# Playout Policy Adaptation with Move Features

Tristan Cazenave

*PSL-Université Paris-Dauphine, LAMSADE CNRS UMR 7243, Paris, France*

**Abstract**

Monte Carlo Tree Search (MCTS) is the state of the art algorithm for General Game Playing (GGP). We propose to learn a playout policy online so as to improve MCTS for GGP. We also propose to learn a policy not only using the moves but also according to the features of the moves. We test the resulting algorithms named Playout Policy Adaptation (PPA) and Playout Policy Adaptation with move Features (PPAF) on Atarigo, Breakthrough, Misere Breakthrough, Domineering, Misere Domineering, Knightthrough, Misere Knightthrough and Nogo. The experiments compare PPA and PPAF to Upper Confidence for Trees (UCT) and to the closely related Move-Average Sampling Technique (MAST) algorithm.

*Keywords:* Monte Carlo Tree Search, Playout Policy, Machine Learning, Computer Games

## 1. Introduction

Monte Carlo Tree Search (MCTS) has been successfully applied to many games and problems [1]. The most popular MCTS algorithm is Upper Confidence bounds for Trees (UCT) [2]. MCTS is particularly successful in the game of Go [3]. It is also the state of the art in Hex [4] and General Game Playing (GGP) [5, 6]. GGP can be traced back to the seminal work of Jacques Pitrat [7]. Since 2005 an annual GGP competition is organized by Stanford at AAAI [8]. Since 2007 all the winners of the competition use MCTS.

Offline learning of playout policies has given good results in Go [9, 10] and Hex [4], learning fixed pattern weights so as to bias the playouts. AlphaGo [11] also uses a linear softmax policy based on pattern weights trained on 8 million positions from human games and improved with hand crafted features.

The RAVE algorithm [12] performs online learning of moves values in order to bias the choice of moves in the UCT tree. RAVE has been very successful in Go and Hex. A development of RAVE is to use the RAVE values to choose moves in the playouts using Pool RAVE [13]. Pool RAVE improves slightly on random playouts in Havannah and reaches 62.7% against random playouts in Go.

The GRAVE algorithm [14] is a simple generalization of RAVE. It uses the RAVE value of the last node in the tree with more than a given number of playouts instead of the RAVE values of the current node. It was successful for many different games.

Move-Average Sampling Technique (MAST) is a technique used in the GGP program CadiaPlayer so as to bias the playouts with statistics on moves [5, 15]. It consists of choosing a move in the playout proportionally to the exponential of its mean. MAST keeps the average result of each action over all simulations. Moves found to be good on average, independent of a game state, will get higher values. In the playout step, the action selections are biased towards selecting such moves. This is done using the Gibbs (or Boltzmann) distribution.

Predicate Average Sampling Technique (PAST) is another technique used in CadiaPlayer. It consists in associating learned weights to the predicates contained in a position represented in the Game Description Language (GDL).

CadiaPlayer also uses Features to Action Sampling Technique (FAST). FAST learns features such as piece values using TD($\lambda$) [16]. FAST is used to bias playouts in combination with MAST but only slightly improves on MAST.

Playout Policy Adaptation (PPA) [17] also uses Gibbs sampling, however the evaluation of an action for PPA is not its mean over all simulations such as in MAST. Instead the value of an action is learned comparing it to the other available actions for the states where it has been played. PPA is therefore closely related to reinforcement learning whereas MAST is about statistics on moves. Adaptive sampling techniques related to PPA have also been tried recently for Go with success [18].

Later improvements of CadiaPlayer are N-Grams and the last good reply policy [19]. They have been applied to GGP so as to improve playouts by learning move sequences. A recent development in GGP is to have multiple playout strategies and to choose the one which is the most adapted to the problem at hand [20].

A related domain is the learning of playout policies in single-player problems. Nested Monte Carlo Search (NMCS) [21] is an algorithm that works well for puzzles. It biases its playouts using lower level playouts. At level zero NMCS adopts a uniform random playout policy. Online learning of playout strategies combined with NMCS has given good results on optimization problems [22].

Online learning of a playout policy in the context of nested searches has been further developed for puzzles and optimization with Nested Rollout Policy Adaptation (NRPA) [23]. NRPA has found new world records in Morpion Solitaire and crosswords puzzles. Stefan Edelkamp and co-workers have applied the NRPA algorithm to multiple problems. They have optimized the algorithm for the Traveling Salesman with Time Windows (TSPTW) problem [24, 25]. Other applications deal with 3D Packing with Object Orientation [26], the physical traveling salesman problem [27], the Multiple Sequence Alignment problem [28] or Logistics [29]. The principle of NRPA is to adapt the playout policy so as to learn the best sequence of moves found so far at each level.

PPA is inspired by NRPA since it learns a playout policy in a related fashion and adopts a similar playout policy. However PPA is different from NRPA in multiple ways. NRPA is not suited for two player games since it memorizes the best playout and learns all the moves of the best playout. The best playout is ill-defined for two player games since the result of a playout is either won or lost. Moreover a playout which is good for one player is bad for the other player so learning all the moves of a playout does not make much sense. To overcome these difficulties PPA does not memorize a best playout and does not use nested levels of search. Instead of learning the best playout it learns the moves of every playout but only for the winner of the playout.

NMCS has been previously successfully adapted to two-player games in a recent work [30]. PPA is a follow-up to this paper since it is the adaptation of NRPA to two-player games. PPA is an online learning algorithm, it starts from scratch for every position and learns a position specific playout policy each time. The PPA algorithm was first described in [17]. In this paper we extend it to use move features so as to have more specific statistics on moves.

The use of features to improve MCTS playouts has also been proposed in the General Game AI settings [31]. The approach is different from the approach in this paper since features are part of the state and are used to evaluate states. Instead we propose to use features to evaluate moves.

As our paper deals with learning action values it is also related to the

detection of action heuristics in GGP [32].

We now give the outline of the paper. The next section details the PPA and the PPAF algorithms and particularly the playout strategy and the adaptation of the policy. The third section gives experimental results for various games. The last section concludes.

## 2. Online Policy Learning

PPA is UCT with an adaptive playout policy. It means that it develops a tree exactly as UCT does. The difference with UCT is that in the playouts it has a weight for each possible move and chooses randomly between possible moves proportionally to the exponential of the weight.

In the beginning PPA starts with a uniform playout policy. All the weights are set to zero. Then, after each playout, it adapts the policy of the winner of the playout. The way it modifies the weights according to the playout is similar to NRPA. In NRPA the weight of the move of the best playout is increased by a constant $\alpha$ and the weight of the other moves are decreased by a value proportional to the exponential of their weight. PPA does a similar update except that it only adapts the policy of the winner of the playout with the moves of the winner as there is no best playout to follow in PPA.

The NRPA adaptation algorithm is given in Algorithm 1. It reinforces all the moves of the best sequence found so far at a level. The algorithm is given here to highlight the differences with the PPA adaptation algorithm described later. For the sake of completeness we also give the NRPA algorithm in Algorithm 2. It is not suited to multi-player games since players alternate in a game and defining a best sequence is not as simple as in single player games.

The different algorithms we deal with are UCT, PPA, PPAF, MAST and MASTF. The playout code used by all algorithms except UCT is given in algorithm 3. Each move of a playout is chosen with a probability proportional to its associated weight. The $k$ constant has to be tuned for MAST and MASTF and it is always set to 1.0 for PPA and PPAF.

Algorithm 3 takes as parameters the board, the next player and the playout policy. The playout policy is an array of real numbers that contains a number for each possible move. The only difference with a random playout is that it uses the policy to choose a move. Each move is associated to

**Algorithm 1** The NRPA adapt algorithm

adaptNRPA (*policy*, *sequence*)
*polp* ← *policy*
*state* ← *root*
**for** *move* in *sequence* **do**
  *polp* [code(*move*)] ← *polp* [code(*move*)] + $\alpha$
  $z \leftarrow 0.0$
  **for** $m$ in possible moves for *state* **do**
    $z \leftarrow z$ + exp (*policy* [code($m$)])
  **end for**
  **for** $m$ in possible moves for *state* **do**
    *polp* [code($m$)] ← *polp* [code($m$)] - $\alpha * \frac{exp(policy[code(m)])}{z}$
  **end for**
  *state* ← play (*state*, *move*)
**end for**
*policy* ← *polp*

**Algorithm 2** The NRPA algorithm

NRPA (*level*, *policy*)
**if** level == 0 **then**
  **return** playout (root, *policy*)
**end if**
*bestScore* ← $-\infty$
**for** N iterations **do**
  (result,new) ← NRPA($level - 1$, *policy*)
  **if** result $\geq$ bestScore **then**
    bestScore ← result
    seq ← new
  **end if**
  policy ← adaptNRPA (policy, seq)
**end for**
**return** (bestScore, seq)

the exponential of its policy number and the move to play is chosen with a probability proportional to this value.

An important detail of the playout algorithm is the code function. It associates a different integer to all possible moves. In usual PPA and MAST the code represents the move and only the move. In PPAF and MASTF the same move can have different codes that depend on the presence of features associated to the move. For example in BREAKTHROUGH a capture move will not have the same code as a move implying the same starting and arriving squares but not implying a capture.

The overall MCTS algorithm that is the same for PPA, PPAF, MAST and MASTF is given in algorithm 6. It repeatedly calls the UCT algorithm given in algorithm 7 and the adapt function. The adapt function differs for PPA and PPAF and for MAST and MASTF. The PPA and PPAF adapt function is given in algorithm 4. It is the same as the original PPA adapt function. The difference between PPA and PPAF lies in the way moves are coded not in the learning algorithm.

Algorithm 6 starts with initializing the policy to a uniform policy containing only zeros for every move. Then it runs UCT for the given number of playouts. UCT uses the playout algorithm for its playouts. They are biased with the policy. The result of a call to the UCT function is one descent of the tree plus one playout that gives the winner of this single playout. The playout and its winner are then used to adapt the policy using the adapt function. When all playouts have been played the algorithm returns the move that has the most playouts at the root as in usual UCT.

The PPA adaptation algorithm is related to the adaptation algorithm of NRPA. The main difference is that it is adapted to games and only learns the moves of the winner of the playout. It does not use a best sequence to learn as in NRPA but learns a different playout every time. It takes as parameter the winner of the playout, the board as it was before the playout, the player to move on this board, the playout to learn and the current playout policy. It has a parameter $\alpha$ which is the number to add to the weight of the move in the policy. The adapt algorithm plays the playout again and for each move of the winner it biases the policy towards playing this move. It increases the weight of the move of the winner of the playout and decreases the weight of the other possible moves on the current board.

In order to be complete the MAST adaptation algorithm is given in Algorithm 5. The $k$ constant of the playout algorithm is set to one for PPA and it has to be tuned for MAST. The MAST adapt function associates the

statistics of the moves to the codes of the moves. It updates the number of games and the number of wins of the moves played during the playout and computes the weight of the code of a move as the average of the outcome of the playouts where the move was played with the same features.

PPA and PPAF learn move weights faster than MAST and MASTF. MAST and MASTF only update the weights of the moves that have been played in the playout. PPA and PPAF also update the weights of the moves that were once possible in the playout but that have not been played. This feature of PPA and PPAF can prove better when there are a lot of possible codes for the moves. This is the case for example when using elaborate features associated to moves. It can increase a lot the number of codes associated to the same move and make PPAF to the point.

**Algorithm 3** The playout algorithm

playout (*board*, *player*, *policy*)
**while** true **do**
  **if** *board* is terminal **then**
    **return** winner (*board*)
  **end if**
  $z \leftarrow 0.0$
  **for** $m$ in possible moves on *board* **do**
    $z \leftarrow z + \exp(k \times policy\,[\text{code}(m)])$
  **end for**
  choose a *move* for *player* with probability proportional to $\frac{exp(k \times policy[code(move)])}{z}$
  play (*board*, *move*)
  *player* $\leftarrow$ opponent (*player*)
**end while**

## 3. Experimental Results

We played MAST, MASTF, PPA and PPAF against UCT with random playouts. All algorithms use 10,000 playouts. The UCT constant is set to 0.4 for all algorithms as is usual in GGP. Each result is the outcome of a 500 games match, 250 playing first and 250 playing second. We test various values of $\alpha$ so as to optimize PPA and PPAF. In the tables the number inside parenthesis for PPA and PPAF is the $\alpha$ constant. We optimize MAST and

**Algorithm 4** The PPA adapt algorithm

adapt (*winner*, *board*, *player*, *playout*, *policy*)
*polp* ← *policy*
**for** *move* in *playout* **do**
  **if** *winner* = *player* **then**
    *polp* [code(*move*)] ← *polp* [code(*move*)] + $\alpha$
    $z \leftarrow 0.0$
    **for** $m$ in possible moves on *board* **do**
      $z \leftarrow z$ + exp (*policy* [code($m$)])
    **end for**
    **for** $m$ in possible moves on *board* **do**
      *polp* [code($m$)] ← *polp* [code($m$)] - $\alpha * \frac{exp(policy[code(m)])}{z}$
    **end for**
  **end if**
  play (*board*, *move*)
  *player* ← opponent (*player*)
**end for**
*policy* ← *polp*

**Algorithm 5** The MAST adapt algorithm

adapt (*winner*, *board*, *player*, *playout*, *policy*)
**for** *move* in *playout* **do**
  *games* [code(*move*)] ← *games* [code(*move*)] + 1
  **if** *winner* = *player* **then**
    *wins* [code(*move*)] ← *wins* [code(*move*)] + 1
  **end if**
  *policy* [code(*move*)] ← $\frac{wins[code(move)]}{games[code(move)]}$
  play (*board*, *move*)
  *player* ← opponent (*player*)
**end for**
*policy* ← *polp*

**Algorithm 6** The overall MCTS algorithm

```
MCTS (board, player)
for i in 0, maximum index of a move code do
  policy[i] ← 0.0
  games[i] ← 0.0
  wins[i] ← 0.0
end for
for i in 0, number of playouts do
  b ← board
  winner ← UCT (b, player, policy)
  b1 ← board
  adapt (winner, b1, player, b.playout, policy)
end for
return the move with the most playouts
```

MASTF using various values for the $k$ constant. In the tables the number inside parenthesis for MAST and MASTF is the $k$ constant. For each game we test the $8 \times 8$ board size.

The games we have experimented with are:

- ATARIGO: the rule are the same as GO except that the first player to capture a string has won. ATARIGO has been solved up to size $6 \times 6$ [33]. The move feature we use for ATARIGO is to add a code for the pattern surrounding the move. The code takes into account the colors of the four intersections next to the move.

- BREAKTHROUGH: The game starts with two rows of pawns on each side of the board. Pawns can capture diagonally and go forward either vertically or diagonally. The first player to reach the opposite row has won. BREAKTHROUGH has been solved up to size $6 \times 5$ using Job Level Proof Number Search [34]. The best program for BREAKTHROUGH $8 \times 8$ uses MCTS combined with an evaluation function after a short playout [35]. The move feature we use for BREAKTHROUGH is to distinguish between capture moves and moves that do not capture.

- MISERE BREAKTHROUGH: The rules are the same as BREAKTHROUGH except that the first player to reach the opposite row has lost. We use the same move feature as in BREAKTHROUGH.

**Algorithm 7** The UCT algorithm

```
UCT (board, player, policy)
moves ← possible moves on board
if board is terminal then
  return winner (board)
end if
t ← entry of board in the transposition table
if t exists then
  bestValue ← −∞
  for m in moves do
    t ← t.totalPlayouts
    w ← t.wins[m]
    p ← t.playouts[m]
    value ← w/p + c × sqrt(log(t)/p)
    if value > bestValue then
      bestValue ← value
      bestMove ← m
    end if
  end for
  play (board, bestMove)
  player ← opponent (player)
  res ← UCT (board, player, policy)
  update t with res
else
  t ← new entry of board in the transposition table
  res ← playout (board, player, policy)
  update t with res
end if
return res
```

- DOMINEERING: The game starts with an empty board. One player places dominoes vertically on the board and the other player places dominoes horizontally. The first player that cannot play has lost. DOMINEERING was invented by Göran Andersson [36]. Jos Uiterwijk recently proposed a knowledge based method that can solve large rectangular boards without any search [37]. The move feature we use for DOMINEERING is to take into account the cells next to the domino played They can be either empty or occupied. This simple feature enables for example to detect moves on cells that cannot be reached by the opponent. This is an important feature at DOMINEERING.

- MISERE DOMINEERING: The rules are the same as DOMINEERING except that the first player unable to move has won. We use the same move feature as in DOMINEERING.

- KNIGHTTHROUGH: The rules are similar to BREAKTHROUGH except that the pawns are replaced by knights that can only go forward. The first player to move a knight on the last row of the opposite side has won. The move feature we use for KNIGHTTHROUGH is to take into account capture in the move code.

- MISERE KNIGHTTHROUGH: The rules are the same as KNIGHTTHROUGH except that the first player to reach the opposite row has lost. We use the same move feature as in KNIGHTTHROUGH.

- NOGO: The rules are the same as GO except that it is forbidden to capture and to suicide. The first player that cannot move has lost. There exist computer NOGO competitions and the best players use MCTS [38, 39, 40]. We use the same move feature as for ATARIGO.

We do not give results for single-player games since PPA is tailored to multi-player games. Also we do not compare with NMCS and NRPA since these algorithms are tailored to single-player games.

In the context of GGP, the time used by GGP programs is dominated by the generation of the possible moves and by the calculation of the next state. So biasing the playout policy is relatively inexpensive compared to the time used for the interpretation of the rules of the game.

### *3.1.* Atarigo $8 \times 8$

Table 1 gives the results of the different algorithms against standard UCT at Atarigo $8 \times 8$. All algorithms are much better than UCT and they have similar winning percentages.

Table 1: Different algorithms against UCT at Atarigo $8 \times 8$ with 10,000 playouts.

| | | | |
|---|---|---|---|
| MAST | 51.2% (0.5) | 58.4% (1.0) | 67.0% (2.0) |
| MAST | 75.6% (4.0) | 88.2% (8.0) | **94.4% (16.0)** |
| MAST | 92.0% (32.0) | | |
| | | | |
| MASTF | 55.8% (0.5) | 63.0% (1.0) | 77.6% (2.0) |
| MASTF | 95.2% (4.0) | **97.2% (8.0)** | 92.8% (16.0) |
| MASTF | 91.0% (32.0) | | |
| | | | |
| PPA | 71.0% (0.005) | 81.6% (0.01) | 85.6% (0.02) |
| PPA | 89.8% (0.04) | 93.2% (0.08) | 94.4% (0.16) |
| PPA | **96.6% (0.32)** | 93.2% (0.64) | 95.8% (1.28) |
| PPA | 95.6% (2.56) | | |
| | | | |
| PPAF | 71.6% (0.005) | 75.2% (0.01) | 83.2% (0.02) |
| PPAF | 89.0% (0.04) | 91.0% (0.08) | 93.0% (0.16) |
| PPAF | 90.6% (0.32) | **94.4% (0.64)** | 92.0% (1.28) |
| PPAF | 91.6% (2.56) | | |

### *3.2.* Breakthrough $8 \times 8$

Table 2 gives the results of playing different algorithms against standard UCT at Breakthrough. We can see that using move features is an improvement both for MAST and for PPA. PPAF is the best algorithm and it improves much on PPA.

### *3.3.* Misere Breakthrough $8 \times 8$

Table 3 gives the results of playing different algorithms against standard UCT at Misere Breakthrough. In misere games there can be a lot of losing moves. Learning to avoid these moves lead to much more informative playouts and can explain the great performance of adaptive playouts for this kind of games.

Table 2: Different algorithms against UCT at BREAKTHROUGH $8 \times 8$ with 10,000 playouts.

| | | | |
|---|---|---|---|
| MAST | 58.4% (0.5) | 51.6% (1.0) | 60.0% (2.0) |
| MAST | **63.4% (4.0)** | 48.6% (8.0) | 27.6% (16.0) |
| MAST | 20.2% (32.0) | 19.6% (64.0) | |
| | | | |
| MASTF | 51.0% (0.5) | 59.0% (1.0) | 63.8% (2.0) |
| MASTF | **70.2% (4.0)** | 63.2% (8.0) | 37.4% (16.0) |
| MASTF | 22.6% (32.0) | | |
| | | | |
| PPA | 50.0% (0.005) | 54.2% (0.01) | 53.2% (0.02) |
| PPA | 57.2% (0.04) | 56.4% (0.08) | 56.2% (0.16) |
| PPA | **59.2% (0.32)** | 55.8% (0.64) | 49.4% (1.28) |
| PPA | 43.0% (2.56) | | |
| | | | |
| PPAF | 59.4% (0.005) | 59.2% (0.01) | 66.2% (0.02) |
| PPAF | 70.0% (0.04) | 75.8% (0.08) | 78.2% (0.16) |
| PPAF | **81.4% (0.32)** | 72.6% (0.64) | 68.0% (1.28) |
| PPAF | 47.6% (2.56) | | |

Table 3: Different algorithms against UCT at Misere Breakthrough $8 \times 8$ with 10,000 playouts.

| | | | |
|---|---|---|---|
| MAST | 53.2% (0.5) | 65.2% (1.0) | 75.8% (2.0) |
| MAST | 54.4% (4.0) | 47.8% (8.0) | 84.0% (16.0) |
| MAST | 91.4% (32.0) | 94.0% (64.0) | 96.4% (128.0) |
| MAST | **97.6% (256.0)** | 94.8% (512.0) | |
| | | | |
| MASTF | 54.8% (0.5) | 64.0% (1.0) | 77.6% (2.0) |
| MASTF | 52.8% (4.0) | 61.4% (8.0) | 95.2% (16.0) |
| MASTF | 98.8% (32.0) | **99.6% (64.0)** | 98.2% (128.0) |
| MASTF | 99.2% (256.0) | 98.0% (512.0) | |
| | | | |
| PPA | 64.0% (0.005) | 68.4% (0.01) | 54.4% (0.02) |
| PPA | 49.2% (0.04) | 54.8% (0.08) | 65.6% (0.16) |
| PPA | 84.4% (0.32) | 95.2% (0.64) | 98.8% (1.28) |
| PPA | **99.6% (2.56)** | 98.6% (5.12) | 97.8% (10.24) |
| | | | |
| PPAF | 60.0% (0.005) | 68.2% (0.01) | 63.4% (0.02) |
| PPAF | 60.6% (0.04) | 74.2% (0.08) | 89.6% (0.16) |
| PPAF | 99.0% (0.32) | 99.8% (0.64) | **100.0% (1.28)** |
| PPAF | 100.0% (2.56) | | |

Table 4: Different algorithms against PPAF(1.28) at MISERE BREAKTHROUGH $8 \times 8$ with 10,000 playouts.

| | | | |
|---|---|---|---|
| MAST | 0.0% (8.0) | 6.6% (16.0) | 17.4% (32.0) |
| MAST | 24.2% (64.0) | 28.2% (128.0) | **28.2% (256.0)** |
| MAST | 25.8% (512.0) | 25.8% (1024.0) | 5.0% (2048.0) |
| | | | |
| MASTF | 0.0% (8.0) | 12.2% (16.0) | 36.2% (32.0) |
| MASTF | 37.2% (64.0) | 33.4% (128.0) | **37.4% (256.0)** |
| MASTF | 32.4% (512.0) | 37.0% (1024.0) | 12.2% (2048.0) |

As the results for adaptive algorithms are extremely good at MISERE BREAKTHROUGH, we ran another experiment to better distinguish between the different algorithms. We played the algorithms against PPAF(1.28). The results are given in table 4. PPAF is much better than MAST and MASTF.

### *3.4.* DOMINEERING $8 \times 8$

Table 5 gives the results of the different algorithms against standard UCT at DOMINEERING $8 \times 8$. PPAF is clearly the best algorithm and it is much better than MAST and MASTF. PPA is better than MAST at DOMINEERING and PPAF is an improvement over PPA.

### *3.5.* MISERE DOMINEERING $8 \times 8$

Table 6 gives the results of the different algorithms against standard UCT at MISERE DOMINEERING $8 \times 8$. All algorithms are much better than UCT. MAST and PPAF are ahead.

### *3.6.* KNIGHTTHROUGH $8 \times 8$

Table 7 gives the results of playing different algorithms against standard UCT at KNIGHTTHROUGH. We can observe that all algorithms improve much on standard UCT and that PPAF is slightly better.

### *3.7.* MISERE KNIGHTTHROUGH $8 \times 8$

Table 8 gives the results of the different algorithms against standard UCT. As in MISERE BREAKTHROUGH there are a lot of losing moves in MISERE KNIGHTTHROUGH. Learning to avoid them in the playouts is much better than random playouts.

Table 5: Different algorithms against UCT at DOMINEERING $8 \times 8$ with 10,000 playouts.

| | | | |
|---|---|---|---|
| MAST | 52.4% (0.5) | 54.4% (1.0) | 53.8% (2.0) |
| MAST | 58.0% (4.0) | **58.2% (8.0)** | 47.0% (16.0) |
| MAST | 35.6% (32.0) | 25.8% (64.0) | 20.4% (128.0) |
| | | | |
| MASTF | 47.4% (0.5) | 47.6% (1.0) | 49.6% (2.0) |
| MASTF | **52.4% (4.0)** | 35.8% (8.0) | 16.2% (16.0) |
| MASTF | 6.4% (32.0) | 4.0% (64.0) | 4.2% (128.0) |
| | | | |
| PPA | 61.8% (0.005) | 63.0% (0.01) | 73.8% (0.02) |
| PPA | **74.2% (0.04)** | 72.8% (0.08) | 67.0% (0.16) |
| PPA | 67.8% (0.32) | 63.0% (0.64) | 59.8% (1.28) |
| PPA | 47.2% (2.56) | | |
| | | | |
| PPAF | 56.0% (0.005) | 63.4% (0.01) | 68.8% (0.02) |
| PPAF | 75.8% (0.04) | 78.4% (0.08) | **80.4% (0.16)** |
| PPAF | 76.6% (0.32) | 72.0% (0.64) | 60.6% (1.28) |
| PPAF | 52.0% (2.56) | | |

Table 6: Different algorithms against UCT at Misere Domineering $8 \times 8$ with 10,000 playouts.

| | | | |
|---|---|---|---|
| MAST | 47.4% (0.5) | 51.8% (1.0) | 50.4% (2.0) |
| MAST | 50.8% (4.0) | 54.6% (8.0) | 64.0% (16.0) |
| MAST | 79.2% (32.0) | 84.6% (64.0) | 91.4% (128.0) |
| MAST | 92.2% (256.0) | **93.8% (512.0)** | 92.2% (1024.0) |
| | | | |
| MASTF | 48.0% (0.5) | 49.4% (1.0) | 53.4% (2.0) |
| MASTF | 46.6% (4.0) | 56.4% (8.0) | 62.2% (16.0) |
| MASTF | 72.4% (32.0) | 81.8% (64.0) | 88.8% (128.0) |
| MASTF | 89.6% (256.0) | **92.4% (512.0)** | 91.2% (1024.0) |
| | | | |
| PPA | 57.8% (0.005) | 63.2% (0.01) | 64.0% (0.02) |
| PPA | 63.8% (0.04) | 68.2% (0.08) | 73.4% (0.16) |
| PPA | 79.2% (0.32) | 79.8% (0.64) | 81.4% (1.28) |
| PPA | **82.2% (2.56)** | 78.0% (5.12) | 72.2% (10.24) |
| | | | |
| PPAF | 59.6% (0.005) | 59.2% (0.01) | 71.6% (0.02) |
| PPAF | 78.6% (0.04) | 83.6% (0.08) | 88.6% (0.16) |
| PPAF | **93.0% (0.32)** | 91.6% (0.64) | 90.8% (1.28) |
| PPAF | 90.0% (2.56) | | |

Table 7: Different algorithms against UCT at KNIGHTTHROUGH 8×8 with 10,000 playouts.

| | | | |
|---|---|---|---|
| MAST | 56.2% (0.5) | 56.2% (1.0) | 64.4% (2.0) |
| MAST | **78.2% (4.0)** | 69.4% (8.0) | 64.4% (16.0) |
| MAST | 57.2% (32.0) | | |
| | | | |
| MASTF | 56.8% (0.5) | 58.8% (1.0) | 71.0% (2.0) |
| MASTF | **75.0% (4.0)** | 74.8% (8.0) | 61.6% (16.0) |
| MASTF | 52.6% (32.0) | | |
| | | | |
| PPA | 66.8% (0.005) | 70.2% (0.01) | 74.6% (0.02) |
| PPA | **76.8% (0.04)** | 74.4% (0.08) | 72.0% (0.16) |
| PPA | 70.4% (0.32) | 64.8% (0.64) | 63.2% (1.28) |
| PPA | 59.6% (2.56) | | |
| | | | |
| PPAF | 60.8% (0.005) | 68.6% (0.01) | 77.2% (0.02) |
| PPAF | 77.6% (0.04) | 83.2% (0.08) | 80.0% (0.16) |
| PPAF | 81.0% (0.32) | **84.0% (0.64)** | 79.0% (1.28) |
| PPAF | 71.2% (2.56) | | |

Table 8: Different algorithms against UCT at MISERE KNIGHTTHROUGH $8 \times 8$ with 10,000 playouts.

| | | | |
|---|---|---|---|
| MAST | 56.0% (0.5) | 61.0% (1.0) | 77.6% (2.0) |
| MAST | 56.6% (4.0) | 59.2% (8.0) | 88.4% (16.0) |
| MAST | 98.0% (32.0) | 98.4% (64.0) | **100.0% (128.0)** |
| MAST | 99.8% (256.0) | | |
| | | | |
| MASTF | 57.0% (0.5) | 61.6% (1.0) | 72.0% (2.0) |
| MASTF | 62.0% (4.0) | 68.8% (8.0) | 96.0% (16.0) |
| MASTF | 99.8% (32.0) | **100.0% (64.0)** | 100.0% (128.0) |
| MASTF | 100.0% (256.0) | | |
| | | | |
| PPA | 63.8% (0.005) | 70.4% (0.01) | 90.6% (0.02) |
| PPA | 93.0% (0.04) | 95.6% (0.08) | 97.2% (0.16) |
| PPA | 98.8% (0.32) | 99.8% (0.64) | **100.0% (1.28)** |
| PPA | 100.0% (2.56) | | |
| | | | |
| PPAF | 57.4% (0.005) | 67.6% (0.01) | 79.4% (0.02) |
| PPAF | 88.8% (0.04) | 93.8% (0.08) | 98.4% (0.16) |
| PPAF | **100.0% (0.32)** | 100.0% (0.64) | 100.0% (1.28) |
| PPAF | 100.0% (2.56) | | |

Table 9: Different algorithms against PPAF at Misere Knightthrough $8 \times 8$ with 10,000 playouts.

| | | | |
|---|---|---|---|
| MAST | 0.0% (0.5) | 0.0% (1.0) | 0.0% (2.0) |
| MAST | 0.0% (4.0) | 0.2% (8.0) | 10.4% (16.0) |
| MAST | 24.6% (32.0) | 28.2% (64.0) | 25.6% (128.0) |
| MAST | 27.0% (256.0) | 25.2% (512.0) | **37.0% (1024.0)** |
| MAST | 3.4% (2048.0) | 1.8% (4096.0) | |
| | | | |
| MASTF | 0.0% (0.5) | 0.0% (1.0) | 0.0% (2.0) |
| MASTF | 0.0% (4.0) | 0.0% (8.0) | 19.4% (16.0) |
| MASTF | 31.0% (32.0) | 31.8% (64.0) | 30.2% (128.0) |
| MASTF | 26.6% (256.0) | 29.2% (512.0) | **34.4% (1024.0)** |
| MASTF | 9.4% (2048.0) | 5.0% (4096.0) | |

As adaptive algorithms win 100% at Misere Knightthrough, we ran another experiment to better distinguish between the different algorithms. We played MAST and MASTF against PPAF. The results are given in table 9. PPAF is much better than MAST and MASTF at Misere Knightthrough.

### *3.8.* Nogo $8 \times 8$

Table 10 gives the results of the different algorithms against standard UCT. We can see that PPA gives better results than MAST and that PPA with move features is a large improvement over other algorithms.

## 4. Conclusion

In the context of GGP we presented PPA and PPAF, algorithms that learn a playout policy online. They were tested on eight different games. For all games they are better than UCT. They are particularly good at misere games, scoring as high as 100% against UCT at Misere Knightthrough $8 \times 8$ with 10,000 playouts.

We have also presented an improvement on PPA and MAST. It modifies the way moves are coded so as to take into account features associated to the moves. The improved algorithms are called PPAF and MASTF. Tuning the $k$ and the $\alpha$ constants is important for MASTF and PPAF. The two algorithms are much better than UCT for all the games we tried.

Table 10: Different algorithms against UCT at NOGO $8 \times 8$ with 10,000 playouts.

| | | | |
|---|---|---|---|
| MAST | 52.8% (0.5) | 56.0% (1.0) | 58.2% (2.0) |
| MAST | 61.6% (4.0) | **62.4% (8.0)** | 58.6% (16.0) |
| MAST | 31.0% (32.0) | | |
| | | | |
| MASTF | 52.4% (0.5) | 58.2% (1.0) | **58.4% (2.0)** |
| MASTF | 50.0% (4.0) | 36.0% (8.0) | 19.4% (16.0) |
| MASTF | 11.2% (32.0) | | |
| | | | |
| PPA | 63.8% (0.005) | 73.0% (0.01) | **77.6% (0.02)** |
| PPA | 75.8% (0.04) | 76.2% (0.08) | 77.4% (0.16) |
| PPA | 66.2% (0.32) | 55.6% (0.64) | 35.2% (1.28) |
| PPA | 27.8% (2.56) | | |
| | | | |
| PPAF | 60.2% (0.005) | 74.4% (0.01) | 87.8% (0.02) |
| PPAF | 92.2% (0.04) | 95.2% (0.08) | **95.4% (0.16)** |
| PPAF | 94.4% (0.32) | 88.2% (0.64) | 79.0% (1.28) |
| PPAF | 69.2% (2.56) | | |

In BREAKTHROUGH, KNIGHTTHROUGH, NOGO, DOMINEERING, and MISERE DOMINEERING, PPAF is an improvement over PPA.

In BREAKTHROUGH, MISERE BREAKTHROUGH, KNIGHTTHROUGH, MISERE KNIGHTTHROUGH, NOGO, and DOMINEERING, PPAF is an improvement over MAST and MASTF.

PPA is tightly connected to the NRPA algorithm for single-player games. The main differences with NRPA are that it does not use nested levels nor a best sequence to learn. Instead it learns the moves of each playout for the winner of the playout.

Future work include combining PPA with the numerous enhancements of UCT. Some of them may be redundant but others will probably be cumulative. For example combining PPA with GRAVE could yield substantial benefits in some games.

[1] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of

Monte Carlo tree search methods, IEEE Transactions on Computational Intelligence and AI in Games 4 (1) (2012) 1–43.

[2] L. Kocsis, C. Szepesvári, Bandit based Monte-Carlo planning, in: 17th European Conference on Machine Learning (ECML'06), Vol. 4212 of LNCS, Springer, 2006, pp. 282–293.

[3] R. Coulom, Efficient selectivity and backup operators in Monte-Carlo tree search, in: H. J. van den Herik, P. Ciancarini, H. H. L. M. Donkers (Eds.), Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers, Vol. 4630 of Lecture Notes in Computer Science, Springer, 2006, pp. 72–83.

[4] S. Huang, B. Arneson, R. B. Hayward, M. Müller, J. Pawlewicz, Mohex 2.0: A pattern-based MCTS Hex player, in: Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013, Revised Selected Papers, 2013, pp. 60–71.

[5] H. Finnsson, Y. Björnsson, Simulation-based approach to general game playing, in: AAAI, 2008, pp. 259–264.

[6] J. Méhat, T. Cazenave, A parallel general game player, KI 25 (1) (2011) 43–47.

[7] J. Pitrat, Realization of a general game-playing program, in: IFIP Congress (2), 1968, pp. 1570–1574.

[8] M. R. Genesereth, N. Love, B. Pell, General game playing: Overview of the AAAI competition, AI Magazine 26 (2) (2005) 62–72.
URL `http://www.aaai.org/ojs/index.php/aimagazine/article/view/1813`

[9] R. Coulom, Computing elo ratings of move patterns in the game of Go, ICGA Journal 30 (4) (2007) 198–208.

[10] S. Huang, R. Coulom, S. Lin, Monte-Carlo simulation balancing in practice, in: H. J. van den Herik, H. Iida, A. Plaat (Eds.), Computers and Games - 7th International Conference, CG 2010, Kanazawa, Japan, September 24-26, 2010, Revised Selected Papers, Vol. 6515 of Lecture Notes in Computer Science, Springer, 2010, pp. 81–92.

[11] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis, Mastering the game of go with deep neural networks and tree search, Nature 529 (7587) (2016) 484–489.

[12] S. Gelly, D. Silver, Monte-Carlo tree search and rapid action value estimation in computer Go, Artif. Intell. 175 (11) (2011) 1856–1875.

[13] A. Rimmel, F. Teytaud, O. Teytaud, Biasing Monte-Carlo simulations through RAVE values, in: H. J. van den Herik, H. Iida, A. Plaat (Eds.), Computers and Games - 7th International Conference, CG 2010, Kanazawa, Japan, September 24-26, 2010, Revised Selected Papers, Vol. 6515 of Lecture Notes in Computer Science, Springer, 2010, pp. 59–68.

[14] T. Cazenave, Generalized rapid action value estimation, in: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015, 2015, pp. 754–760.

[15] H. Finnsson, Y. Björnsson, Learning simulation control in general game-playing agents, in: AAAI, 2010.
URL `http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1892`

[16] H. Finnsson, Y. Björnsson, Cadiaplayer: Search-control techniques, KI-Künstliche Intelligenz 25 (1) (2011) 9–16.

[17] T. Cazenave, Playout policy adaptation for games, in: Advances in Computer Games - 14th International Conference, ACG 2015, Leiden, The Netherlands, July 1-3, 2015, Revised Selected Papers, 2015, pp. 20–28.

[18] T. Graf, M. Platzner, Adaptive playouts in monte-carlo tree search with policy-gradient reinforcement learning, in: Advances in Computer Games - 14th International Conference, ACG 2015, Leiden, The Netherlands, July 1-3, 2015, Revised Selected Papers, 2015, pp. 1–11.

[19] M. J. W. Tak, M. H. M. Winands, Y. Björnsson, N-grams and the last-good-reply policy applied in general game playing, IEEE Transactions on Computational Intelligence and AI in Games 4 (2) (2012) 73–83.

[20] M. Swiechowski, J. Mandziuk, Self-adaptation of playing strategies in general game playing, IEEE Transactions on Computational Intelligence and AI in Games 6 (4) (2014) 367–381.

[21] T. Cazenave, Nested Monte-Carlo Search, in: C. Boutilier (Ed.), IJCAI, 2009, pp. 456–461.

[22] A. Rimmel, F. Teytaud, T. Cazenave, Optimization of the Nested Monte-Carlo algorithm on the traveling salesman problem with time windows, in: Applications of Evolutionary Computation - EvoApplications 2011: EvoCOMNET, EvoFIN, EvoHOT, EvoMUSART, EvoSTIM, and EvoTRANSLOG, Torino, Italy, April 27-29, 2011, Proceedings, Part II, Vol. 6625 of Lecture Notes in Computer Science, Springer, 2011, pp. 501–510.

[23] C. D. Rosin, Nested rollout policy adaptation for Monte Carlo tree search, in: IJCAI, 2011, pp. 649–654.

[24] T. Cazenave, F. Teytaud, Application of the nested rollout policy adaptation algorithm to the traveling salesman problem with time windows, in: Learning and Intelligent Optimization - 6th International Conference, LION 6, Paris, France, January 16-20, 2012, Revised Selected Papers, 2012, pp. 42–54.

[25] S. Edelkamp, M. Gath, T. Cazenave, F. Teytaud, Algorithm and knowledge engineering for the tsptw problem, in: Computational Intelligence in Scheduling (SCIS), 2013 IEEE Symposium on, IEEE, 2013, pp. 44–51.

[26] S. Edelkamp, M. Gath, M. Rohde, Monte-carlo tree search for 3d packing with object orientation, in: KI 2014: Advances in Artificial Intelligence, Springer International Publishing, 2014, pp. 285–296.

[27] S. Edelkamp, C. Greulich, Solving physical traveling salesman problems with policy adaptation, in: Computational Intelligence and Games (CIG), 2014 IEEE Conference on, IEEE, 2014, pp. 1–8.

[28] S. Edelkamp, Z. Tang, Monte-carlo tree search for the multiple sequence alignment problem, in: Eighth Annual Symposium on Combinatorial Search, 2015.

[29] S. Edelkamp, M. Gath, C. Greulich, M. Humann, O. Herzog, M. Lawo, Monte-carlo tree search for logistics, in: Commercial Transport, Springer International Publishing, 2016, pp. 427–440.

[30] T. Cazenave, A. Saffidine, M. J. Schofield, M. Thielscher, Nested monte carlo search for two-player games, in: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA., 2016, pp. 687–693.

[31] D. Perez, S. Samothrakis, S. Lucas, Knowledge-based fast evolutionary mcts for general video game playing, in: Computational Intelligence and Games (CIG), 2014 IEEE Conference on, IEEE, 2014, pp. 1–8.

[32] M. Trutman, S. Schiffel, Creating action heuristics for general game playing agents, in: Computer Games - Fourth Workshop on Computer Games, CGW 2015, and the Fourth Workshop on General Intelligence in Game-Playing Agents, GIGA 2015, Held in Conjunction with the 24th International Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 26-27, 2015, Revised Selected Papers, 2015, pp. 149–164.

[33] F. Boissac, T. Cazenave, De nouvelles heuristiques de recherche appliquées à la résolution d'Atarigo, in: Intelligence artificielle et jeux, Hermes Science, 2006, pp. 127–141.

[34] A. Saffidine, N. Jouandeau, T. Cazenave, Solving Breakthrough with race patterns and job-level proof number search, in: ACG, 2011, pp. 196–207.

[35] R. Lorentz, T. Horey, Programming Breakthrough, in: H. J. van den Herik, H. Iida, A. Plaat (Eds.), Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013, Revised Selected Papers, Vol. 8427 of Lecture Notes in Computer Science, Springer, 2013, pp. 49–59.

[36] M. Gardner, Mathematical games, Scientific American 230 (1974) 106–108.

[37] J. W. H. M. Uiterwijk, Perfectly solving Domineering boards, in: T. Cazenave, M. H. M. Winands, H. Iida (Eds.), Computer Games -

Workshop on Computer Games, CGW 2013, Held in Conjunction with the 23rd International Conference on Artificial Intelligence, IJCAI 2013, Beijing, China, August 3, 2013, Revised Selected Papers, Vol. 408 of Communications in Computer and Information Science, Springer, 2013, pp. 97–121.

[38] M. Enzenberger, M. Muller, B. Arneson, R. Segal, Fuego - an open-source framework for board games and Go engine based on Monte Carlo tree search, IEEE Transactions on Computational Intelligence and AI in Games 2 (4) (2010) 259–270.

[39] C.-W. Chou, O. Teytaud, S.-J. Yen, Revisiting Monte-Carlo tree search on a normal form game: NoGo, in: Applications of Evolutionary Computation, Vol. 6624 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 73–82.

[40] T. Cazenave, Sequential halving applied to trees, IEEE Transactions on Computational Intelligence and AI in Games 7 (1) (2015) 102–105.