Memorizing the Playout Policy

Tristan Cazenave[1] and Eustache Diemert[2]

[1] Université Paris-Dauphine, PSL Research University, CNRS, LAMSADE, PARIS, FRANCE
[2] CRITEO, GRENOBLE, FRANCE

Abstract. Monte Carlo Tree Search (MCTS) is the state of the art algorithm for General Game Playing (GGP). Playout Policy Adaptation with move Features (PPAF) is a state of the art MCTS algorithm that learns a playout policy online. We propose a simple modification to PPAF consisting in memorizing the learned policy from one move to the next. We test PPAF with memorization (PPAFM) against PPAF and UCT for Atarigo, Breakthrough, Misere Breakthrough, Domineering, Misere Domineering, Knightthrough, Misere Knightthrough and Nogo.

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) [27]. MCTS is particularly successful in the game of Go [9]. It is also the state of the art in Hex [25] and General Game Playing (GGP) [17, 29]. GGP can be traced back to the seminal work of Jacques Pitrat [31]. Since 2005 an annual GGP competition is organized by Stanford at AAAI [22]. Since 2007 all the winners of the competition use MCTS.

Offline learning of playout policies has given good results in Go [10, 26] and Hex [25], learning fixed pattern weights so as to bias the playouts. AlphaGo [36] 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 [21] 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 [33]. Pool RAVE improves slightly on random playouts in Havannah and reaches 62.7% against random playouts in Go.

The GRAVE algorithm [3] 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 [17, 18]. 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(λ) [19]. FAST is used to bias playouts in combination with MAST but only slightly improves on MAST.

Later improvements of CadiaPlayer are N-Grams and the last good reply policy [38]. 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 [37].

A related domain is the learning of playout policies in single-player problems. Nested Monte Carlo Search (NMCS) [2] 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 [32].

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) [34]. 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 [7, 11]. Other applications deal with 3D Packing with Object Orientation [13], the physical traveling salesman problem [14], the Multiple Sequence Alignment problem [15] or Logistics [12]. 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.

Playout Policy Adaptation (PPA) [4] 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.

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. 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 [23, 24].

The use of features to improve MCTS playouts has also been proposed in the General Game AI settings [30]. The approach is different from PPAF since features are part of the state and are used to evaluate states. Instead PPAF 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 [39].

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 explains the PPAF algorithm with memorization of the policy. The fourth section gives experimental results for various games. The last section concludes.

2 Playout Policy Adaptation with Move Features

PPAF [6] 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. The playout algorithm for PPAF is given in algorithm 1.

For each game state where it has to find a move to play, PPAF 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 weights of the moves of the winner of the playout are increased by a constant α and the weight of the other moves of the same state are decreased by a value proportional to the exponential of their weight. The Adapt algorithm is given in algorithm 2. The Adapt algorithm replays the playout and for the states where the winner has played it modifies the weights of the possible moves, increasing the played move weight and decreasing the possible moves weights proportionally to their probability of being played.

Move features are enriched information about the moves. A move is represented in PPAF by a code. When not using features the code is calculated using the location of the move on the board. When using features both the location of the move and properties of the move are coded. An example of a property is whether a move is a capture or not. Another example is to code the colors of the intersections adjacent to the move.

The PPAF algorithm is given in algorithm 3. The policy is initialized at first with a uniform policy, then for each playout PPAF adapts the policy for the winner of the playout.

In order to be complete, the UCT algorithm is given in algorithm 4. When UCT uses a uniform playout policy it is named UCT in the following. When it is called by the PPAF algorithm, the same code is used as in UCT for the descent of the tree but the playouts use a non uniform policy in algorithm 1.

3 PPAF with Memorization of the Playout Policy

The principle of PPAFM is to initialize the playout policy before each move with an already trained policy instead of initializing it with an uniform policy. In the first two moves of the game, the policy can be initialized with a game specific policy. In order to test the efficiency of game specific initial policies we will test PPAFM both with an initial uniform policy and with an initial game specific policy.

For moves after the first two moves, PPAFM initializes its policy with the policy learned during the previous call to PPAFM for the state two moves before. It is better

Algorithm 1 The 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 (k × policy [code(m)])
    end for
    choose a move for player with probability proportional to exp(k×policy[code(move)]) / z
    play (board, move)
    player ← opponent (player)
end while
```

Algorithm 2 The 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)] + α
        z ← 0.0
        for m in possible moves on board do
            z ← z + exp (policy [code(m)])
        end for
        for m in possible moves on board do
            polp [code(m)] ← polp [code(m)] - α * exp(policy[code(m)]) / z
        end for
    end if
    play (board, move)
    player ← opponent (player)
end for
policy ← polp
```

Algorithm 3 The PPAF algorithm

```
PPAF (board, player)
for i in 0, maximum index of a move code 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
```

than using a policy learned for any game state since the state of the previous call is much closer to the current state than another state. A policy learned for any state is less relevant than the last state policy since it does not capture state specific knowledge.

The PPAFM algorithm is given in algorithm 5. The descent of the tree is the same as in UCT and the adapt function is the same as in PPAF. The playout algorithm is also the same as in PPAF and is different from UCT. PPAFM uses Gibbs sampling and UCT uses uniform playouts. The main difference with PPAF is the initialization of the playout policy. The first test in the PPAFM algorithm enables to start a game with a policy already learned on the initial state, is can also be a uniform policy. If the move is not the first move of a game then we enter the code following the else and the playout policy is initialized with the memorized policy. At the end of the algorithm the policy learned for the board is memorized.

A nice property of PPAF is that the move played after the algorithm has been run is the most simulated move, this is also the case for UCT. In the case of PPAFM it means that the memorized policy is related to the state after the move played by the algorithm since it is the most simulated move. So when starting with the memorized policy for the next state, this state has already been partially learned.

Algorithm 5 The PPAFM algorithm.

```
PPAFM (board, player)
if board has less than two moves then
    for i in 0, maximum index of a move code do
        policy[i] ← initialPolicy[i]
    end for
else
    for i in 0, maximum index of a move code do
        policy[i] ← memorizedPolicy[i]
    end for
end if
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
for i in 0, maximum index of a move code do
    memorizedPolicy[i] ← policy[i]
end for
return  the move with the most playouts
```

4 Experimental Results

PPAFM was tested against PPAF without memorization and also against UCT. As the best overall performing α constant for PPAF against UCT among the tested games is

0.32, we use this constant both for PPAF and for PPAFM. Each result is the winning percentage of PPAF with memorization in a 500 games match, 250 with Black and 250 with White. In order to decide which move to play, all algorithms use 10 000 playouts.

4.1 Games

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. 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 6×5 using Job Level Proof Number Search [35]. The best program for Breakthrough 8×8 uses MCTS combined with an evaluation function after a short playout [28]. 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.
- 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 [20]. Jos Uiterwijk recently proposed a knowledge based method that can solve large rectangular boards without any search [40]. 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 [16, 8, 5]. We use the same move feature as for Atarigo.

For all the games we use standard 8×8 boards in the experiments.

4.2 Memorizing the policy from one move to the next starting a game with a uniform policy

In the following experiments we use an initial uniform policy for PPAFM. Table 1 gives the results for PPAFM against PPAF. Table 2 gives the results for PPAFM against UCT with an uniform playout policy. It is clear from the first table that PPAFM is stronger than PPAF except for Nogo where it is of equal strength. It is particularly good at Misere Breakthrough and Misere Knightthrough where it scores an almost perfect score. We find the same phenomenon as when playing PPAF against UCT. In these misere games avoiding bad moves in playouts is extremely important and PPAFM is much better than PPAF at learning move weights.

Table 2 shows that PPAFM is much stronger than UCT for all tested games.

Table 1. PPAFM with an initial uniform policy versus PPAF for different games.

Game	Score
Atarigo	66.0%
Breakthrough	87.4%
Domineering	58.0%
Knightthrough	84.6%
Misere Breakthrough	97.2%
Misere Domineering	56.8%
Misere Knightthrough	99.2%
Nogo	49.4%

Table 2. PPAFM with an initial uniform policy versus UCT for different games.

Game	Score
Atarigo	95.4%
Breakthrough	94.2%
Domineering	81.8%
Knightthrough	96.6%
Misere Breakthrough	100.0%
Misere Domineering	95.8%
Misere Knightthrough	100.0%
Nogo	91.6%

4.3 Starting with an initial learned policy

For each game an initial policy was computed using 100 000 playouts on each of the possible states with less than two moves. The UCT tree was forgotten and only the learned policy was memorized for each state. The learned policy is used to initialize the PPAFM policy for the first call to PPAFM in a game. Table 3 gives the winning percentage of PPAFM with an initial policy against PPAF. Table 4 gives the results for PPAFM with an initial policy against UCT. According to these two tables, using an initial learned policy is beneficial at Atarigo and Domineering. It is worse at Nogo and it is equal for the other games.

Table 3. PPAFM with an initial learned policy versus PPAF for different games.

Game	Score
Atarigo	79.2%
Breakthrough	86.4%
Domineering	67.0%
Knightthrough	86.6%
Misere Breakthrough	97.6%
Misere Domineering	56.2%
Misere Knightthrough	99.0%
Nogo	43.0%

Table 4. PPAFM with an initial learned policy versus UCT for different games.

Game	Score
Atarigo	97.2%
Breakthrough	93.0%
Domineering	86.4%
Knightthrough	97.2%
Misere Breakthrough	100.0%
Misere Domineering	94.8%
Misere Knightthrough	100.0%
Nogo	91.4%

5 Conclusion

PPAF is an algorithm that learns a playout policy using move features. It is much better than UCT for all the tested games. We propose a simple improvement to PPAF which is to memorize the learned playout policy from one move to the next. Experimental results show that it is a large improvement over PPAF. It is also a large improvement against UCT.

In future work we plan to improve move features, possibly learning them and to improve the policy learning algorithm.

References

1. C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, Mar. 2012.
2. T. Cazenave. Nested Monte-Carlo Search. In C. Boutilier, editor, *IJCAI*, pages 456–461, 2009.
3. 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*, pages 754–760, 2015.
4. 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*, pages 20–28, 2015.
5. T. Cazenave. Sequential halving applied to trees. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(1):102–105, 2015.
6. T. Cazenave. Playout policy adaptation with move features. *Theor. Comput. Sci.*, 644:43–52, 2016.
7. T. Cazenave and 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*, pages 42–54, 2012.
8. C.-W. Chou, O. Teytaud, and S.-J. Yen. Revisiting Monte-Carlo tree search on a normal form game: NoGo. In *Applications of Evolutionary Computation*, volume 6624 of *Lecture Notes in Computer Science*, pages 73–82. Springer Berlin Heidelberg, 2011.
9. R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.
10. R. Coulom. Computing elo ratings of move patterns in the game of Go. *ICGA Journal*, 30(4):198–208, 2007.
11. S. Edelkamp, M. Gath, T. Cazenave, and F. Teytaud. Algorithm and knowledge engineering for the tsptw problem. In *Computational Intelligence in Scheduling (SCIS), 2013 IEEE Symposium on*, pages 44–51. IEEE, 2013.
12. S. Edelkamp, M. Gath, C. Greulich, M. Humann, O. Herzog, and M. Lawo. Monte-carlo tree search for logistics. In *Commercial Transport*, pages 427–440. Springer International Publishing, 2016.
13. S. Edelkamp, M. Gath, and M. Rohde. Monte-carlo tree search for 3d packing with object orientation. In *KI 2014: Advances in Artificial Intelligence*, pages 285–296. Springer International Publishing, 2014.
14. S. Edelkamp and C. Greulich. Solving physical traveling salesman problems with policy adaptation. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.
15. S. Edelkamp and Z. Tang. Monte-carlo tree search for the multiple sequence alignment problem. In *Eighth Annual Symposium on Combinatorial Search*, 2015.
16. M. Enzenberger, M. Muller, B. Arneson, and 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):259–270, 2010.
17. H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In *AAAI*, pages 259–264, 2008.

18. H. Finnsson and Y. Björnsson. Learning simulation control in general game-playing agents. In *AAAI*, 2010.
19. H. Finnsson and Y. Björnsson. Cadiaplayer: Search-control techniques. *KI-Künstliche Intelligenz*, 25(1):9–16, 2011.
20. M. Gardner. Mathematical games. *Scientific American*, 230:106–108, 1974.
21. S. Gelly and D. Silver. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artif. Intell.*, 175(11):1856–1875, 2011.
22. M. R. Genesereth, N. Love, and B. Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.
23. T. Graf and 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*, pages 1–11, 2015.
24. T. Graf and M. Platzner. Adaptive playouts for online learning of policies during monte carlo tree search. *Theor. Comput. Sci.*, 644:53–62, 2016.
25. S. Huang, B. Arneson, R. B. Hayward, M. Müller, and 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*, pages 60–71, 2013.
26. S. Huang, R. Coulom, and S. Lin. Monte-Carlo simulation balancing in practice. In H. J. van den Herik, H. Iida, and A. Plaat, editors, *Computers and Games - 7th International Conference, CG 2010, Kanazawa, Japan, September 24-26, 2010, Revised Selected Papers*, volume 6515 of *Lecture Notes in Computer Science*, pages 81–92. Springer, 2010.
27. L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *17th European Conference on Machine Learning (ECML'06)*, volume 4212 of *LNCS*, pages 282–293. Springer, 2006.
28. R. Lorentz and T. Horey. Programming Breakthrough. In H. J. van den Herik, H. Iida, and A. Plaat, editors, *Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013, Revised Selected Papers*, volume 8427 of *Lecture Notes in Computer Science*, pages 49–59. Springer, 2013.
29. J. Méhat and T. Cazenave. A parallel general game player. *KI*, 25(1):43–47, 2011.
30. D. Perez, S. Samothrakis, and S. Lucas. Knowledge-based fast evolutionary mcts for general video game playing. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.
31. J. Pitrat. Realization of a general game-playing program. In *IFIP Congress (2)*, pages 1570–1574, 1968.
32. A. Rimmel, F. Teytaud, and 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*, volume 6625 of *Lecture Notes in Computer Science*, pages 501–510. Springer, 2011.
33. A. Rimmel, F. Teytaud, and O. Teytaud. Biasing Monte-Carlo simulations through RAVE values. In H. J. van den Herik, H. Iida, and A. Plaat, editors, *Computers and Games - 7th International Conference, CG 2010, Kanazawa, Japan, September 24-26, 2010, Revised Selected Papers*, volume 6515 of *Lecture Notes in Computer Science*, pages 59–68. Springer, 2010.
34. C. D. Rosin. Nested rollout policy adaptation for Monte Carlo tree search. In *IJCAI*, pages 649–654, 2011.
35. A. Saffidine, N. Jouandeau, and T. Cazenave. Solving Breakthrough with race patterns and job-level proof number search. In *ACG*, pages 196–207, 2011.

36. 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, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.
37. M. Swiechowski and J. Mandziuk. Self-adaptation of playing strategies in general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):367–381, 2014.
38. M. J. W. Tak, M. H. M. Winands, and 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):73–83, 2012.
39. M. Trutman and 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*, pages 149–164, 2015.
40. J. W. H. M. Uiterwijk. Perfectly solving Domineering boards. In T. Cazenave, M. H. M. Winands, and H. Iida, editors, *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*, volume 408 of *Communications in Computer and Information Science*, pages 97–121. Springer, 2013.