Playout Policy Adaptation for Games

Tristan Cazenave(✉)

LAMSADE, Université Paris-Dauphine, Paris, France
cazenave@lamsade.dauphine.fr

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 test the resulting algorithm named Playout Policy Adaptation (PPA) on Atarigo, Breakthrough, Misere Breakthrough, Domineering, Misere Dominee-ring, Go, Knightthrough, Misere Knightthrough, Nogo and Misere Nogo. For most of these games, PPA is better than UCT with a uniform random playout policy, with the notable exceptions of Go and Nogo.

1 Introduction

Monte-Carlo Tree Search (MCTS) has been successfully applied to many games and problems [2]. The most popular MCTS algorithm is Upper Confidence bounds for Trees (UCT) [17]. MCTS is particularly successful in the game of Go [7]. It is also the state of the art in Hex [15] and General Game Playing (GGP) [10,20]. GGP can be traced back to the seminal work of Jacques Pitrat [21]. Since 2005 an annual GGP competition is organized by Stanford at AAAI [14]. Since 2007 all the winners of the competition use MCTS.

Offline learning of playout policies has given good results in Go [8,16] and Hex [15], learning fixed pattern weights so as to bias the playouts.

The RAVE algorithm [13] 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 [23]. Pool RAVE improves slightly on random playouts in Havannah and reaches 62.7 % against random playouts in Go.

Move-Average Sampling Technique (MAST) is a technique used in the GGP program Cadia Player so as to bias the playouts with statistics on moves [10,11]. 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. Playout Policy Adaptation (PPA) 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.

A. Plaat et al. (Eds.): ACG 2015, LNCS 9525, pp. 20–28, 2015.
DOI: 10.1007/978-3-319-27992-3_3

Later improvements of Cadia Player are N-Grams and the last good reply policy [27]. 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 [26].

A related domain is the learning of playout policies in single-player problems. Nested Monte-Carlo Search (NMCS) [3] 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) [24]. NRPA has found new world records in Morpion Solitaire and crosswords puzzles. The principle 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 [5]. PPA is a follow-up to this paper since it is the adaptation of NRPA to two-player games.

In the context of GGP we look for general enhancements of UCT that work for many games without game-specific tweakings. This is the case for PPA. In our experiments we use the exact same algorithm for all the games. It is usually more difficult to find a general enhancement than a game specific one. PPA is an online learning algorithm, it starts from scratch for every position and learns a position specific playout policy each time.

We now give the outline of the paper. The next section details the PPA algorithm and particularly the playout strategy and the adaptation of the policy. The third section gives experimental results for various games, various board sizes and various numbers of playouts. 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 PPA 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 principle is the same as the adaptation of NRPA except that it only adapts the policy of the winner of the playout with the moves of the winner.

The PPA-playout algorithm is given in Algorithm 1. It 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 the exponential of its policy number and the move to play is chosen with a probability proportional to this value.

The PPA-adaptation algorithm is given in Algorithm 2. It 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 is parameterized by α 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 and decreases the weight of the other possible moves on the current board.

The PPA algorithm is given in Algorithm 3. It 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 PPA-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 PPA playout that gives the winner of this single playout. The playout and its winner are then used to adapt the policy using the PPA-adapt function. When all playouts have been played the PPA function returns the move that has the most playouts at the root as in usual UCT.

The UCT algorithm called by the PPA algorithm is given in Algorithm 4.

3 Experimental Results

We played PPA against UCT with random playouts. Both algorithms use the same number of playouts. The UCT constant is set to 0.4 for both algorithms as is usual in GGP. α is set to 1.0 for PPA. For each game we test two board sizes: 5×5 and 8×8, and two numbers of playouts: 1,000 and 10,000.

The games we have experimented with are:

- Atarigo: the rules are the same as for Go except that the first player to capture a string has won. Atarigo has been solved up to size 6×6 [1].
- 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

Algorithm 1. The PPA-playout algorithm

```
playout (board, player, policy)
while true do
  if board is terminal then
    return winner (board)
  end if
  z ← 0.0
  for m in possible moves on board do
    z ← z + exp (policy [m])
  end for
  choose a move for player with probability proportional to exp(policy[move]) / z
  play (board, move)
  player ← opponent (player)
end while
```

Algorithm 2. The PPA-adaptation algorithm

```
adapt (winner, board, player, playout, policy)
polp ← policy
for move in playout do
  if winner = player then
    polp [move] ← polp [move] + α
    z ← 0.0
    for m in possible moves on board do
      z ← z + exp (policy [m])
    end for
    for m in possible moves on board do
      polp [m] ← polp [m] - α * exp(policy[m]) / z
    end for
  end if
  play (board, move)
  player ← opponent (player)
end for
policy ← polp
```

Algorithm 3. The PPA algorithm

```
PPA (board, player)
for i in 0, maximum index of a move do
  policy[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
```

Algorithm 4. 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
```

has been solved up to size 6×5 using Job Level Proof Number Search [25]. The best program for Breakthrough 8×8 uses MCTS combined with an evaluation function after a short playout [19].

- Misere Breakthrough: The rules are the same as for Breakthrough except that the first player to reach the opposite row has lost.
- 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 [12]. Jos Uiterwijk recently proposed a knowledge based method that can solve large rectangular boards without any search [28].
- Misere Domineering: The rules are the same as for Domineering except that the first player that cannot play has won.
- Go: The game starts with an empty grid. Players alternatively place black and white stones on the intersections. A completely surrounded string of stones is removed from the board. The score of a player at the end of a game with

Table 1. Win rate against UCT with the same number of playouts as PPA for various games of various board sizes using either 1,000 or 10,000 playouts per move.

	Size	Playouts	
		1,000	10,000
Atarigo	5×5	81.2	90.6
Atarigo	8×8	72.2	94.4
Breakthrough	5×5	60.0	56.2
Breakthrough	8×8	55.2	54.4
Misere Breakthrough	5×5	95.0	99.6
Misere Breakthrough	8×8	99.2	97.8
Domineering	5×5	62.6	50.0
Domineering	8×8	48.4	58.0
Misere Domineering	5×5	63.4	62.2
Misere Domineering	8×8	76.4	83.4
Go	5×5	21.2	23.6
Go	8×8	23.0	1.2
Knightthrough	5×5	42.4	30.2
Knightthrough	8×8	64.2	64.6
Misere Knightthrough	5×5	95.8	99.8
Misere Knightthrough	8×8	99.8	100.0
Nogo	5×5	61.8	71.0
Nogo	8×8	64.8	46.4
Misere Nogo	5×5	66.4	67.8
Misere Nogo	8×8	80.6	89.4

chinese rules is the number of her[1] stones on the board plus the number of her eyes. The player with the greatest score has won. We use a komi of 7.5 for white. Go was the first tremendously successful application of MCTS to games [7–9,18]. All the best current Go programs use MCTS.

- Knightthrough: The rules are similar to Breakthrough except that the pawns are replaced by knights that can only go forward.
- Misere Knightthrough: The rules are the same as for Knightthrough except that the first player to reach the opposite row has lost.
- Nogo: The rules are the same as Go except that it is forbidden to capture and to commit suicide. The first player that cannot move has lost. There exist computer Nogo competitions and the best players use MCTS [4,6,9].
- Misere Nogo: The rules are the same as for Nogo except that first player that cannot move has won.

[1] For brevity, we use 'he' and 'his', whenever 'he or she' and 'his or her' are meant.

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 and perform poorly when applied directly to two-player games. We give results of 1,000 and 10,000 playouts per move.

Results are given in Table 1. Each result is the outcome of a 500 games match, 250 playing first and 250 playing second.

PPA has worse results than UCT in three games: Go, Knightthrough 5×5 and Nogo 8×8. For the other 17 games it improves over UCT. It is particularly good at misere games, a possible explanation is that it learns to avoid losing moves in the playouts and that it may be important for misere games that are waiting games.

We observe that PPA scales well in Atarigo, Misere Breakthrough, Misere Domineering, Knightthrough, Misere Knightthrough and Misere Nogo: it is equally good or even better when the size of the board or the number of playouts is increased. On the contrary it does not scale for Go and Nogo.

A possible explanation of the bad behaviour in Go could be that moves in Go can be either good or bad depending on the context and that learning an overall evaluation of a move can be misleading.

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 unexpensive compared to the time used for the interpretation of the rules of the game.

4 Conclusion

In the context of GGP we presented PPA, an algorithm that learns a playout policy online. It was tested on ten different games for increasing board sizes and increasing numbers of playouts. On many games it scales well with board size and number of playouts and it is better than UCT for 33 out of the 40 experiments we performed. It is particularly good at misere games, scoring as high as 100 % against UCT at Misere Knightthrough 8×8 with 10,000 playouts.

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 RAVE could yield substantial benefits in some games.

A second line of research is understanding why PPA is good at many games and bad at other games such as Go. It would be interesting being able to tell the features of a game that make PPA useful.

References

1. Boissac, F., Cazenave, T.: De nouvelles heuristiques de recherche appliquées à la résolution d'Atarigo. In: Intelligence artificielle et jeux, pp. 127–141. Hermes Science (2006)
2. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte Carlo tree search methods. IEEE Trans. Comput. Intell. AI Games **4**(1), 1–43 (2012)
3. Cazenave, T.: Nested Monte-Carlo search. In: Boutilier, C. (ed.) IJCAI, pp. 456–461 (2009)
4. Cazenave, T.: Sequential halving applied to trees. IEEE Trans. Comput. Intell. AI Games **7**(1), 102–105 (2015)
5. Cazenave, T., Saffidine, A., Schofield, M., Thielscher, M.: Discounting and pruning for nested playouts in general game playing. GIGA at IJCAI (2015)
6. Chou, C.-W., Teytaud, O., Yen, S.-J.: Revisiting Monte-Carlo tree search on a normal form game: NoGo. In: Di Chio, C., et al. (eds.) EvoApplications 2011, Part I. LNCS, vol. 6624, pp. 73–82. Springer, Heidelberg (2011)
7. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M.J. (eds.) CG 2006. LNCS, vol. 4630, pp. 72–83. Springer, Heidelberg (2007)
8. Coulom, R.: Computing elo ratings of move patterns in the game of go. ICGA J. **30**(4), 198–208 (2007)
9. Enzenberger, M., Muller, M., Arneson, B., Segal, R.: Fuego - an open-source framework for board games and go engine based on Monte Carlo tree search. IEEE Trans. Comput. Intell. AI Games **2**(4), 259–270 (2010)
10. Finnsson, H., Björnsson, Y.: Simulation-based approach to general game playing. In: AAAI, pp. 259–264 (2008)
11. Finnsson, H., Björnsson, Y.: Learning simulation control in general game-playing agents. In: AAAI (2010)
12. Gardner, M.: Mathematical games. Sci. Am. **230**, 106–108 (1974)
13. Gelly, S., Silver, D.: Monte-Carlo tree search and rapid action value estimation in computer go. Artif. Intell. **175**(11), 1856–1875 (2011)
14. Genesereth, M.R., Love, N., Pell, B.: General game playing: overview of the AAAI competition. AI Mag. **26**(2), 62–72 (2005)
15. Huang, S., Arneson, B., Hayward, R.B., Müller, M., Pawlewicz, J.: Mohex 2.0: a pattern-based MCTS hex player. In: Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, 13–15 August 2013, Revised Selected Papers, pp. 60–71 (2013)
16. Huang, S.-C., Coulom, R., Lin, S.-S.: Monte-Carlo simulation balancing in practice. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 81–92. Springer, Heidelberg (2011)
17. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
18. Lee, C., Wang, M., Chaslot, G., Hoock, J., Rimmel, A., Teytaud, O., Tsai, S., Hsu, S., Hong, T.: The computational intelligence of MoGo revealed in taiwan's computer go tournaments. IEEE Trans. Comput. Intell. AI Games **1**(1), 73–89 (2009)
19. Lorentz, R., Horey, T.: Programming breakthrough. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2013. LNCS, vol. 8427, pp. 49–59. Springer, Heidelberg (2014)

20. Méhat, J., Cazenave, T.: A parallel general game player. KI **25**(1), 43–47 (2011)
21. Pitrat, J.: Realization of a general game-playing program. IFIP Congr. **2**, 1570–1574 (1968)
22. Rimmel, A., Teytaud, F., Cazenave, T.: Optimization of the nested Monte-Carlo algorithm on the traveling salesman problem with time windows. In: Di Chio, C., et al. (eds.) EvoApplications 2011, Part II. LNCS, vol. 6625, pp. 501–510. Springer, Heidelberg (2011)
23. Rimmel, A., Teytaud, F., Teytaud, O.: Biasing Monte-Carlo simulations through RAVE values. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 59–68. Springer, Heidelberg (2011)
24. Rosin, C.D.: Nested rollout policy adaptation for Monte Carlo tree search. In: IJCAI, pp. 649–654 (2011)
25. Saffidine, A., Jouandeau, N., Cazenave, T.: Solving BREAKTHROUGH with race patterns and job-level proof number search. In: van den Herik, H.J., Plaat, A. (eds.) ACG 2011. LNCS, vol. 7168, pp. 196–207. Springer, Heidelberg (2012)
26. Swiechowski, M., Mandziuk, J.: Self-adaptation of playing strategies in general game playing. IEEE Trans. Comput. Intell. AI Games **6**(4), 367–381 (2014)
27. Tak, M.J.W., Winands, M.H.M., Björnsson, Y.: N-grams and the last-good-reply policy applied in general game playing. IEEE Trans. Comput. Intell. AI Games **4**(2), 73–83 (2012)
28. Uiterwijk, J.W.H.M.: Perfectly solving domineering boards. In: Cazenave, T., Winands, M.H.M., Lida, H. (eds.) CGW 2013. Communications in Computer and Information Science, vol. 408, pp. 97–121. Springer, Switzerland (2013)