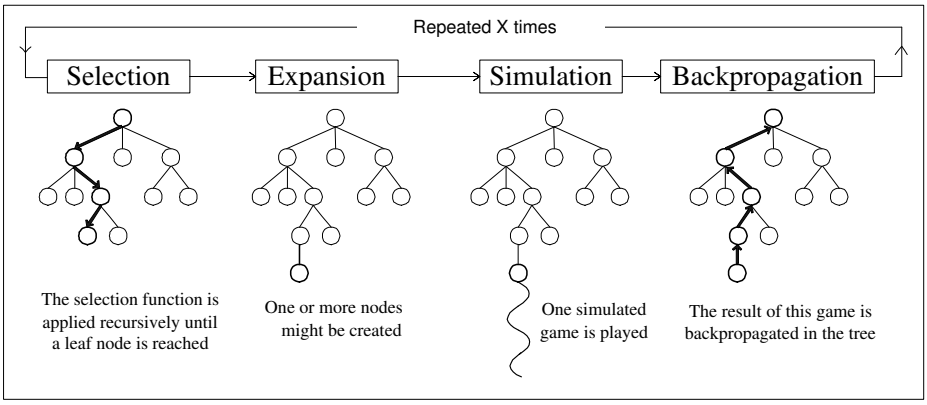


**Fig. 2.** Scheme of a Monte-Carlo Tree Search.

**Selection** is the strategic task that selects one of the children of a given node. In chance nodes, the next move will always be chosen randomly, whereas in a choice node it controls the balance between exploitation and exploration.

On the one hand, the task is often to select the move that leads to the best results (exploitation). On the other hand, the least promising moves still have to be explored, due to the uncertainty of the evaluation (exploration). This problem is similar to the Multi-Armed Bandit problem [6]. As an example, we mention hereby the strategy UCT [9] (UCT stands for Upper Confidence bound applied to Trees). This strategy is easy to implement, and used in many programs. The essence is choosing the move $i$ which maximizes formula 1:

$$v_i + C \times \sqrt{\frac{\ln N}{n_i}} \tag{1}$$

where $v_i$ is the value of the node $i$, $n_i$ is the visit count of $i$, and $N$ is the visit count of the parent node of $i$. $C$ is a coefficient, which has to be tuned experimentally.

**Expansion** is the strategic task that, for a given leaf node $L$, decides whether this node will be expanded by storing some of its children in memory. The simplest rule, proposed by [7], is to expand one node per simulation. The node expanded corresponds to the first position encountered that was not stored yet.

**Simulation** (also called **playout**) is the strategic task that selects moves in self-play until the end of the game. This task might consist of playing plain random moves or – better – pseudo-random moves. Indeed, the use of an adequate simulation strategy has been shown to improve the level of play significantly [3, 8]. The main idea is to play better moves by using patterns, capture considerations, and proximity to the last move.

**Backpropagation** is the procedure which backpropagates the *result* of a simulated game (win/loss) to the nodes it had to traverse to reach the leaf. The *value* $v_i$ of a node is computed by taking the average of the results of all simulated games made through this node. This procedure affects equally chance nodes and choice nodes.

Finally, the move played by the program is the child of the root with the highest visit count.

## 4 The Computation of the Online Random Games

Until today, computing the random games very fast was not really an issue. Since the computation was made offline, winning some microseconds was not that important. The difficult part in playing random games is to be able to compute the list of all the possible moves efficiently. Once this is done, a simple program can be used. It should choose dice values at random, call the functions that find all the possible moves and choose one of them randomly, and repeat for both players until the end of the game is reached.

### 4.1 Computing all the Possible Moves

When the dice are not doubles, the number of possible moves is small and easy to compute. Even if we do have doubles only once every six dice rolls, the number of possible moves will be so much higher that it is very important to take care of computing this very fast.

Let us define a *tower* as a point containing own checkers. The straightforward idea for doubles in many programs is to use a simple loop which traverses the board to find a tower from which it is possible to make a first move. Then, another loop is embedded to find all the possible ways to make a second move from either the same tower or another one not already traversed by the main loop. And so on for the third and fourth move.

The obvious problem of this method is that the embedded loops consume time by traversing the same parts of the board again and again. A less obvious problem of this method is that it generates clones when the same position can be obtained through different moves. They must then be deleted afterwards.

Therefore, we have used a software-engineering approach for solving this problem. We have isolated a simple subproblem and solved it efficiently. Then, we have added more and more features to this subproblem to finally be able to solve the original question. The idea is to compute how often each tower can be used maximally. This depends on the number of checkers of the tower, and of the maximum number of steps that can be done from this tower with the same checker.

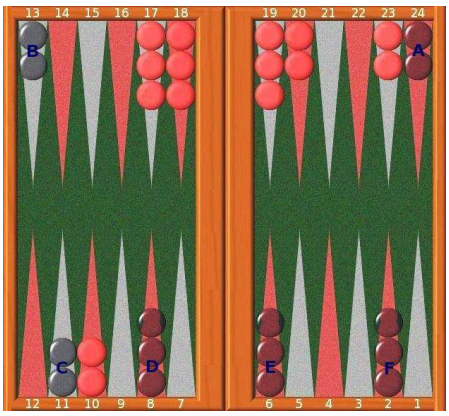


**Fig. 3.** A typical backgammon position. Black has six towers labelled from A to F.

For example, in the position of figure 3, suppose that it is Black's turn and that the dice are 3-3. The towers B and F cannot be used at all, because the 10-point is blocked and because it is not allowed to start the bearing off already.

Tower A can be used maximally two times since each checker can be moved to the 21-point, but none can go to the 18-point which is blocked. Tower E can be used maximally three times since each checker can be used to move one step. Finally, all the dice could be used on the towers C and D so that they can be used maximally four times. This gives us, if we ignore the towers that cannot be used, the following table of the possible towers uses:

| A | C | D | E |
|---|---|---|---|
| 2 | 4 | 4 | 3 |

Our primitive function takes as argument this table and the number of dice that will be used. From this, it computes all the possible ways to use the towers. In our example, it would be the following list:

$AACC, AACD, AACE, AADD, AADE, AAEE, ACCC, ..., DEEE$

The idea of our algorithm is recursive. One basic case is when only one die must be used. In this case, we return the list of all the towers that can be used at least once. The second basic case is when the table is empty. An empty list is returned as there is no tower to use at all. In the other cases, we separate the solutions in two complementary groups:

- The solutions for which the first tower of the table is not used. These are computed by a recursive call where the first entry of the table has been removed and the number of dice has been kept.

- The solutions for which the first tower of the table is used at least once. These are computed by a recursive call where the table has been kept and the number of dice has been decreased by one. Adding one more use of the first tower to all these solutions solves the problem.

Now that we know all the possible towers uses, we still do not know all the possible moves. Indeed, using a tower three times for example can be executed in different ways: move the upper checker of the tower three steps ahead, make two steps with the upper checker and one step with the second checker, or make three one-step moves with three checkers. Note that making one step with the upper checker and then two steps with the second is not another possibility. It leads to the same position as the second possibility.

The second step of our engineering approach was thus to extend the solution found by our primitive function to all the possible real moves. This can be achieved very fast. The idea is again recursive. If the list of the tower moves is empty, the list of the real moves is also empty. Else, we compute all the possible real moves corresponding to the first tower move of the list. Then we compute recursively all the other possible real moves and add the new ones in front.

The computation of all the possible real moves corresponding to a particular tower move has been precomputed manually. This means that the computer does not have to waste time by computing them using an algorithm, because we have coded all the elementary cases one by one. The code is thus a long cascade of $if..then..else$ statements, but it executes extremely fast.

Now we have a function that finds all the possible moves when we are free to move as we want. However, when some of our own checkers are taken, we are not. We must first reenter the checkers from the bar and can only then choose the remaining moves freely. As we are in the case of doubles, deciding if we can reenter checkers at all or not is trivial. It is possible if and only if the $x$-th point of the opponent's home board is not blocked, where $x$ is the dice value. And if it is possible to reenter one, it is also possible to reenter two, three or four (on the same point).

## 5 McGammon 1.0

Our program has been called McGammon, where MC of course stands for Monte Carlo. In this paper we present the very first version of the program. We have made a simplification of the game rules to compute the random games faster and already have early results. The simplification is that when all the checkers are in the home board, the player directly wins the game.

Our program plays thus bad at the approach of the endgame, where it tries to accumulate the checkers on the left-part of the home board. We are currently working on another version which implements good bearing-off strategies. We expect thus this problem to disappear in our next version. The new version will be finished soon and will compete in the Olympiad 2007 competition.

### 5.1 Results

The program is able to play about 6500 games per second on a quad-opteron 2.6 GHz. As a test-bed, we have used the starting position of the game. For the fifteen possible initial dice (no doubles are allowed for the first move), we have run 200,000 random games to choose our first move. We have compared the moves found with the suggestions given online by professional players [1], see table 1.

| Dice roll | McGammon move | % Expert playing this move | Main move played by the experts |
|---|---|---|---|
| 1-2 | 8/6, 24/23 | 0% | 13/11, 24/23 (60.1%) |
| 1-3 | 8/5, 6/5 | 99.9% | 8/5, 6/5 (99.9%) |
| 1-4 | 13/8 | 2.2% | 24/23, 13/9 (74.5%) |
| 1-5 | 24/23, 13/8 | 72.8% | 24/23, 13/8 (72.8%) |
| 1-6 | 13/6 | 0% | 13/7, 8/7 (99.8%) |
| 2-3 | 13/10, 8/6 | 0% | 24/21, 13/11 (51.8%) |
| 2-4 | 13/9, 8/6 | 0% | 8/4, 6/4 (99.8%) |
| 2-5 | 13/8, 8/6 | 0% | 13/11, 13/8 (55.4%) |
| 2-6 | 8/2, 8/6 | 0% | 24/18, 13/11 (81.4%) |
| 3-4 | 13/6 | 0% | 24/20, 13/10 (38.8%) |
| 3-5 | 8/3, 6/3 | 97.8% | 8/3, 6/3 (97.8%) |
| 3-6 | 13/4 | 0% | 24/18, 13/10 (76.4%) |
| 4-5 | 13/9, 13/8 | 30.5% | 24/20, 13/8 (63.1%) |
| 4-6 | 8/2, 6/2 | 27.3% | 24/18, 13/9 (37.0%) |
| 5-6 | 13-2 | 0% | 24/13 (99.3%) |

**Table 1.** Opening moves played by McGammon

Our program has found three strong advises and two openings played by around 30% of the professionals. It has also found one move played by a small minority of experts. For the other nine initial dice, McGammon plays very strange moves, especially for 1-6 and 2-4 where most of the professional players agree on another move than the one found by our program. This is due to the fact that we work with simplified rules and the program tries to put checkers in the home board as soon as possible because it thinks that this is the way to win a game. This explains why it plays a checker on the 6-point as soon as it can. We expect this kind of moves to disappear in our next version.

## 6 Conclusion

In this paper, we have shown the first results of a Monte-Carlo tree search in the game of Backgammon. Even if we have only implemented a simplification of the game, our program is already able to find some expert moves. Despite of the poor quality of the random games, the games played with good starting moves achieve more wins than the ones played from obvious bad moves.

As future work, we will very soon finish the implementation of the bearing-off strategies. Then, we plan to add human-expert strategies to the random games to improve their quality. In the long run, it could also be envisaged to combine learning and Monte-Carlo Tree Search. One possible idea is to learn strategies offline and use them in our random games. A more promising idea is to play random games with a program from the learning community but pilot them with the Monte-Carlo Tree Search algorithm.

## References


1. http://en.wikipedia.org/wiki/backgammon.
2. http://www.bkgm.com/openings.html.
3. Bruno Bouzy. Associating Domain-Dependent Knowledge and Monte-Carlo Approaches within a Go program. *Information Sciences, Heuristic Search and Computer Game Playing IV*, 175(4):247–257, 2005.
4. Guillaume Chaslot, Steven De Jong, Jahn-Takeshi Saito, and Jos W. H. M. Uiterwijk. Monte-Carlo Tree Search in Production Management Problems. In Pierre-Yves Schobbens, Wim Vanhoof, and Gabriel Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 91–98, 2006.
5. Guillaume Chaslot, Jahn-Takeshi Saito, Bruno Bouzy, Jos W. H. M. Uiterwijk, and H. Jaap van den Herik. Monte-Carlo Strategies for Computer Go. In Pierre-Yves Schobbens, Wim Vanhoof, and Gabriel Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91, 2006.
6. Pierre-Arnaud Coquelin and Rémi Munos. Bandit Algorithm for Tree Search. Technical Report 6141, INRIA, 2007.
7. Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proceedings of the 5th Computers and Games Conference*, 2006.
8. Sylvain Gelly, Yizao Wang, Remi Munos, and Olivier Teytaud. Modifications of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA, 2006.
9. Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning*, 2006.
10. Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved Monte-Carlo Search. In *White paper*, 2006. http://zaphod.aml.sztaki.hu/papers/cg06-ext.pdf.
11. Gerald Tesauro. Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3), 1995.