Solving BREAKTHROUGH with Race Patterns and Job-Level Proof Number Search

Abdallah Saffidine[1], Nicolas Jouandeau[2], and Tristan Cazenave[1]

[1] LAMSADE, Université Paris-Dauphine
[2] LIASD, Université Paris 8

Abstract. BREAKTHROUGH is a recent race based board game usually played on size 8×8. We describe a method to solve 6×5 boards based on race patterns and an extension of Job-Level Proof Number Search (JLPNS).
Race patterns is a new domain specific technique that allows early endgame detection. The patterns we use enable positions as far as 7 moves from the end to be safely and statically pruned.
We also present an extension of the parallel algorithm JLPNS when a PN search is used as the underlying job. In this extension, partial results are regularly sent by the clients to the server.

1 Introduction

In this paper, we address the use of parallelization to solve games. We use the game BREAKTHROUGH [10] as the testbed for our experiments with parallel solving algorithms.

BREAKTHROUGH, which has already been used as a testbed in other work [20, 9], is an interesting game which offers new challenges to the AI community. We therefore also try to improve on domain specific techniques for the game of BREAKTHROUGH. To this effect, we present *race patterns*, a new kind of static patterns that allow to detect win several moves before the actual game ends. The use of race patterns has some links with the use of threats to solve GO-MOKU [1]. However the threats used in GO-MOKU by Allis were designed to select a small number of moves to search, whereas the race patterns are designed to stop search early.

Research on parallel game tree search was initially mainly about the parallelization of the Alpha-Beta algorithm. A survey on the parallelization of Alpha-Beta can be found in Mark Brockington PhD thesis [4]. Other sources about the use of transposition tables in parallel game tree search and Alpha-Beta are Feldmann's paper [8] and Kishimoto's paper [12].

More recently the work on the parallelization of game tree search algorithms has addressed the parallelization of Monte-Carlo Tree Search algorithms [5–7]

Other related works deal with the parallelization of PDS [14, 13] and of Depth First Proof Number search (DF-PN) [11]. A technique to reduce the memory usage of DF-PN is the garbage collection of solved trees [15].

Previous attempts at parallelizing the Proof Number Search (PNS) algorithm used randomization [16] or a specialized algorithm called at the leaves of the main search tree [21].

Proof-Number search and parallel algorithms were also already successfully used in solving Checkers [18, 19].

In this paper we focus on the parallelization of the PN^2 algorithm. The PN^2 algorithm has enabled to solve complex games such as FANORONA [17]. Our goal is to solve such games faster with a similar but parallel algorithm.

The second section is about PNS, Job-Level Proof Number Search (JLPNS) and our algorithm Parallel PN^2 (PPN^2). The third section deals with race patterns at BREAKTHROUGH and the fourth section details experimental results.

2 Job-Level Proof Number Search

In this section we start presenting PNS. Then we recall the parallelization of PNS with Job-Level parallelization. The last section presents our Parallel PN^2 algorithm.

2.1 Proof Number Search

PNS was proposed by V. Allis [2]. The goal of the algorithm is to solve sequential perfect information games. Starting from the root position, it develops a tree in a best first manner. PNS uses the concept of effort numbers to compare leaves.

Effort numbers are associated to nodes in the search tree and try to quantify the progress made towards some goal. Specifically in PNS, two effort number are used: the proof number PN of a node n estimates the remaining effort to prove that n is winning for Max. Conversely, the disproof number DN, estimates the remaining effort to prove a win for Min. Originally, the PN (resp. the DN) of a node n was a lower bound on the number of node expansions needed below n to prove that n is a Max win (resp. loss). When the PN reaches 0 (resp. ∞), the DN reaches ∞ (resp. 0), and the node has been proved to be a Max win (resp. loss).

The PN and DN are recursively defined as shown in Table 1 where Win (resp. Lose) designate a terminal node corresponding to a position won by Max (resp. Min), Frontier designate a non-expanded non-terminal leaf node. *Max* (resp. *Min*) designate an expanded internal node with Max (resp. Min) to play.

To select which node to expand next, Allis defined the set of *most proving nodes* [2] and showed that it is possible to select one of them by the following descent procedure. Iterate until a Frontier node is reached: when at a *Max* node, select a child minimizing PN, when at a *Min* node, select a child minimizing DN.

Node type	PN	DN
Win	0	∞
Lose	∞	0
Frontier	1	1
Max	$\min_{c \in \text{chil}(n)} \text{PN}(c)$	$\sum_{c \in \text{chil}(n)} \text{DN}(c)$
Min	$\sum_{c \in \text{chil}(n)} \text{PN}(c)$	$\min_{c \in \text{chil}(n)} \text{DN}(c)$

Table 1: Determination of effort numbers for PNS

2.2 Job-Level Parallelization

Job-Level Proof Number Search [21] has been used to solve CONNECT6 positions. The principle is to have a main Proof Number Search tree, but instead of having simples leaves, a solver is called at each leaf in order to evaluate it.

In order to avoid having several clients trying to prove the same leaf, JLPNS uses a virtual-loss mechanism.[3] When a leaf is sent to a client, it is temporarily assumed to be proved a loss until the client returns a meaningful result.

A disadvantage of the virtual-loss mechanism is that it is possible for a node to be considered losing for some time, but then to be updated to a non-solved state. Put another way, 0 and ∞ are no longer attractor values for the proof and disproof numbers.

An advantage of the approach taken by JLPNS, is that it allows an easy parallelization over a distributed system with a very small communication overhead.

2.3 Parallel PN^2

The principle of the PN^2 algorithm [1, 3] is to develop another PN search at each leaf of the main PN search tree in order to have more informed proof and disproof numbers. For PPN^2 search, the PN search tree at the leaves is developed on a remote client.

A first difference with JLPNS is that the algorithm that is called at the leaves is also a Proof Number Search instead of a specialized solver as in JLPNS. As a result, partial results from the unfinished remote search in one client can be sent back to the server to update the main PNS tree in order to influence the next searches of the other clients.

Another difference is that we do not use the virtual loss mechanism to avoid currently computed leaves but a flag on these leaves. First, in our technique, a node is never considered to be losing unless it has actually been proved to be losing, thus 0 and ∞ remain attractors. Then, note that the set of most-proving nodes usually contains several nodes, our technique ensures that we will

[3] The authors of JLPNS also try a virtual-win policy and a greedy mechanism which are conceptually similar to virtual-loss [21].

pick tasks from the set of most-proving nodes of the current tree. Finally, the virtual-loss mechanism does not fit well with the partial result update described described in the preceding paragraph.

On the other hand, our algorithm also has BREAKTHROUGH specific knowledge: it uses the mobility heuristic and race patterns defined in Section 3.3.

Just as in PN^2, the size of the remote tree can either be fixed or a function of the size of the main search tree.

The main algorithm which is run on the server, is described in Algorithm 1. It consists in receiving results from the clients and updating the tree according to these results. A result can either be a partial result or a final result. In both cases, we need to update the proof and disproof numbers of the concerned leaf with the result. We also update recursively its ancesters. When the result is final, however, we expand the tree and need to find a new not reserved leaf for the now idle client. Finding a not reserved leaf is done using a backtracking algorithm where the choice points are the nodes with several children minimizing the proof or disproof number.

Algorithm 1 Main algorithm.

```
while root is not proved do
  receive result r from any client c
  if r is a partial result then
    update the PN and DN with r
  else
    expand the tree
    update the PN and DN with r
    if root is proved then
      break
    end if
    find the most proving and not reserved leaf l
    reserve leaf l
    send the position at l to client c
  end if
end while
collect and discard the remaining client messages
send stop to all clients
```

The remote algorithm which is run on the clients is described in Algorithm 2. It consists in developing a PNS tree until a given threshold and regularly sending partial results to the server.

3 BREAKTHROUGH and Race Patterns

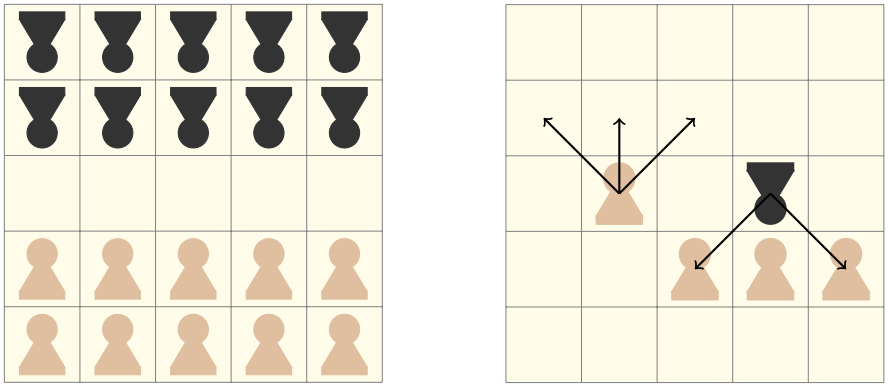
3.1 Rules of BREAKTHROUGH

BREAKTHROUGH is race game invented in 2001 by Dan Troyka. The game is played on a rectangular board of size 8×8. Each player starts with two rows of

Algorithm 2 Remote algorithm.

```
while the stop message is not received do
    receive a position, a player and a threshold N from the server
    while root is not proved and number of descents is less than N do
        if the number of descent is a multiple of a parameter p then
            send as partial results to the server the current PN and the DN of the root
        end if
        expand the tree using Proof Number Search
    end while
    send the definitive results to the server
end while
```

pawns situated on opposite borders as shown in Figure 1a. The pawns progress in opposite direction and the first player to bring a pawn to the opposite last row wins the game. A pawn can always move diagonally forward possibly capturing an opponent pawn but can move forward one cell only if it is empty (Figure 1b).



(a) Starting position on size 5×5.

(b) Possible movements.

Fig. 1: Rules for the game BREAKTHROUGH.

BREAKTHROUGH was originally designed to be played on a 7×7 board but was adapted to participate in the 8×8 board game design competition which it won [10].

3.2 Retrograde Analysis for Small Boards

The state space complexity of BREAKTHROUGH on a board $m \times n$ with $m \geq 2$ and $n \geq 4$ can be upper bounded by the following formula $2^{2m} \times 3^{(n-2)m}$. This formula derives from the fact that each cell on the top row can only be empty or Black, each cell on the bottom row can only be empty or White, and all three possibilities are available for cells in the $n-2$ central rows. This upper bound

is relatively accurate for board with small height, but it includes positions that cannot be reached from standard starting position as does not take into account the fact that each player has at most $2m$ pieces. As a result, it is rather loose for larger boards.

Jan Haugland used retrograde analysis to solve BREAKTHROUGH on small boards.[4] The largest sizes solved by his program were 5×5 and 3×7, both turned to be a second player win.

To reduce the state space complexity and ease the retrograde analysis, Haugland avoided to store positions that could be won in one move. That is, positions with a pawn on the one before last row were not stored. It allows to reduce the state space complexity to $2^{4m} \times 3^{(n-4)m}$. The reduction factor is $r = \frac{3}{2}^{2m}$ which is $r = 58$ when $m = 5$.

3.3 Race patterns

After a couple of games played, human players start to get a sense of tactics in BREAKTHROUGH. It allows the experimented player to spot a winning path sometimes as early as 15 plies before the actual game end. A game of BREAKTHROUGH proceeds as follows, in the opening, the players strive to control the center or to obtain a strong outpost on the opponent's side without exchanging many pieces. Then, the players perform waiting moves until one of them enters a zugzwang position and need to weaken his or her structure. The opponent will now try to take advantage of the breach, usually the attack involves sacrificing one or two pawns to force the opponent's defense to collapse. Thus, at this point both players could *break through* if the opponent passed, and the paths of both are usually disjoint, therefore it is necessary to count the number of moves needed by both players and the quickest to arrive wins (Figure 3a is an example of such a situation).

As we can see, detecting an early win involves looking separately at the possible winning paths of both players and deciding which is the shortest. Formalizing this technique can improve the playing level of an artificial player or the performance of a solver.

Defining race patterns We define *race patterns* that allow to spot such winning paths. To be able to deal softly with the left and right sides of the board, we will consider a generalization of BREAKTHROUGH with walls.[5] Walls are static cells which cannot be traversed nor occupied by any player.

In the following, we assume that we are looking for a winning path for player White. Formally, a *pattern* for player White is a two dimensional matrix in which each element is of one of the following type {*occupied*, *free*, *passive*, *crossable*, *don't care*}. The representation and the relationship between these types is presented in Figure 2. A cell of type *passive* should not contain a Black pawn to

[4] Available on `http://www.neutreeko.net/neutreeko.htm`.

[5] This generalization was used in the 2011 GGP competition.

begin with, but it will not be necessary for any white pawn to cross it. On the other hand it should be allowed to bring a White pawn on a cell of type *crossable*, so it cannot be a wall but it could already hold a White or a Black pawn.

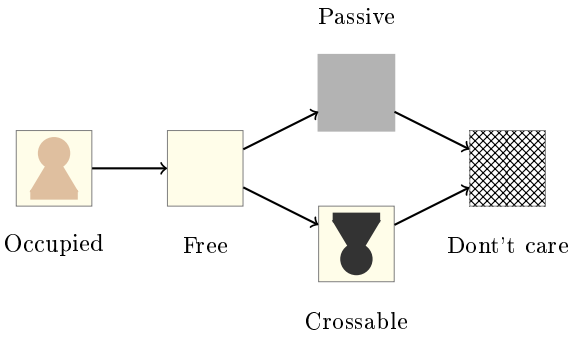


Fig. 2: Pattern representation. An arrow from a to b indicates that any cell satisfying a satisfies b.

To verify whether a pattern is matched on a given board, we first extend the board borders with walls and then check for each possible pattern location that every cell are compatible as defined by Table 2. For instance, if the attacking player is White, then a White pawn will match any cell type in the pattern. Put another way, if the pattern cell corresponding to a Black pawn is not *Crossable* or *Don't Care*, then the pattern does not match.

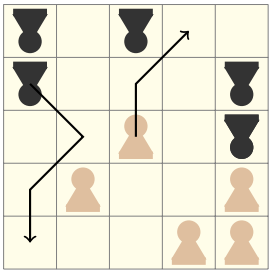
	Occupied	Free	Passive	Crossable	Don't Care
White Pawn	✓	✓	✓	✓	✓
Empty cell		✓	✓	✓	✓
Black Pawn				✓	✓
Wall			✓		✓

Table 2: Checking race patterns for White.

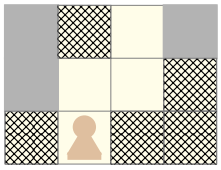
The *order* of a race pattern is defined as the maximal number of pass moves Black is allowed to do before White wins in the restricted position designated by the race patterns.

We compute for each player the lowest-order matching race pattern and if they only intersect on *don't care* cells, we know the outcome of the game. For instance in Figure 3a, we can see that White has two-move second player win

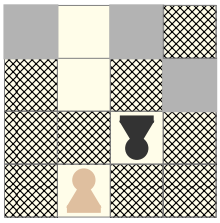
pattern (Figure 3b) and that Black has a three-move first player win pattern (Figure 3c). Given that player Black does not have a one-move nor a two-move race pattern, we conclude that the position is a White win. It is thus possible to statically solve this position four moves before the actual game end.



(a) Sample game with Black to play. White player can force a win.



(b) Two-move second player win pattern.



(c) Three-move first player win pattern.

Fig. 3: Early win detection using race patterns. White can match pattern (b) and Black can match pattern (c).

In general, this technique allows to solve positions $2 \times n$ moves before the actual game end only if we have access to every n-move race pattern. However, a position cannot be solved this way if its solution tree involves a zugzwang.

In our experiments, we used 26 handwritten patterns of order up to 2. The biggest patterns we used were 4×3 such as the one presented in Figure 3b. We do not have yet a tool for automatic correctness checking, therefore we had to limit the number of patterns used to keep confidence in their correctness.

4 Experimental Results

Experiments are done on a simple network of Linux computers connected with Gigabyte switches. The network area includes 17 computers with 3.2 GHz Intel i5 quad core CPU with 4 GB of RAM. The master is run alone on one of these computers. The maximum number of clients is set to 16×4.

In the following experiments, we report the total time needed to solve the starting position of a BREAKTHROUGH game of various sizes. We also report the number of nodes expanded and touched that were needed in Algorithm 1.

A node is expanded when all of its children have been added to the tree. In our server-side implementation, one node is expanded per iteration. The number of nodes expanded is proportional to the memory needed to store the PN tree on the server side. It also corresponds to the total number of tasks that have been sent to the clients. For a touched node, we only store the proof and disproof numbers as given by a client search. On the other hand, an expanded node also

needs to store a pointer to every child. As a result touched nodes take much less memory than expanded nodes. We bounded to 1k or 100k the number of descents in one search in the clients, so memory resources in the clients were never a problem in these experiments.

4.1 Scalability

Table 3 gives the time needed in sec., the number of expanded and touched nodes saved on the server side to solve the 4×5 game with the PPN^2 algorithm with 1k descents in the clients. Solving 4×5 with 64 clients with 100k descents in each remote search takes 577 seconds, while with 10k descents in each remote search, it takes 216 seconds. Therefore, increasing the number of descents in the remote search does not necessarily improve the solving time. On the other hand, performing 100 descents in each remote search made it necessary to go over 1m descents in the main search which is very memory consuming.

Clients	Time	Speed-up	Expanded	Touched
1	3397s		107k	915k
4	1559s	2.2	126k	1073k
8	803s	4.2	130k	1106k
16	472s	7.2	152k	1298k
32	305s	11.1	196k	1651k
64	186s	18.3	232k	1930k

Table 3: Time needed, number of expanded and touched nodes for the PPN^2 with fixed search size on 4×5 board with 1k descents at most in the remote search. Partial results were sent from a client every 100 descents.

We can see from Table 3, that the number of expanded nodes on the server increases as the number of clients rises. Put another way, running many clients in parallel makes it harder to avoid unnecessary work. This is an expected behavior in a parallel algorithm. Nevertheless, the time needed to solve the position also decreases steadily as the number of clients rises. The speedup factor with 8 and 64 clients compared to 1 single client are respectively 4 and 18. We can conclude that although the algorithm is not perfectly parallelizable, the scaling factor is satisfactory.

4.2 Partial Results Updates

Table 4 gives the time needed in sec., the number of expanded and touched nodes saved in memory to solve the 4×5 game with the PPN^2 algorithm with partial results. The first column gives the partial results frequency. Each solved position turned to be a second-player win.

Partial	Time	Expanded	Touched
None	263s	324k	2645k
500	233s	281k	2336k
250	205s	253k	2105k
100	186s	232k	1930k
50	190s	233k	1944k
25	201s	243k	2023k
12	193s	223k	1855k

Table 4: Time needed, number of expanded and touched nodes for PPN^2 algorithm with partial results and fixed remote search size of 1k descents on 4×5 board, involving 64 clients.

As we can see, sending partial results makes it possible to direct better the search but also increases the communication overhead. It is therefore needed to find a balance between spending too much time in communications and not taking advantage of the information available. In this setting, sending partial results every 100 descents in the client seems the best compromise. Using partial informations, solving time is less dependent to the search size.

4.3 Patterns

Table 5 gives the time in sec. and the number of expanded nodes needed to solve different games with the PPN^2 algorithm with partial results, fixed search size and some patterns. The patterns we used allowed to statically solve a position up to 4 moves before the game end, 26 patterns were hand-written for this purpose. Checking whether a pattern can be matched on a given position is done in the most naive way and implementing more efficient pattern matching techniques is left as future work.

Using race patterns, the solving time is divided by 5.96 for the 4×5 board with 1k search, and by 9.85 for the 5×5 board. It takes 927 sec. to solve the 5×5 board with 1k search in the clients. Without patterns, 5×5 board with 1k search fails with 1 million nodes saved and goes beyond the server allowable memory with 2 million nodes.

Combining PPN^2 and race patterns allows us to solve the 6×5 board in 25,638 sec. (i.e. 7 hours 7 minutes 18 sec.) with 10k search and in 47,134 sec. (i.e. 13 hours 5 minutes 34 sec.) with 100k search.

As we can see, using race patterns makes it unnecessary to examine many positions in the main search. Race patterns also allow for a time reduction of one order of magnitude on boards of small sizes and probably more on larger boards.

Board size	Search size	Patterns	Time	Expanded
5×4	1k	No	2s	4132
5×4	1k	Yes	1s	72
4×5	1k	No	161s	241k
4×5	1k	Yes	27s	4k
5×5	1k	Yes	927s	78k
5×5	100k	No	29,170s	208k
5×5	100k	Yes	2959s	3k
6×5	10k	Yes	25,638s	14k
6×5	100k	Yes	47,134s	21k

Table 5: Time needed and number of expanded nodes for the PPN^2 algorithm with partial results, fixed remote search size and patterns with 64 clients.

5 Discussion and Conclusion

In this paper, we have defined race patterns and used them to ease the solving of BREAKTHROUGH positions. Indeed, in our experiments, using race patterns usually allows to examine about two orders of magnitude fewer positions. We have also shown how to successfully parallelize the PN^2 algorithm. The PPN^2 algorithm associated to race patterns has enabled to solve 6×5 BREAKTHROUGH: the game is a second player win. We have found that on the smaller 4×5 board the speedup due to parallelization is important until at least 64 clients.

In future work, we will try to solve BREAKTHROUGH for larger sizes. The race patterns used in this work had been devised by hand but it is impractical if we need many more patterns to statically solve positions earlier. We therefore need to devise an algorithm to generate and check for correctness race patterns automatically.

Zugzwang positions are still difficult to solve. Indeed, no winning race pattern will be found in a zugzwang position, so an extension of the concept of race patterns to be compatible with zugzwang positions or an orthogonal technique would be desirable.

We will also apply the Parallel PN^2 algorithm to other games. Moreover we will try to enhance the algorithm itself in order to have even greater speedups.

References

1. Louis Victor Allis. *Searching for Solutions in Games an Artificial Intelligence*. Phd thesis, Vrije Universitat Amsterdam, Department of Computer Science, Rijksuniversiteit Limburg, 1994.
2. Louis Victor Allis, M. van der Meulen, and H. Jaap van den Herik. Proof-Number Search. *Artificial Intelligence*, 66(1):91–124, 1994.

3. Dennis Michel Breuker. *Memory versus Search in Games.* Phd thesis, Universiteit Maastricht, 1998.
4. Mark Brockington. *Asynchronous Parallel Game-Tree Search.* Phd thesis, University of Alberta, 1997.
5. Tristan Cazenave and Nicolas Jouandeau. On the parallelization of UCT. In *Proceedings of the Computer Games Workshop*, pages 93–101, 2007.
6. Guillaume Chaslot, Mark Winands, and H. Jaap van Den Herik. Parallel monte-carlo tree search. *Computers and Games*, pages 60–71, 2008.
7. Markus Enzenberger and Martin Müller. A lock-free multithreaded monte-carlo tree search algorithm. *Advances in Computer Games*, pages 14–20, 2010.
8. R. Feldmann, P. Mysliwietz, and B. Monien. Game tree search on a massively parallel system,. In *Advances in Computer Chess 7*, pages 203–219, 1993.
9. Hilmar Finnsson and Yngvi Björnsson. Game-tree properties and mcts performance. In *Proceedings of the IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'11)*, pages 23–30, 2011.
10. Kerry Handscomb. 8×8 game design competition: The winning game: Breakthrough... and two other favorites. *Abstract Games Magazine*, 7:8–9, 2001.
11. Tomoyuki Kaneko. Parallel depth first proof number search. In *AAAI*, 2010.
12. A. Kishimoto and J. Schaeffer. Distributed game-tree search using transposition table driven work scheduling. In *Proceedings. International Conference on Parallel Processing*, pages 323–330. IEEE, 2002.
13. Akihiro Kishimoto and Yoshiyuki Kotani. Parallel and/or tree search based on proof and disproof numbers. In *Game Programming Workshop in Japan*, pages 24–30, 1999.
14. A. Nagai. A new and/or tree search algorithm using proof number and disproof number. In *Complex Games Lab Workshop*, pages 40–45, ETL, Tsukuba, Japan, 1998.
15. Ayumu Nagai. *Df-pn algorithm for searching AND/OR trees and its applications.* PhD thesis, 2002.
16. Jahn-Takeshi Saito, Mark H. M. Winands, and H. Jaap van den Herik. Randomized parallel proof-number search. In H. van den Herik and Pieter Spronck, editors, *Advances in Computer Games*, Lecture Notes in Computer Science, pages 75–87. Springer Berlin / Heidelberg, 2009.
17. Maarten P. D. Schadd, Mark H. M. Winands, Jos W. H. M. Uiterwijk, H. Jaap van den Herik, and M. H. J. Bergsma. Best Play in Fanorona leads to Draw. *New Mathematics and Natural Computation*, 4(3):369–387, 2008.
18. Jonathan Schaeffer, Yngvi Björnsson, Neil Burch, Akihiro Kishimoto, Martin Müller 0003, Robert Lake, Paul Lu, and Steve Sutphen. Solving checkers. In *IJCAI*, pages 292–297, 2005.
19. Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518, 2007.
20. Pálmi Skowronski, Yngvi Björnsson, and Mark Winands. Automated discovery of search-extension features. In H. van den Herik and Pieter Spronck, editors, *Advances in Computer Games*, volume 6048 of *Lecture Notes in Computer Science*, pages 182–194. Springer Berlin / Heidelberg, 2010.
21. I-Chen Wu, Hung-Hsuan Lin, Ping-Hung Lin, Der-Johng Sun, Yi-Chih Chan, and Bo-Ting Chen. Job-level proof-number search for Connect6. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games*, volume 6515 of *Lecture Notes in Computer Science*, pages 11–22. Springer, 2010.