PN, PN^2 AND PN* IN LINES OF ACTION

Mark H.M. Winands and Jos W.H.M. Uiterwijk

Universiteit Maastricht, Department of Computer Science, Institute of Knowledge and Agent Technology IKAT, P.O. Box 616, 6200 MD Maastricht, The Netherlands.
Email: {m.winands, uiterwijk, herik}@cs.unimaas.nl.

ABSTRACT

The endgame of Lines of Action (LOA) is problematic because (1) LOA evaluation functions are not good predictors in the case of forced wins and (2) the state-space complexity of the LOA endgame is too large to use endgame databases. Because of the frequency of forced moves and the mobility component in the LOA endgame the utility of proof-number (PN) search is investigated in this paper. Also the PN variants PN^2 and PN* are tested and compared with each other. In the case of PN^2 small enhancements (caching second-level subtrees, deleting (dis)proved subtrees in the second-level search, and quitting second-level search) are investigated. Finally, it is discussed how to combine PN search with an alpha-beta search by using transposition tables.

1. INTRODUCTION

Most modern game-playing computer programs successfully use an α-β search with enhancements for *online* game-playing. Unfortunately, the α-β search per se sometimes is not sufficient to play well in the endgame. In some games, such as chess, this problem can be solved by the use of endgame databases (Van den Herik and Herschberg, 1985) in the $\alpha\beta$ search. One of the drawbacks of this method is that it is only feasible, due to memory constraints, for endgames with a relatively small state-space complexity. An alternative approach is the use of a specialised binary (win or non-win) search method, like proof-number (PN) search (Allis *et al*,. 1994). In some games PN search outperforms α-β search in proving the game-theoretic value of endgame positions. PN search has been applied successfully to the endgame of Awari (Allis *et* al., 1994), chess (Breuker *et al.*, 1994a), checkers (Schaeffer and Lake, 1996) and Tsumi-Shogi (Seo *et al.*, 2001). In this paper we look at several PN algorithms, which are tested in the domain of Lines of Action (LOA).

The article is organised as follows. Section 2 explains the rules of LOA, and Section 3 describes PN, PN^2 and PN* with their (possible) enhancements. The experimental set-up is explained and the results of the experiments are presented in Section 4. Section 5 contains the conclusions and some suggestions for future research.

2. LINES OF ACTION

Lines of Action (LOA) is a two-person zero-sum game with perfect information; it is a chess-like game with a connection-based goal played on an 8×8 board. LOA is invented by Claude Soucie around 1960. Sid Sackson (1969) described it in his first edition of *A Gamut of Games*. LOA has an average game length of 38 plies and an average branching factor of 30 (Winands, 2000). An interesting property of the game is that most terminal positions still have more than 10 pieces remaining on the board, which makes the game not suitable for endgame databases. Although reasonable effort has been undertaken to construct adequate evaluation functions for LOA (Winands *et al.*, 2001a), experiments still suggest that these are not very good predictors

in the case of forced wins. Therefore, LOA seems an appropriate test domain for PN algorithms.

2.1. Rules of LOA

LOA is played on an 8×8 board by two sides, Black and White. Below the rules appearing in the second edition of *A Gamut of Games* (Sackson, 1982) are formulated in eight points.

1. The black pieces are placed in two rows along the top and bottom of the board, while the white pieces are placed in two files at the left and right side of the board. The initial position of the game is shown in Diagram 1.
2. The players alternately move, starting with Black.
3. A player to move must move one of its pieces. A move takes place in a straight line, exactly as many squares as there are pieces of either colour *anywhere* along the line of movement. (These are the *Lines of Action.*)
4. A player may jump over its own pieces.
5. A player may not jump over the opponent's pieces, but can capture them by landing on them.
6. The goal of a player is to turn all own pieces on the board into one connected unit. The first player to do so is the winner. The connections within the group may be either orthogonal or diagonal. For example, in Diagram 2 Black has won because the black pieces form one connected unit.
7. If one player's pieces are reduced by captures to a single piece, the game is a win for this player.
8. If a move simultaneously creates a single connected unit for both players, the game is a win[1].

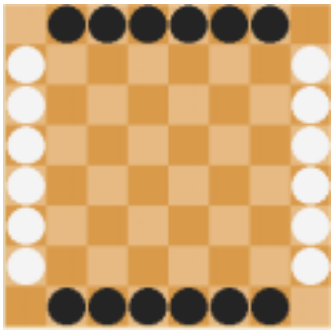

Diagram 1: The initial position.

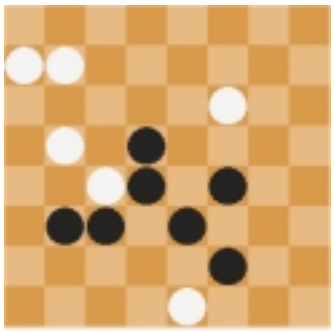

Diagram 2: A terminal position.

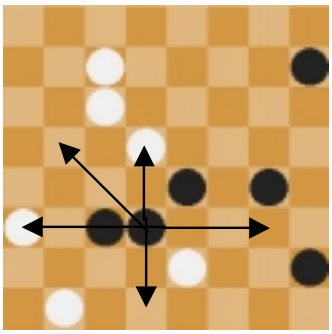

Diagram 3: Movement of pieces.

Since some situations are not covered by the original rules, ad hoc rules are introduced. For instance: (1) repetition of positions is considered as a draw, and (2) if a player cannot move, this player loses.[2]

In the article we use the standard chess notation. The possible moves of the black piece on **d3** in Diagram 3 are indicated by arrows. The piece cannot move to **f1** because its path is blocked by an opposing piece. The move to **h7** is not allowed because the square is occupied by a black piece.

[1] In the first edition of *A Gamut of Games* (1969) simultaneous connection is described as a draw.
[2] The use of this rule is disputable, some LOA players allow the opponent to pass in such positions.

3. DESCRIPTION OF THE SEVERAL PROOF-NUMBER SEARCH ALGORITHMS.

In this section we give a short description of the PN-, PN^2- and PN^*-search algorithms. Some possible enhancements are discussed.

3.1. Proof-number search

Proof-number (PN) search is a best-first search algorithm especially suited for finding the game-theoretical value in game trees (Allis, 1994). Its aim is to prove the true value of the root of the tree. A tree can have three values: *true*, *false* or *unknown*. In the case of a forced win, the tree is *proved* and its value is true. In the case of a forced loss or draw, the tree is *disproved* and its value is false. Otherwise the value of the tree is unknown. In contrast to other best-first algorithms PN search does not need a domain-dependent heuristic evaluation function in order to determine the most-promising node (to be expanded next) (Breuker *et al.*, 1994a). PN search selects the most-promising node using two criteria: (1) the shape of the search tree (the number of children of every internal node) and (2) the values of the leaves. These two criteria enable PN search to treat efficiently game trees with a non-uniform branching factor. A disadvantage of PN search is that the whole search tree has to be stored in memory. When the memory is full, the search process has to be terminated prematurely or countermeasures have to be taken (Allis, 1994; Breuker, 1998).

In the naïve implementation, proof and disproof numbers are each initialised to unity at the unknown leaves. In other implementations, to distinguish between leaves, before expansion, we set the proof number and disproof number to 1 and *n* for a MAX[3] node (and the reverse for a MIN node), where *n* is the number of legal moves. This means that we actually are using a mobility component, which is reported to be promisingly in LOA (Billings and Björnsson, 2000). For the rest of the paper we will call initialisation by the number of moves *mobility*. Notice, that we have an overhead of counting the number of nodes. Therefore we have pre-computed for each possible line configuration the number of moves for each side. This idea originates from Björnsson (2000).

In (Winands *et al.*, 2001b) a method of combining PN search with α-β search is presented in real-time applications. Nodes once proved by PN search are stored in a transposition table (TT), which is used in the α-β search. A proved-node transposition table (TT) is implemented as a hash table. The well-known Zobrist-hashing method is used. Each entry in the transposition table has a length of 64 bits. These bits are reserved for the hash key to distinguish among different positions having the same hash index. When two positions have the same hash index, the last examined position is preferred over early ones (replacement scheme *New*, see (Breuker *et al.*, 1994b)). The usual schemes *Big* or *Deep* are not used, because the number of nodes of a subtree or the depth of a subtree are not appropriate measures for replacement in case of PN search. A position is stored if and only if it is a proved interior node. At present, disproved nodes are not stored because it is not known whether the game-theoretic value of the node is a loss or a draw. If it is known that draws are impossible in a game (such as Hex), then it is possible to store also the disproved nodes in a separate table. The PN-TT can be also used in the PN search, which performance is tested in Section 4. We want to determine what the possible benefit is of storing only proved positions in a transposition table.

[3] In this article MAX and MIN node can be read as OR and AND node, respectively.

3.2. PN^2 search

PN^2 is first described by Allis (1994), as a technique to reduce memory requirements in the PN search. Its implementation and testing for chess positions was performed by Breuker *et al.* (2001). PN^2 consists of two levels of PN search. The first level consists of a PN-search (pn_1), which calls as evaluation of a node a PN search at the second level (pn_2). This pn_2 search is bound by a maximum number N of nodes stored in the memory. This number N is a fraction of the size of the pn_1-search tree and given by the function $f(x)$, x being the number of nodes of the first-level search:

$$f(x) = \frac{1}{1+e^{(a-x)/b}}$$

with parameters a and b, both strictly positive. The parameters (a,b) are set at (1800K, 240K). The pn_2 search is stopped when the numbers of nodes exceeds N or the subtree is (dis)proved. After completion of the pn_2-search tree, the children of the root of the pn_2-search tree are preserved, but subtrees are removed from memory.

We remark that several enhancements to PN^2 search have been suggested. One example of such an enhancement suggested by Schaeffer involves storing the deleted pn_2-search trees in a cache, instead of deleting them. Another way is to cache pn_2 search trees until a certain bound is reached (e.g. 10% of N in our implementation). Then the trees are randomly deleted until it is possible again to cache trees. This is tested in Subsection 4.2.

A disadvantage is that pn_2 search only stops when the root is (dis)proved or N is reached. A suggestion is to quit the pn_2 search earlier, when it will not likely to be proved. We have implemented a simple mechanism, called *Quitting*. If the root of the pn_2 search is a MAX (OR) node, we quit the pn_2 when the proof number is two times the disproof number. The danger exist that we quit too early in the root position when there are very few moves. Therefore, proof and disproof number have to exceed a threshold (in our implementation 100).

As we have seen in subsection 3.1, proved or disproved subtrees can be deleted. If we do not delete subtrees in the pn_2 search the number of nodes searched is the same as N, otherwise we can continue the search longer. The question is whether this is also beneficial to do in the pn_2 search? If there are many (dis)proved trees the pn_2 search will continue much longer. More effort is taken in trees where there are many (dis)proved nodes than in trees where there are few. It is likely that almost (dis)proved subtrees will be (dis)proved now and therefore there will be less regeneration. But almost disproved subtrees are unlikely to be visited twice and to be regenerated, because the goal is to prove the root. Thus, the extra effort taken to disprove the subtree can be wasted.

3.3. PN* search

PN and eventually also PN^2 suffer from memory disadvantages. This problem is tackled by PN* search (Seo *et al.*, 2001). PN* transforms the PN-search algorithm into an iterative-deepening depth-first approach. For a description of this algorithm see (Seo *et al*., 2001). PN* is enhanced with methods as recursive iterative deepening and dynamic evaluation and successor ordering. The last one is not implemented at the moment.

4. EXPERIMENTS AND RESULTS

In this section we will compare PN, PN^2, PN^* and α-β with each other. For PN and PN^2 some enhancements are tested.

4.1. Experimental details

All experiments have been performed in the framework of the tournament program MIA (Maastricht In Action). MIA can be played at the website: http://www.cs.unimaas.nl/m.winands/loa/. It has been written in Java and runs on every well-known operating system. MIA performs an α-β depth-first iterative-deepening search. The program uses a *two-deep* transposition table (Breuker *et al.*, 1996), the history heuristic (Schaeffer, 1983) and killer moves (Akl and Newborn, 1977). For all experiments described in this paper, the null move is not used because of possible negative effects of zugzwang in LOA (Winands, 2000).

For PN and PN^2 search all leaf nodes evaluated during the search are counted, while for the α-β depth-first iterative-deepening searches nodes at depth *i* are counted only during depth *i*. This is the way of comparison as done in Allis (1994). For PN^* all expanded nodes are counted. The maximum number of nodes searched is 50,000,000. This limit corresponds roughly to tournament conditions. The maximum number of nodes stored in the memory is 900,000. PN, PN^2, PN^* and $\alpha\beta$ are tested on a set of 116 LOA win positions.

4.2. PN and PN^2

In table 1 we have compared PN search with several enhancements. From the 116 positions, 85 positions were solved by the PN search using mobility, 53 positions by using only TT, and only 44 positions when nothing was used. The TT improved the PN search with 20% when nothing was used. In the last column of table 1 we see that TT only improves the search with 10%, when mobility is used. If we compare PN+TT with PN+TT+Mobility, 31 additional positions are solved. In the set of 53 positions that both could solve, the one using mobility was 5 times faster.

	# of pos. solved (out of 116)	Total Nodes (44 positions)	Total Nodes (53 positions)	Total Nodes (85 positions)
PN	44	16,266,638	-	-
PN+TT	53	13,363,676	24,357,832	-
PN+Mobility	85	2,171,628	5,423,500	23,102,938
PN+TT+Mobility	85	2,133,311	5,053,630	21,443,322

Table 1: Mobility and Transposition Table in PN.

In table 2 we have compared PN^2 search with several enhancements. From the 116 positions, 109 positions were solved by the PN^2 search using TT+ Mobility, 107 positions by using only mobility, 91 positions by only using TT and 89 positions when nothing was used. The TT improved the PN search with 20% when nothing was used. In the last column of table 2 we see that TT only improves the search with 3%, when mobility is used. If we compare PN+TT with PN+TT+Mobility, 18 additional positions are solved more. In the set of 91 positions that both could solve, the one using mobility was 6 times faster.

	# of pos. solved (out of 116)	Total Nodes (89 positions)	Total Nodes (91 positions)	Total Nodes (107 positions)
PN^2	89	334,650,398	-	-
PN^2+TT	91	273,503,443	345,986,639	-
PN^2+Mobility	107	44,351,259	57,271,268	387,248,435
PN^2+TT+Mobility	109	41,461,089	56,809,635	376,022,338

Table 2: Mobility and Transposition Table in PN^2.

In general we can conclude that mobility speeds up the PN and PN^2 with a factor 5 to 6. The extra time spent on this is only 20%. Thanks to mobility PN search can solve many more positions, because the memory constraint is violated less. Storing proved positions is a clear improvement in PN search, but is less so in PN^2.

4.3. PN^2 Enhancements

In table 3 we have compared several possible PN^2 enhancements with each other. In the first row the results of the plain PN^2-search algorithm is given. The second row gives the result of deleting (dis)proved nodes at the pn_2 search. In the third row the results are given when deleted search trees are cached. In the fourth row the results are given when probably-unsolvable trees are left prematurely. In the second column we can see that deleting (dis)proved trees improves the search with 10% and solves one position more. In the third column we can see that the caching and quitting enhancements do not speed up the search significantly. Probably, the reason caching is not successful is that when pn_2 trees are going to be regenerated, the tree is not available in the cache anymore.

	# of pos. solved (out of 116)	Total Nodes (108 positions)	Total Nodes (109 positions)
PN^2	108	463,076,682	-
PN^2 + Delete	109	416,168,419	464,606,624
PN^2 + Delete + Cached	109	-	463,060,849
PN^2 + Delete + Quitting	109	-	463,342,848

Table 3: PN^2 enhancements.

4.4. The search algorithms compared

In table 4 we have compared PN, PN^2, PN^* and α-β with each other. From the 116 positions, 109 positions were solved by the PN^2 search, 88 positions by only using PN^*, 85 positions by using PN, and 70 positions by α-β. If we have a look at the second column, we see that α-β is 40 times slower than PN or PN^2. Even the not-tuned PN^* is almost 10 times faster than α-β. The third column suggests that PN^* is 6 times slower than PN^2, but for the harder positions showed in the fifth column, the disadvantage is maybe less. But this set of 86 positions is too unbalanced. PN and PN^2 are compared in the fourth column. PN^2 is 1.3 times slower than PN search.

	# of pos. solved (out of 116)	All	Without α-β	Without α-β and PN*	Without α-β and PN
		Total Nodes (68 pos.)	Total Nodes (76 pos.)	Total Nodes (85 pos.)	Total Nodes (86 pos.)
α-β	70	487,494,281	-	-	-
PN*	88	52,219,919	128,478,302	-	298,464,775
PN	85	10,739,340	16,062,379	21,443,322	-
PN^2	109	14,865,337	21,820,252	27,955,501	182,668,261

Table 4: Comparing the search algorithms.

In general we can say that PN, PN* or PN^2 search are good mate provers in LOA. For instance, the position showed in Diagram 1 is a win in 21 ply. PN can solve this position investigating 484,453 nodes, whereas α-β is not able to solve this position within 50,000,000 nodes. PN solves fewer positions than PN^2 or PN* due to memory constraint. We would like to remark that PN* is not able to solve 9 positions, which are solved by PN. But PN* solves 2 positions, which PN^2 cannot prove (e.g., Diagram 5). Finally, because of the constraint of maximum nodes searched was set to 50,000,000 PN^2 did not run in memory problems. Therefore, this experiment shows not the real advantage of PN* over PN^2.

Diagram 4: Black to move and to win in 21 plies. MONA vs. YL, Game 4.[4]

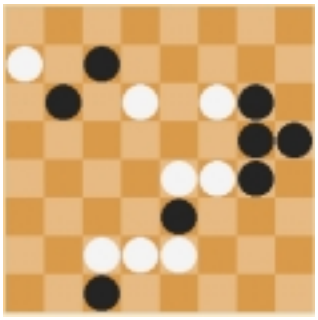

Diagram 5: White to move and win in 13 plies. Fifth Annual E-mail Tournament, Thordsen vs. MONA, after 14. c3:e3.[4]

5. CONCLUSIONS

In this paper we have seen that PN, PN*, PN^2 search are quite good proving LOA positions. Even a poor implemented PN* outperforms α-β in LOA positions. It is a subject of future research to enhance PN* with a good successor ordering. Mobility improves the PN search, whereas the benefits of a transposition table are less. Deleting (dis)proved nodes in pn_2 improves the PN^2 search. The ideas of Caching and Quitting have not paid off yet.

REFERENCES

Akl, S.G. and Newborn, M.M. (1977). The Principal Continuation and the Killer Heuristic. *1977 ACM Annual Conference Proceedings*, pp. 466-473. ACM, Seattle.

Allis, L.V., Meulen, M. van der, and Herik, H.J. van den (1994). Proof-Number Search. *Artificial Intelligence*, Vol. 66, pp. 91-124.

[4] The positions and their solution can be found at: http://www.cs.ualberta.ca/~darse/LOA.

Allis, L.V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. thesis Rijksuniversiteit Limburg, Maastricht, The Netherlands. ISBN 90-9007488-0.

Billings, D. and Björnsson, Y. (2000). Mona and YL's Lines of Action Page. http://www.cs.ualberta.ca/~darse/LOA.

Björnsson, Y. (2000). *Personal Communication*.

Breuker, D.M., Allis, L.V., and Herik, H.J. van den (1994a). How to Mate: Applying Proof-Number Search. *Advances in Computer Chess 7* (eds. H.J. van den Herik, I.S. Herschberg and J.W.H.M. Uiterwijk), pp. 251-272. University of Limburg, Maastricht, The Netherlands. ISBN 90-621-6101-4.

Breuker, D.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (1994b). Replacement Schemes for Transposition Tables. *ICCA Journal*, Vol. 17, No. 4, pp. 183-193.

Breuker, D.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (1996). Replacement Schemes and Two-Level Tables. *ICCA Journal*, Vol. 19, No. 3, pp. 175-180.

Breuker, D.M. (1998). *Memory versus Search in Games*. Ph.D. thesis, Universiteit Maastricht. ISBN 90-9012006-8.

Breuker, D.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001). The PN^2-Search Algorithm. *Advances in Computer Chess 9* (eds. H.J. van den Herik and B. Monien), pp. 115-132. Universiteit Maastricht, Maastricht, The Netherlands. ISBN 90-6216-566-4.

Herik, H.J. van den and Herschberg, I.S. (1985). The Construction of an Omniscient Endgame Data Base. *ICCA Journal*, Vol. 8, No. 2, pp. 66-87.

Sackson, S. (1969). *A Gamut of Games*. Random House, New York, NY, USA. A second edition (1982) has been republished in 1992 by Dover Publications, New York, NY, USA. ISBN 0-486-27347-4.

Schaeffer, J. (1983). The History Heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16-19.

Schaeffer, J. and Lake, R. (1996). Solving the Game of Checkers. *Games of No Chance* (ed. J. Nowakowski). *MSRI Publications*, Vol. 29, pp. 119-133. Cambridge University Press, Cambridge, UK. ISBN 0-521-57411-0.

Seo, M., Iida, H., and Uiterwijk, J.W.H.M. (2001). The PN*-Search Algorithm: Application to Tsume-Shogi. *Artificial Intelligence*, Vol. 129, Nos 1-2, pp. 253-277.

Winands, M.H.M. (2000). *Analysis and Implementation of Lines of Action*. M.Sc. thesis. Universiteit Maastricht, Maastricht, The Netherlands.

Winands, M.H.M, Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001a). The Quad Heuristic in Lines of Action. *ICGA Journal*, Vol. 24, No. 1, pp 3-15.

Winands, M.H.M, Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001b). Combining Proof-Number Search with Alpha-Beta Search. Accepted for the *Proceedings of the Thirteenth Belgium-Netherlands Artificial Intelligence Conference* (BNAIC 2001).