Transposition Tables in Computer Chess

Dennis M. Breuker and Jos W.H.M. Uiterwijk
Department of Computer Science
University of Limburg
P.O. Box 616
6200 MD Maastricht, The Netherlands
{breuker,uiterwijk}@cs.rulimburg.nl

ABSTRACT

Search algorithms are used in many game-playing programs. To enhance the strength of these programs additional heuristics and improvements are used. One of these improvements is the use of *transposition tables*. In these tables information concerning a position investigated can be stored in order to re-use this information when the same position is encountered again during the search process (a *transposition*). Amongst the information a transposition-table entry contains is the best move for that position and the score of that move. In order to estimate their contributions we have done experiments separating the effects of using the transposition move and the transposition value.

The transposition table is typically implemented as a large hash table. Even though this table is usually made as large as possible, subject to memory constraints, collisions occur frequently. When a collision occurs, a choice has to be made which position to store or keep in the table (using some replacement scheme). A second topic of our research is to compare the performance of several replacement schemes. The experiments are conducted on middle-game positions taken from chess games.

Doubling the number of entries in the transposition table reduces the size of the search tree. This is an obvious result, since the more information in the table, the larger the chance that the information in the table can be re-used during search. Part of our research then is to quantify *how much* the size of the search tree is reduced when doubling the number of entries in the transposition table.

It is known that the benefits of doubling the number of entries decrease at large table sizes. Therefore, it is interesting to examine other means of using much memory than increasing the number of entries in the transposition table. In particular, what additional information can be stored in a transposition-table entry to speed up the search? Preliminary results are given and some suggestions to be tested in the near future will be made.

1 Introduction

Board games are an essential part of how people use their spare time. Most of these games, such as chess, are *two-person*, *zero-sum* games with *perfect information* (Von Neumann and Morgenstern, 1944). In two-person games there are two adversary players who alternately make a move, transforming a position into a new position. In zero-sum games the gain of a player is equal to his opponent's loss. For instance, in checkers, if a checker is captured, the capturing player has won a checker, while his opponent has lost one. Perfect information means that each player has complete information about the position and the choices available to him. Thus, at each turn the rules of the game define both which moves are legal and what effect each possible move will have, leaving no room for luck. This is in contrast with games with imperfect information or luck, such as bridge or backgammon, respectively.

1.1 Games and Artificial Intelligence

One of the advantages of the domain of intelligent (board) games for Artificial-Intelligence (AI) research is that it is well structured. These games have well-defined rules, and are complex enough at the same time. For instance, a child can learn the rules of a game like chess in a few days, but chess professionals still make mistakes because of the enormous complexity of the game. In the fifties, Alan Turing and Claude Shannon, two pioneers in the field of AI, proposed computer chess as a research domain in AI. The first sentence of Turing's (1950) article reads: "I propose to consider the question, "Can machines think?"". To answer this question he formulated "the imitation game", nowadays known as the *Turing test*. This test requires a person (interviewer) to ask questions (through a terminal) to a machine and a human. The interviewer does not know who is the machine and who is the human. If the interviewer is not able to distinguish between the human and the machine, the machine is said to be intelligent. Shannon (1950) wrote one of the first articles about computer chess. In this article he collected ideas from several people and formalized them into the concept of a chess program. His ideas are still used in today's chess playing programs.

The purpose of research on computer chess is twofold. First, many useful techniques can be developed, which also can be used in other domains. Second, by developing a chess-playing program, we hopefully get more insight in the human thinking process. This is based on the idea that chess can be seen as an intellectual game (because it requires non-trivial thinking). Therefore, not only computer scientists, but also psychologists are interested in computer chess. Newell and Simon wrote their own chess program (Newell *et al.*, 1958) in order to get more insight in the human way of reasoning. The models of human problem-solving abilities they developed later (Newell and Simon, 1972) were mainly based on insights they obtained during their research on computer chess.

In due time, it appeared that a chess program which searches all moves in every position (the brute-force approach) played better than chess programs which used the human way of playing chess (i.e., searching only a few moves at every position). Although the former type of programs are able to think according to the Turing test, they do not give insight in the human thinking process. Therefore, today, many people believe that computer-games research is not any longer at the core of AI. We believe this to be not true, since it presently is being used

as a test bed for many new techniques under development, such as parallel programming (Feldmann, 1993) and advanced new search methods using either little knowledge (proof-number search (Breuker *et al.*, 1994a)) or much knowledge (opponent-model search (Iida *et al.*, 1994)).

1.2 Games and search

When a program is searching for the best move in a position, a *search tree* is generated. A node in the tree represents a position in the game, while a branch represents a move. Each node is expanded by generating all successors of the position. If a node represents an *end position* (as determined by the rules of the game), it needs not be expanded. In an end position the rules of the game determine whether the result is a win, a draw, or a loss. We distinguish two types of nodes in the tree: nodes with successors (*interior* nodes), and nodes without successors (*terminal* nodes). We note that the root node (or root for short), representing the current board position, is a special case of an interior node, i.e., the only node without predecessor. The direct successors of an interior node are termed the *children* of the node (which in turn is termed *parent* of its children). A path from the root to a terminal node is called a *variation*. The depth of a (sub)tree is often counted in *plies*. A ply can be viewed as a half move (a move by one of the two players). The term ply was firstly introduced by Samuel (1959).

In order to choose a move in a position a backtrack mechanism is invoked. It requires all terminal nodes in the search tree to have a value assigned (the *score* of that node). This score, representing the goodness of a position, is given by an *evaluation function*. The better a position, the higher its score will be. We assume that the evaluation function also recognizes end positions. The score of a won position is equal to $+\infty$, the score of a lost position is equal to $-\infty$, and the score of a drawn position is equal to 0. Assume that the person to move in the root position is called MAX and his opponent is called MIN. It is obvious that MAX will choose a move that maximizes his score, while MIN will choose a move that minimizes MAX's score (i.e., maximizes his own score). The *minimax* algorithm (Von Neumann, 1928) assigns a score to every interior node in a search tree according to this principle. For nodes representing positions where MAX is to move, this score equals the maximum of the scores of its children, whereas the minimum of the scores is taken when MIN is to move. The *principal variation* is a sequence of moves from the root to a terminal node that results in the highest score for the root[1] (the minimax value of the search tree). The principal variation is based on best play for both players (according to the evaluation function used).

We will exemplify the minimax algorithm with Figure 1. White nodes represent White-to-move positions and black nodes represent Black-to-move positions. The values *below* the nodes D, E, F, G and H denote evaluation-function scores. The values to the *right* of the nodes A, B and C denote back-up values. All values are seen from White's point of view. Thus, White wants to maximize and Black wants to minimize the scores.

At node B, Black can choose between two moves, leading to positions D and E, with evaluation-function scores of 6 and 7, respectively. Therefore, Black will choose the move

[1] There may be more than one such path.

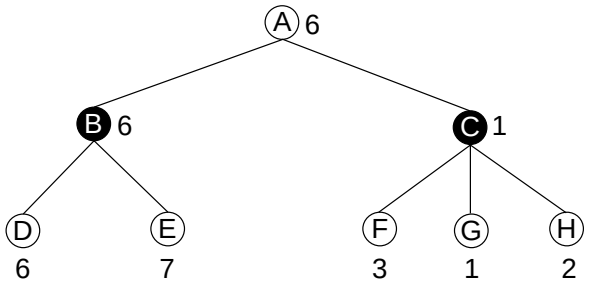


Figure 1: A minimax search tree.

which leads to node D, and node B receives a back-up value of 6. The same reasoning holds for node C, receiving a back-up value of 1. Therefore, at node A (the root node), White will choose the move which leads to node B, and A receives a back-up score of 6. The principal variation leads from node A via node B to node D, along which path all nodes have the value of 6, the minimax value of this tree.

In many of today's game-playing programs a modification of the minimax algorithm is used to find the best move in a position. This algorithm is called the α-β algorithm (Knuth and Moore, 1975). It prunes many irrelevant branches and it therefore is able to search much smaller trees than the minimax algorithm, while still warranted to yield the correct value and move. As an example, in Figure 1 the α-β algorithm would not consider nodes G and H, being irrelevant for the value of node A.

2 Transposition tables

When a human is calculating what move to play, he often encounters the same position again by a *transposition*. For instance, the position in Figure 2 can be reached via different move orders: 1. e4 Nf6 2. Nc3, and 1. Nc3 Nf6 2. e4.

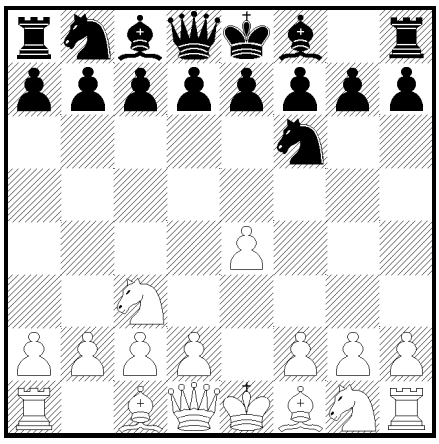


Figure 2: A position that can be reached by different move orders (BTM).

If a transposition occurs, the human knows he has already looked at that position, and skips further analysis of it.

The same reasoning is implemented in game-playing programs. When a position is examined, the results are stored in a large table, the *transposition table* (Greenblatt *et al.*, 1967; Slate and Atkin, 1977). Relevant information for a position in the transposition table includes the score of the position, the best move and the depth of the subtree searched. When a position is encountered, it is looked up in the table. If the position is found there, the information concerned can be used in the search process. In chess, transposition tables are especially useful in positions without Pawns or with locked Pawns because more transpositions are bound to occur in such positions.

As an example, consider problem no. 70 from Fine (1941), shown in Figure 3.

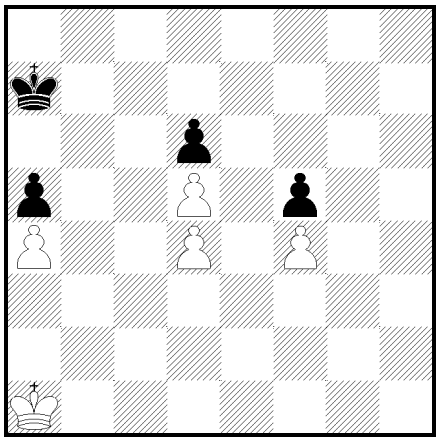


Figure 3: A position with locked Pawns (WTM).

The test program has searched this position *without* and *with* a transposition table for approximately two minutes. In the first case, the program reached a search depth of 13 plies (i.e., 7 moves for White and 6 for Black), whereas in the second case, this depth was reached in only two seconds, while in two minutes a search depth of 24 plies (i.e., 12 moves for White and 12 for Black) was reached! The use of a transposition table completely accounts for this result.

In the ideal case one would preserve every position encountered in a search process, together with all its relevant information. Unfortunately, the memory then required exceeds the available capacity of most present-day computers. Therefore, in practice, a transposition table is usually implemented as a *hash table* (Knuth, 1973). With hashing, a large set (all legal chess positions) is projected onto a small set (the transposition table). The most popular method used by chess programmers is described by Zobrist (1970).

The disadvantage of implementing a transposition table as a hash table is that two different positions can be mapped onto the same entry in the transposition table. This is commonly known as a *collision* (Knuth, 1973). When a collision occurs, a choice has to be made which of the two positions involved should be preserved in the transposition table. Such a choice is based on a *replacement scheme*. Several replacement schemes are discussed in Section 3.

Evidently, the probability of the occurrence of collisions can be lowered by increasing the number of entries in the transposition table.

3 The experiments

The test set used for the experiments discussed in this article consists of 94 positions from six grandmaster games. For more details on the test set, see Breuker *et al.* (1994b) and the forthcoming Ph.D. thesis of the first author.

This article describes four experiments on transposition tables. The number of nodes investigated during a search is used as a measure, including *all* nodes, i.e., interior nodes and leaf nodes.

As mentioned in Section 2 the best move and the score of the position are important parts of the information stored in the transposition table. It is interesting to examine what part of the information gives more benefit: the use of the best move or the use of the score. The effects are examined seperately.

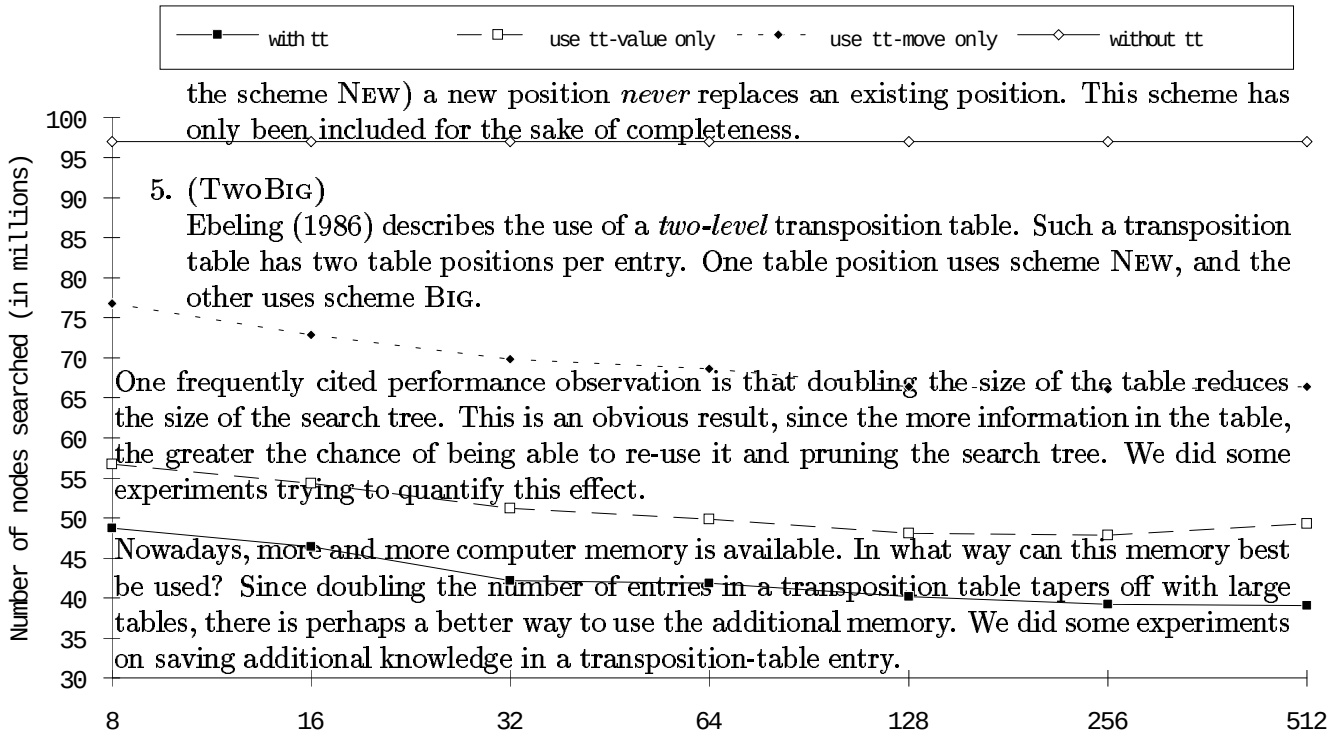
Since the number of examined positions usually is much larger than the number of entries in the transposition table, collisions are bound to occur. When a collision occurs, a choice has to be made which of the two positions involved should be stored in the table. This choice is governed by a replacement scheme. Replacement schemes can be based on several concepts. The following five replacement schemes are dicussed:

1. (DEEP)
   The replacement scheme DEEP is traditional. It is based on the depths of the subtrees examined for the positions involved. At a collision, the position with the *deepest* subtree is preserved in the table (Marsland, 1986; Hyatt *et al.*, 1990). The concept behind this scheme is that for a subtree searched to a greater depth usually more time was invested than for a subtree searched to a shallower depth. Hence, storing the former position in the transposition table potentially saves more work than storing a position less deeply investigated.

2. (BIG)
   Sometimes the depth of the search tree fails to be a good indicator of the amount of search already performed and therefore potentially to be saved. It then may be attractive to select, for retention, the position with the *biggest* subtree rather than the one with the deepest subtree, going by number of nodes rather than by their depths.

3. (NEW)
   The replacement scheme NEW *always* replaces any position in the table when a collision occurs. This concept is based on the observation that most transpositions occur locally, within small subtrees of the global search tree (Ebeling, 1986).

4. (OLD)
   We have also tested the replacement scheme OLD. With this scheme (the opposite of



the scheme NEW) a new position *never* replaces an existing position. This scheme has only been included for the sake of completeness.

5. (TWOBIG)
   Ebeling (1986) describes the use of a *two-level* transposition table. Such a transposition table has two table positions per entry. One table position uses scheme NEW, and the other uses scheme BIG.

One frequently cited performance observation is that doubling the size of the table reduces the size of the search tree. This is an obvious result, since the more information in the table, the greater the chance of being able to re-use it and pruning the search tree. We did some experiments trying to quantify this effect.

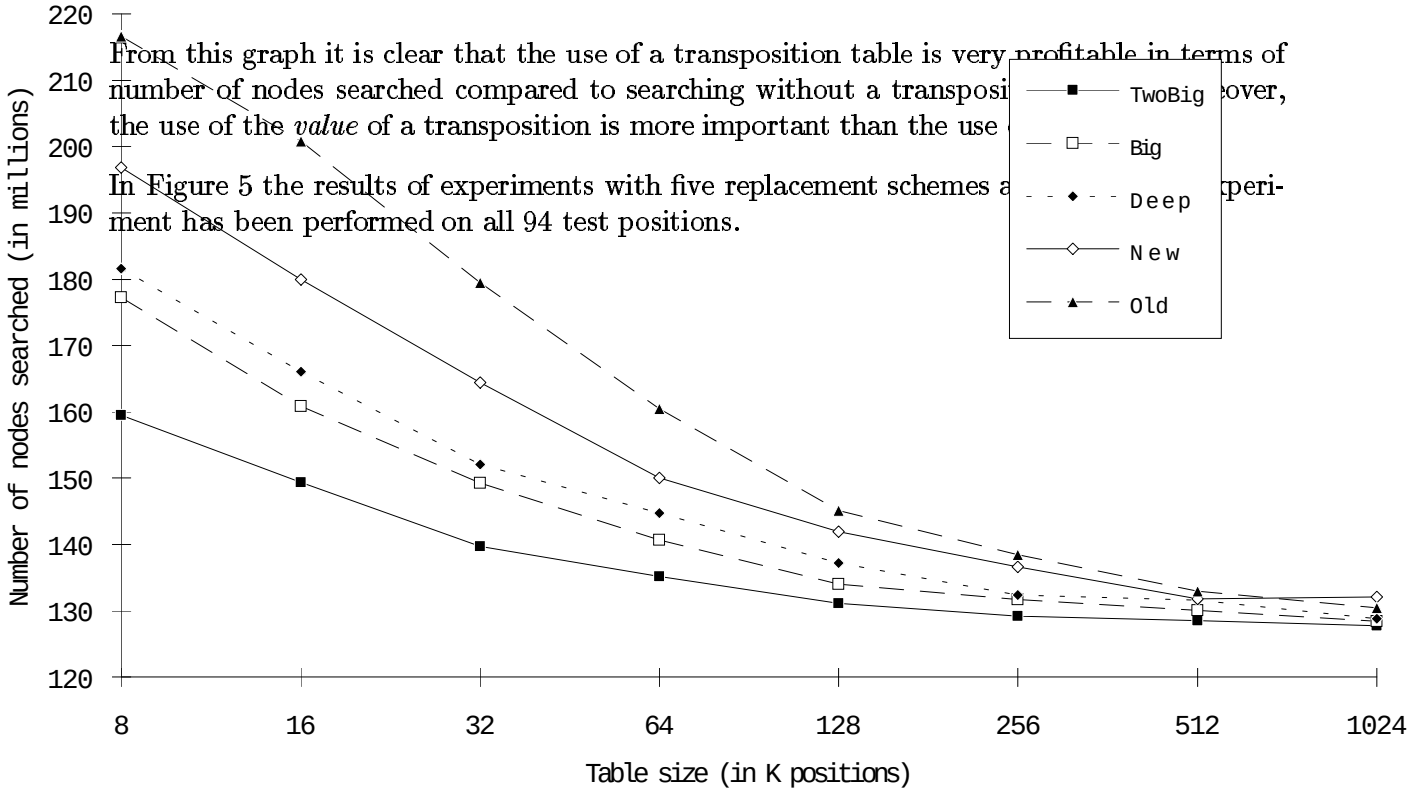
Nowadays, more and more computer memory is available. In what way can this memory best be used? Since doubling the number of entries in a transposition table tapers off with large tables, there is perhaps a better way to use the additional memory. We did some experiments on saving additional knowledge in a transposition-table entry.

4 Results

This Section shows the results of the four experiments described in the previous Section. The results will be illustrated using three graphs.

In the first graph (Figure 4), the results of the use of a transposition table are depicted. The experiment has been performed on a set of 18 test positions taken from a middle-game.

Figure 4: The effect of the use of a transposition table, 7-ply searches.



From this graph it is clear that the use of a transposition table is very profitable in terms of number of nodes searched compared to searching without a transposi ...over, the use of the *value* of a transposition is more important than the use ...

In Figure 5 the results of experiments with five replacement schemes a ...xperiment has been performed on all 94 test positions.

Figure 5: The effect of using different replacement schemes, 7-ply searches.

It is clear that a two-level transposition table is a good idea. Moreover, the scheme most frequently used, scheme DEEP, is *not* the best scheme. For a more detailed description of this experiment and the results, see Breuker *et al.* (1994b).

It is also evident that the shapes of the lines flatten with larger tables. It follows that not much can be won from another doubling of the number of entries in the transposition table. In the range of 8K to 128K, a doubling results in a reduction of number of nodes searched of roughly 3%, whereas in the range of 128K to 1024K, it is less than 1%.

Instead of increasing the table size when more memory is available, it may be better to use the additional memory by storing more information in a transposition-table entry. We have tested the results of storing a 5-ply principal variation in an entry instead of only the best move (a 1-ply principal variation). In Figure 6 the results of these experiments are depicted. The experiment has been performed on a set of 18 test positions taken from a middle-game.

From this graph it follows that with a transposition table of 256K entries, storing a 5-ply variation instead of a 1-ply variation wins roughly 3% which outperforms the 1% gain by simply doubling the table size. More research is needed to substantiate this observation.

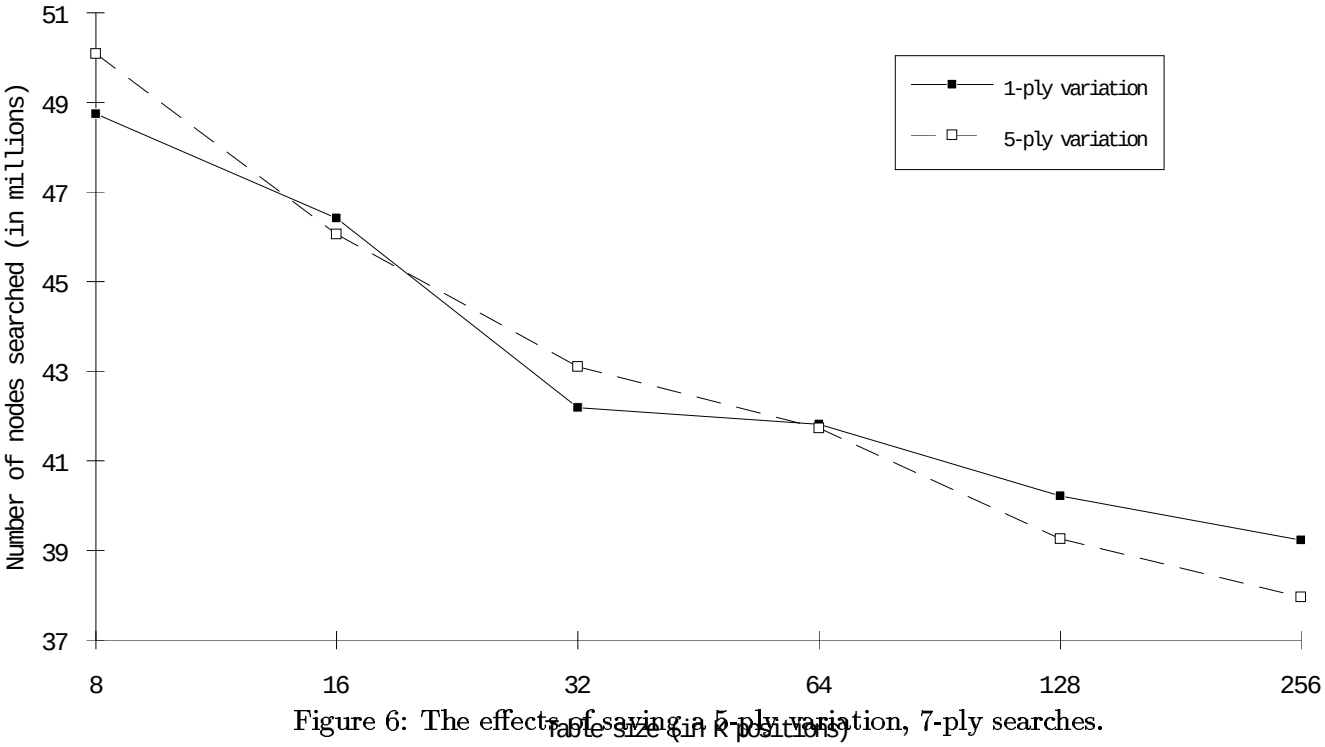


Figure 6: The effects of saving a 5-ply variation, 7-ply searches.

5 Conclusions and future research

In this article, we described four experiments concerning the use of a transposition table in a board game. We have taken chess as the test domain. From the experiments it follows that the use of a transposition table may save a great amount of work, where the value of the transposition is more inportant than the best move in that position.

Further, a replacement scheme based on the number of nodes searched is a better scheme than the traditional one, using the depth of the search as criterion.

The advantage of doubling the transposition-table size is about 3% and tapers off with larger tables. Therefore, it is interesting to experiment with saving additional information in the transposition table when more memory is available.

We have tested the effect of the use of a 5-ply principal variation in a transposition, instead of only the best move. Preliminary results show that a 5-ply variation wins roughly 3%. More experiments are needed to validate this result. For future research several other means of using more memory will also be investigated.

Since it is known that a transposition table works very well in endgames, the results may differ from the middle-game results. Therefore, for future research, it is interesting to investigate the differences for middle-games and endgames for the same experiments.

References

[1] 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.

[2] 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.

[3] Ebeling C. (1986). *All the Right Moves: A VLSI Architecture for Chess.* Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

[4] Feldmann R. (1993). *Game Tree Search on Massively Parallel Systems.* Ph.D. thesis, University of Paderborn.

[5] Fine R. (1941). *Basic Chess Endings.* David McKay Company, New York.

[6] Greenblatt R.D., Eastlake D.E., and Crocker S.D. (1967). The Greenblatt Chess Program. *Proceedings of the AFIPS Fall Joint Computer Conference 31*, pp. 801–810. Reprinted (1988) in *Computer Chess Compendium* (ed. D.N.L. Levy), pp. 56-66. B.T. Batsford Ltd, London.

[7] Hyatt R.M., Gower A.E., and Nelson H.L. (1990). Cray Blitz. *Computers, Chess, and Cognition* (eds. T.A. Marsland and J. Schaeffer), pp. 111–130. Springer-Verlag, New York.

[8] Iida H., Uiterwijk J.W.H.M., and Herik H.J. van den (1994). Thoughts on the Application of Opponent-Model Search. *Advances in Computer Chess 7* (eds. H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk), pp. 61–78. University of Limburg, Maastricht, The Netherlands.

[9] Knuth D.E. and Moore R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326.

[10] Knuth D.E. (1973). *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison-Wesley Publishing Company, Reading, Massachusetts.

[11] Marsland T.A. (1986). A Review of Game-Tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3–19.

[12] Newell A. and Simon H.A. (1972). *Human Problem Solving.* Prentice-Hall, Inc., Englewood Cliffs, NY.

[13] Newell A., Shaw J.C., and Simon H.A. (1958). Chess-Playing Programs and the Problem of Complexity. *IBM Journal of Research and Development*, Vol. 2, pp. 320–335. Reprinted (1988) in *Computer Games I* (ed. D.N.L. Levy), pp. 89-115. Springer Verlag, New York.

[14] Samuel A.L. (1959). Some Studies in Machine Learning using the Game of Checkers. *IBM Journal of Research and Development*, Vol. 3, No. 3, pp. 210–229. Reprinted (1963) in *Computers and Thought* (eds. E.A. Feigenbaum and J. Feldman), pp. 71-105. McGraw-Hill Book Company, New York.

[15] Shannon C.E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Vol. 41, No. 7, pp. 256–275.

[16] Slate J.D. and Atkin L.R. (1977). CHESS 4.5: The Northwestern University Chess Program. *Chess Skill in Man and Machine* (ed. P.W. Frey), pp. 82–118. Springer-Verlag, New York. Second Edition, 1983.

[17] Von Neumann J. and Morgenstern O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton. Second Edition, 1947.

[18] Von Neumann J. (1928). Zur Theorie der Gesellschaftsspiele. *Math. Ann.*, Vol. 100, pp. 295–320. Reprinted (1963) in *John von Neumann Collected Works* (ed. A.H. Taub), vol. VI, pp. 1-26. Pergamon Press, Oxford.

[19] Zobrist A.L. (1970). *A New Hashing Method with Application for Game Playing*. Technical Report #88, Computer Science Department, The University of Wisconsin, Madison, Wisconsin. Reprinted (1990) in *ICCA Journal*, Vol. 13, No. 2, pp. 69-73.