Chapter 2

The transposition table

This chapter is an updated and abridged version of[1]

1. Breuker D.M., Uiterwijk J.W.H.M., and Herik H.J. van den (1994a). Replacement Schemes for Transposition Tables. *ICCA Journal*, Vol. 17, No. 4, pp. 183–193,
2. Breuker D.M. and Uiterwijk J.W.H.M. (1995). Transposition Tables in Computer Chess. *New Approaches to Board Games Research: Asian Origins and Future Perspectives* (ed. A.J. de Voogt), pp. 135–143. International Institute for Asian Studies, Leiden, The Netherlands,
3. 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,
4. Breuker D.M., Uiterwijk J.W.H.M., and Herik H.J. van den (1997b). Information in Transposition Tables. *Advances in Computer Chess 8* (eds. H.J. van den Herik and J.W.H.M. Uiterwijk), pp. 199–211. Universiteit Maastricht, Maastricht, The Netherlands, and
5. Breuker D.M., Uiterwijk J.W.H.M., and Herik H.J. van den (1998b). Solving Domineering. Submitted as journal publication. Also published (1998) as Technical Report CS 98-05, Universiteit Maastricht, Maastricht, The Netherlands.

In this chapter we try to obtain more insight into the first problem statement: which methods exist to improve the efficiency of a transposition table?

In Section 2.1 some important notions and concepts, used throughout the thesis, are introduced. Section 2.2 explains what transpositions are and why it is important to recognize them. The concept behind the transposition table is given in Section 2.3. Section 2.4 lists several data structures suitable for a transposition table. The experimental set-up of our research is given in Section 2.5. Section 2.6 discusses the test domains. Three series of experiments to improve the efficiency of the transposition table are presented in Section 2.7. Section 2.8 provides conclusions.

---

[1] Thanks are due to the Editors of *Advances in Computer Chess 8*, the Editor of *New Approaches to Board Games Research*, and the Editorial Board of the *ICCA Journal* for giving permission to use the contents of the articles in this chapter.

2.1 Notions and concepts

In this section we define several notions and introduce various concepts which we will use throughout the thesis. The main notions are: game tree, search tree and search methods.

Game tree

A *game tree* is a representation of the state space of a game. In the case of a two-player zero-sum game, the game tree is an oriented AND/OR tree. A *node* in the tree represents a position in the game; an *edge* represents a move. A sequence of edges forms a *path* if each edge shares one node in common with the preceding edge, and the other node in common with the succeeding edge. The root of the tree is a representation of the initial position. A *terminal position* is a position where the rules of the game determine whether the result is a win, a draw, or a loss. A *terminal node* represents a terminal position. A node is *expanded* by generating all successors of the position represented by the node. A direct successor of a node is termed a *child* of the node. Analogously, the direct predecessor of a node is termed the *parent* of the node. Nodes having the same parent are termed *siblings*. A node with at least one successor is termed an *interior* node. We note that the root is the only interior node without a parent.

A game tree is generated by expanding all the interior nodes. This process is repeated until all unexpanded nodes are terminal nodes. It follows that the game tree for the initial position is an explicit representation of all possible paths of the game (Pearl, 1984). Zermelo (1912) was the first person stating that every position (not necessarily a terminal position) can be theoretically characterized as a win, a draw, or a loss in the game of chess. The *game-theoretic value* is the value of the initial position, given that both players play optimally. A *minimal game tree* is defined as a minimal part of the game tree necessary to determine the game-theoretic value. The game-theoretic value can, in principle, be determined by examining the complete game tree. Since for most games the game tree (and even a minimal game tree) is extremely large, this is not feasible in practice. For instance, in chess the game tree consists of roughly 10^{43} nodes (Shannon, 1950). Chinchalkar (1996) gives 1.77894×10^{46} as an upper bound and, according to Bonsdorff *et al.* (1978), N. Petrović assumes that the upper bound is approximately 2×10^{43}.

Search tree

When the game tree is too large to be generated completely, a *search tree* is generated instead. This tree is only a part of the game tree. The root represents the position under investigation, and all other nodes of the search tree are generated during the search process. The nodes which do not have children (yet) are termed *leaves*. Leaves include terminal nodes and nodes which are not yet expanded.

The *depth of a node* in a tree is zero for the root, and one plus the depth of its parent otherwise. A node P with a smaller depth than a node Q is an *ancestor* of node Q if node P is on the path from the root to node Q. In this case, node Q is a *descendant*[2] of node P. A *subtree* of a tree is formed by a node together with all its descendants. The *depth of a tree* is equal to the largest depth of all leaves, 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 introduced by Samuel (1959). A path from the root to a leaf is called a *variation*. Leaves are *evaluated* (given a value) with the aid of an evaluation function. A *principal variation* is a sequence of moves where both players play optimally, according to the evaluation function used.

The order in which the nodes of the search tree are generated is defined by the type of search method.

Search methods

Several search methods have been developed. They fall into three categories[3]: (1) depth-first search algorithms, (2) breadth-first search algorithms, and (3) best-first search algorithms.

In *depth-first* search algorithms the root is expanded and one of its *children* is chosen for further investigation. If the node chosen is not a terminal node, the node is expanded and again one of its children is chosen for further investigation. If the child chosen is a terminal node, one of the node's siblings is chosen for further investigation. If all children have been investigated, one of the siblings of the parent is chosen for further investigation, and so on. This process is repeated throughout the whole tree. In summary, the children are expanded *before* the sibling nodes. In Figure 2.1 a search tree of depth three is depicted. For all AND/OR trees/graphs in this thesis white squares represent OR nodes (positions with the first player to move), and black circles represent AND nodes (positions with the second player to move). As an aid to the reader we mention that OR nodes can be seen as playing positions with White to move (WTM), with one or two selected strategies in mind; AND nodes as playing positions with Black to move (BTM), in which case White has to be prepared for *all* countermoves. The numbers represent the order in which the nodes are generated with depth-first search.

An advantage of depth-first search is that it may find a solution rather quickly. However, a disadvantage is that this method often spends much time exploring unfruitful paths. An example of a depth-first search algorithm is $\alpha\beta$ search (Knuth and Moore, 1975).

In *breadth-first* search algorithms, first the node representing the initial state is expanded. Then one of the leaves of the next level is chosen for further investigation. If it is not a terminal node, it is expanded. Thereafter, the next leaf on this level is chosen for further investigation. If all the leaves on this level have been chosen, one of the leaves of the next level is chosen for further investigation. This process is repeated throughout the whole tree. In summary, the children are expanded *after*

---

[2] We note that a parent is a special case of an ancestor, and a child is a special case of a descendant.

[3] Here we split the brute-force search into two categories, effectively creating three categories instead of the two mentioned in Section 1.2.

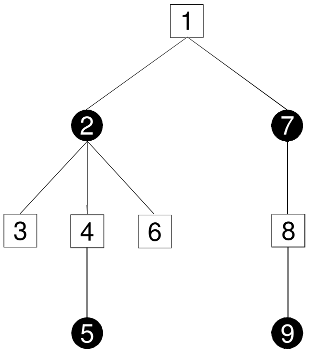

Figure 2.1: A depth-first traversal of a search tree.

the sibling nodes. This is illustrated in Figure 2.2. The numbers indicate the order in which the nodes are generated with breadth-first search.

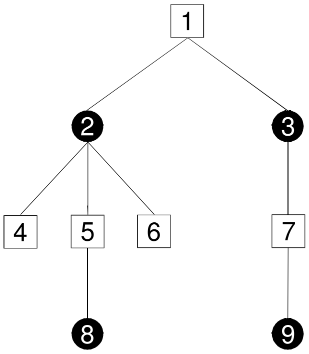

Figure 2.2: A breadth-first traversal of the search tree of Figure 2.1.

An advantage of breadth-first search is that the first solution found will be the solution with the shortest path. A major disadvantage is that it requires a large amount of memory to store all the nodes of the tree: a node is not needed (and does not have to be preserved in memory any more) *after* its subtree is expanded, but since breadth-first search expands the nodes one level after another, all nodes have to be kept in memory. Depth-first search first expands all the children of a node, and therefore a chosen node is not needed (and does not have to be preserved in memory any more) as soon as one of its siblings is chosen for expansion.

Finally, *best-first* search combines the advantages of both depth-first search and breadth-first search. At each step of the search process, the most promising path (according to some criterion) is expanded. Usually what happens is that some depth-

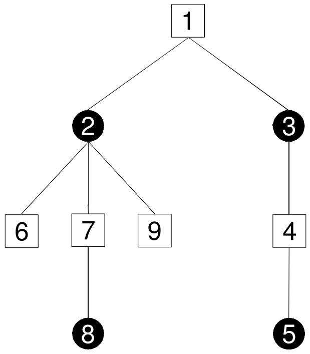

Figure 2.3: A best-first traversal of the search tree of Figure 2.1.

first searching occurs when the most promising branch is explored. Eventually, if the path looks less promising, one of the lower-level branches will be explored. However, search at the old branch is only suspended, and the search can return to it whenever it seems necessary. An example of a best-first search algorithm is proof-number search (Allis *et al.*, 1994). A best-first traversal is depicted in Figure 2.3. The numbers indicate a possible order in which the nodes might be generated.

Plaat (1996) states that the border between best-first search algorithms and depth-first search algorithms is not as clear as shown above. Plaat *et al.* (1996) give a new formulation of the SSS* algorithm (Stockman, 1979), based on the $\alpha\beta$ algorithm. Furthermore, they present a framework, termed MTD(f), that facilitates the construction of several best-first fixed-depth game-tree search algorithms, based on the depth-first minimal-window $\alpha\beta$ search, enhanced with storage.

2.2 Transpositions

When searching for a move, game programs build large search *trees*. Since a position can sometimes be arrived at by several distinct move sequences, the size of the search tree can be reduced considerably if the results of a position previously encountered remain available. The results can be stored in a large direct-access table, called a *transposition table* (Greenblatt *et al.*, 1967; Slate and Atkin, 1977). A closer inspection shows that the search tree then can be considered as a search *graph*, due to the *transpositions*. As an example we provide the chess position of Figure 2.4. It can be reached via the distinct move orders **1. e4 ♘f6 2. ♘c3**, and **1. ♘c3 ♘f6 2. e4**. To complicate matters, the following sequence of seven plies, **1. ♘f3 ♘f6 2. ♘c3 ♘g8 3. e4 ♘f6 4. ♘g1**, also leads to the same position.

Assume that the position of Figure 2.4 appears somewhere in the search tree. After examining the position, a best move is found together with its score, based on a subtree of a certain depth. Since it is possible that this position exists elsewhere in

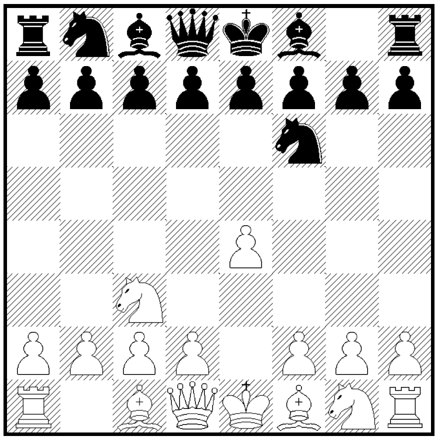

Figure 2.4: A BTM position that can be reached by distinct move orders.

the tree, the relevant information of the position is saved in the transposition table. The relevant information includes the score of the position, the best move, and the depth to which the subtree was searched. Adhering to $\alpha\beta$ search (Knuth and Moore, 1975), the score need not be an exact value, but may be a lower or an upper bound.

Slate and Atkin (1977) already remarked that, for chess, "Strictly speaking, positions reached via different branches are rarely truly identical, because the 50-move and three-time repetition draw rules make the identity of a position dependent on the history of moves leading to that position. This effect is small, and we decided to ignore it." However, ignoring the history of a position can give an incorrect result. This is known as the graph-history-interaction (GHI) problem, of which a solution is presented in Chapter 5. Up to Chapter 5 we concur in ignoring the history of a position.

2.3 A transposition table

2.3.1 Hashing

In the ideal case one would preserve every position encountered in a search process, together with its relevant information[4]. However, the memory required usually exceeds the available capacity of most present-day computers. Therefore, a transposition table is implemented as a finite *hash table* (Knuth, 1973). A position is converted into a sufficiently large number (the *hash value*) by using some hashing

---

[4] In chess, the side to move, castling rights and *en-passant* status are all part of the description of a position.

method. The most popular method used by game programmers is described by Zobrist (1970).

Hashing in chess

In chess there are twelve different pieces (Pawn, Knight, Bishop, Rook, Queen, King for both colours) and 64 squares. For any combination of a piece and a square a random number is generated. In addition, four unique random numbers are generated for castling rights, eight for *en-passant* rights, and one for changing the side to move. Thus, in total 781 ($12 \times 64 + 4 + 8 + 1$) unique numbers are available. The hash value for a position is calculated by doing an exclusive-or (XOR) of the numbers associated with the piece-square combinations of that position. If applicable, the castling and *en-passant* numbers are included too. This way of calculating a hash value has two advantages.

1. The XOR operation is a fast, bitwise operation.

2. The hash value can be updated incrementally. The hash value for a position resulting from some move can simply be obtained by doing an XOR between the hash value of the old position and the two numbers associated with the piece-fromSquare and the piece-toSquare of the move involved[5].

Warnock and Wendroff (1988) implemented in their program LACHEX a hashing-algorithm method used less frequently, based on the theory of error-correcting codes. Their hashing set is constructed from a Bose-Chaudhuri-Hocquenghem (BCH) code (MacWilliams and Sloane, 1977). The only other program we know which uses this method is ZUGZWANG (Feldmann, 1993). The method is not widely used; for details we refer to Warnock and Wendroff (1988).

Hashing in domineering

(For a description of the domineering game we refer to subsection 2.5.2.) For any occupied square on a board a unique random number is generated. (It is irrelevant whether a square is occupied by a vertically or horizontally placed domino.) So for the standard (8×8) board 64 unique numbers are sufficient. No random number for changing the side to move is needed, since it is impossible to have two equal positions with different players to move for the same starting player. The hash value of a position is calculated by doing an XOR of the numbers associated with the occupied squares. The hash value for a position resulting from some move is obtained by doing an XOR between the hash value of the old position and the two numbers associated with the squares of the move involved.

---

[5] One additional XOR is needed for changing the side to move. When capturing, castling or *en passant* is involved, one or a few additional XORs have to be applied.

Hash value and hash key

Figure 2.5 illustrates how the hash value is generally used. If the transposition table consists of 2^n entries, the n low-order bits of the hash value are used as a *hash index.* The remaining bits (the *hash key*) are used to distinguish among different positions mapping onto the same hash index (i.e., the same entry in the transposition table). Therefore, the total number of bits should be sufficiently large (Hyatt *et al.*, 1990). For instance, the chess program CRAY BLITZ uses a 64-bit hash value. For more details, we refer to subsection 2.4.2.

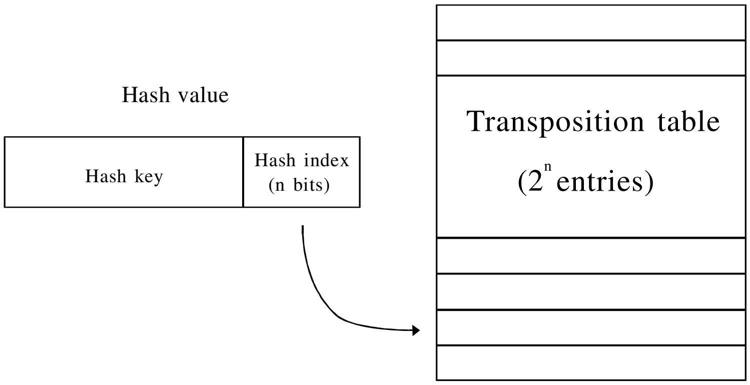

Figure 2.5: The hash value.

2.3.2 The traditional components

For an entry in a transposition table to be effective, it should at least contain the following information (Marsland, 1986; Hyatt *et al.*, 1990):

key [6]: contains the more significant bits of the hash value (see Figure 2.5). The key is used to distinguish among different positions having the same hash index.

move : contains the best move in the position obtained from a search. This is the move which either caused a cut-off, or obtained the highest score. The move is used for the directing knowledge (move ordering).

score : contains the value of the best move in the position obtained from a search. Since we adhere to $\alpha\beta$ search, the score can be an exact value, an upper bound or a lower bound. The score can be used to adjust the α and β bounds of the search.

flag : contains information on the score. The flag indicates whether the score is an exact value, an upper bound, or a lower bound.

---

[6] Marsland (1986) uses the term 'lock'.

depth : contains the relative depth of the subtree searched. When doing an n-ply search from the root and a position is stored at ply m of the tree, the search depth is $n-m$. The depth indicates how deep a previously encountered position has been investigated.

We call a transposition table with these five information fields a *traditional table*.

During the search, each position encountered is looked up in a table. If the position is found, the information stored can be used in three distinct ways, depending on the contents of *flag* and *depth*.

1. The depth still to be searched is less than or equal to the depth retrieved from the table *and* the retrieved value is an exact value. The position does not have to be searched: the search value is retrieved from the table[7]. Usually, the best move is also retrieved from the table, and used for determining the principal variation.

2. The depth still to be searched is less than or equal to the depth retrieved from the table *and* the retrieved value is *not* an exact value. The retrieved value can be used to adjust either the α value (if the retrieved value is a lower bound) or the β value (if the retrieved value is an upper bound). If this causes α to be greater than or equal to β, then a cut-off occurs and the position does not have to be searched. Otherwise, the retrieved move can be used as a first candidate, since it was considered best (or at least good enough to yield a cut-off) previously.

3. The depth still to be searched is greater than the depth retrieved from the table. In this case only the retrieved move is useful[8]. It can be investigated first, since it was considered best for a shallow search, the probability being high that it also will be best for deeper searches. Thus the move is used to improve the directing knowledge (move ordering).

When using iterative deepening (Gillogly, 1972; Slate and Atkin, 1977) and minimal-window search (Pearl, 1980; Marsland and Campbell, 1982; Reinefeld, 1983), transposition tables may significantly reduce the search effort, especially in chess endgame positions with only a few pieces on the board. Nelson (1985) states that “In normal situations the move generator is called only about 35% of the time, the other 65% being handled by the transposition-table move.” Ebeling (1986) concludes that “not using the hash table for moves affects the search size by at least a factor of two.” Hyatt *et al.* (1990) show that “these rules let Cray Blitz find about 30% of typical middle-game positions in the transposition table, and well beyond 90% in certain endgame positions.” Berliner and Ebeling (1990) show that the use

---

[7] If the depth still to be searched is less than the depth retrieved, the search results may differ from the results when searching without a transposition table.

[8] Many heuristics, like aspiration search (Brudno, 1963; Berliner, 1974; Gillogly, 1978), ProbCut (Buro, 1995), and fail-high reductions (Feldmann, 1997) also use the retrieved value. It is used for setting the search window.

of transposition tables combined with good move-ordering heuristics may yield that "on average, the program searches only about 1.4 times the number of nodes that an $\alpha\beta$ search with perfect move ordering would search."

In chess, transposition tables are especially useful in positions without Pawns or with blocked Pawns. As an example, consider problem no. 70 from Fine (1941), shown in Figure 2.6. At first sight, this seems an easy position. However, White has only one winning move, which is the unexpected move **1. ♔b1!**. It is possible to find this move by using knowledge about *distant opposition* (Fine, 1941), or by doing a deep (at least 24 ply) search. Without a transposition table this is not possible in tournament time.

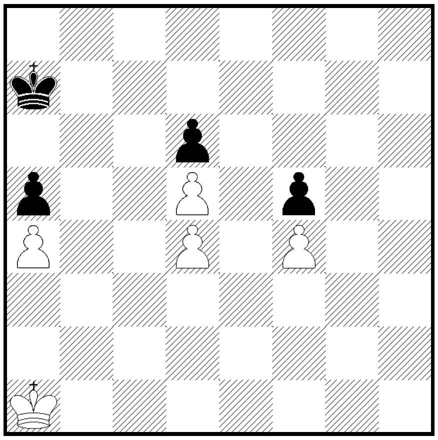

Figure 2.6: A WTM position with blocked Pawns.

For instance, assuming that both sides have five moves on average at their disposal for each position (an underestimation), the minimal game tree when searching 33 ply consists of some 9×10^{11} nodes ($\approx w^{\lfloor (d/2) \rfloor} + w^{\lceil (d/2) \rceil} - 1$ (Knuth and Moore, 1975), with $w = 5$, and $d = 33$). However, Hyatt *et al.* (1984) show that CRAY BLITZ searches only about 4×10^6 nodes when searching this position to a depth of 33 ply (reached in only 65 seconds on a Cray X-MP). The reduction in nodes searched by CRAY BLITZ (a factor of more than 200,000) is caused by the transposition table.

Sometimes transposition tables are used to store information about only a specific part of the position (e.g., the pawn structure, or king safety)[9]. Since this only replaces a part of the evaluation function, not reducing the number of nodes searched, it is outside the scope of our experiments.

---

[9] Warnock and Wendroff (1988) use the name *search tables* when talking about transposition tables in the broadest sense.

2.4 Implementing a transposition table

2.4.1 Data structures

Several data structures come to mind for implementing transposition tables (Pronk, 1987; Van Diepen and Van den Herik, 1987). Two main choices exist.

1. A table with a variable number of positions per entry (array of linked lists). Two advantages of this implementation are (1) the available memory can be divided flexibly among the entries, and (2) no memory is wasted on empty entries. Two major disadvantages are (1) the pointers of the linked list (needed to implement the variable number) take up much memory compared to the size of an entry position, and (2) more computation is needed to check for the existence of a position in a chain of the linked list.

2. A table with a fixed number of positions per entry (two-dimensional array). The advantage of this implementation is that no memory is wasted on extra pointers. The disadvantage is that memory will be wasted when the search is shallow and the table is not filled completely.

The disadvantages of a table with a variable number of positions per entry are more serious than the disadvantage of a table with a fixed number of positions per entry (Van Diepen and Van den Herik, 1987), leading to the logical choice of the latter implementation.

A position which needs to be stored in an entry where all positions are occupied is called an overflow. Overflows can be stored in an overflow area. Two choices for the overflow area exist.

1. The overflow area is implemented as another table or binary tree. Two disadvantages of this implementation are (1) in the overflow area the complete hash value has to be stored in memory, and (2) many comparisons may be needed to find a position in the table.

2. The overflow area is in the same table. The overflows are stored using *double hashing* (Knuth, 1973). An advantage of this implementation is that only one table needs to be used. A disadvantage is that again many comparisons may be needed to find a position in the table. For an extended review of this implementation, we refer to Beal and Smith (1996).

On the matter of implementation we distributed a questionnaire among readers of the *ICCA Journal* and the newsgroup `rec.games.chess.computer` (then called `rec.games.chess`). In the paragraph below we refer to their responses.

The implementation with double hashing is used by, amongst others, Hyatt (1994), Stanback (1994) and Weill (1994). Since using an overflow area may cause more computation to check whether a position exists in the table, a table with one position per entry, not using an overflow area, is used most frequently (Feldmann, 1994; Uiterwijk, 1994; Wendroff, 1994). Distinguishing between two identical positions with different side to move can be done in two ways: (1) use two different transposition tables (one for White and one for Black), or (2) use one transposition table, and use one additional random number for the player to move, which is XORed with the hash value. The latter method is used most frequently (Feldmann, 1994; Hyatt, 1994; Schaeffer, 1994; Uiterwijk, 1994; Weill, 1994; Wendroff, 1994).

2.4.2 Probability of errors

Implementing a transposition table as a hash table introduces two types of error, identified as early as 1970 by Zobrist. The first type of error (*type-1 error*) is the most important one. A type-1 error only occurs when the number of available hash values is much less than the total number of positions in a game, such as in chess. In this case, it can happen that two different positions yield the same hash value. This is a serious error, because when a type-1 error occurs, the information in this entry will be used for the wrong position and, if so, will introduce search errors. One way of detecting this error is to store the whole position in the transposition table. However, in many games this takes up too much space, and is therefore not feasible in practice. Another way of detecting this error is to test the move suggested by that transposition-table entry for legality in the position, effectively lowering the error rate. If the move is illegal, then the table entry must concern another position than the one being investigated. Note that if the move *is* legal, the positions still may differ. The probability of the occurrence of type-1 errors can be lowered by increasing the number of bits in the hash value.

The second type of error (*type-2 error*, or *clash*) occurs when two different positions map onto the same entry in the transposition table, i.e., the positions have equal hash indices, but different hash keys. This is 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 subsection 2.7.1. The probability of the occurrence of collisions can be lowered by increasing the number of bits in the hash index (thus increasing the number of entries in the transposition table).

The probability of a type-1 error and the probability of a collision are both calculated in the same way. The only difference is the number of distinguishable positions (for a type-1 error this is the number of possible hash values[10] and for a collision this is the number of possible hash indices, i.e., table entries[11]).

Let N be the number of distinguishable positions, and M be the number of different positions which have to be stored in the transposition table[12]. The probability that all M positions will have different hash values (i.e., the probability that no

---

[10] i.e., 2^k, where k is the number of bits of the hash value.

[11] i.e., 2^n, where n is the number of bits of the hash index.

[12] This number is equal to the number of non-empty positions in the transposition table after the search has been completed, augmented with the number of collisions during the search.

errors occur) is given by

$$P(\text{no errors}) = (1 - \frac{1}{N}) \times (1 - \frac{2}{N}) \times \cdots \times (1 - \frac{M-1}{N}).$$

If M is small compared to N, then all cross products can be neglected and we have the following approximation

$$P(\text{no errors}) \approx 1 - \frac{1 + 2 + \cdots + M - 1}{N} = 1 - \frac{M(M-1)}{2N}.$$

For small positive x we have $\log(1 - x) \approx -x$, and thus

$$\log P(\text{no errors}) \approx -\frac{M(M-1)}{2N}.$$

Thus, it follows that

$$P(\text{no errors}) \approx e^{-\frac{M(M-1)}{2N}}.$$

If M is sufficiently large, this formula yields

$$P(\text{no errors}) \approx e^{-\frac{M^2}{2N}}. \quad (2.1)$$

This result equals the formula given by Gillogly (1989)[13].

We note that the problem of calculating the probability that at least one error occurs (being $1 - P(\text{no errors})$), is analogous to the problem widely known as the *birthday paradox* (Feller, 1950), where the probability of at least two persons having the same birthday in a group of M persons (N being 365) has to be calculated.

The expected number of errors can be calculated as well. Feldmann (1993) derives the following formula for the expected number of errors (E):

$$E = M - N \times (1 - (\frac{N-1}{N})^M).$$

When N is sufficiently large (which is the case for a transposition table), this formula can be approximated by

$$E \approx M - N \times (1 - e^{-\frac{M}{N}}). \quad (2.2)$$

As an example we consider a program which searches 100,000 nodes per second. If it plays a game using a total of two hours of thinking time, the number of nodes searched is 7.2×10^8. Assume that for about 30% of the nodes, an attempt is made to store them in the transposition table. In the example, this is 216 million nodes. If the hash value consists of 32 bits, the probability of at least one type-1 error is

$$1 - e^{-\frac{216{,}000{,}000^2}{2 \times 2^{32}}}$$

which is very close to 1. So a hash value of 32 bits clearly is too small. If we want to reduce the probability of at least one error to less than 1 percent, Equation 2.1 says that at least 62 bits are required. When using a 64-bit hash value, the probability is reduced to about 1×10^{-3}. In this case, the expected number of type-1 errors for the example above is about 0.05.

---

[13] The article contains a typing error. The probability given here is correct (Gillogly, 1994).

2.5 Experimental set-up

The transposition-table experiments are performed in the domains of chess and domineering. The experimental set-ups for both domains are described in the next two subsections.

2.5.1 The game of chess

For the chess experiments we have developed a test program ALIBABA, being a simple chess program, designed to be easily reproducible by other researchers[14]. This reproducibility serves to promote a uniform platform for research. The major components of ALIBABA constitute the remainder of this section, viz. the search engine, the evaluation function, the move-ordering heuristics, and the transposition table.

The search engine

The search engine is based on a variant of $\alpha\beta$ search: iterative-deepening, minimal-window, principal-variation search[15] (Marsland, 1986). Furthermore, ALIBABA uses *aspiration search* (Brudno, 1963; Berliner, 1974; Gillogly, 1978). At the start of each new iteration, the upper bound and lower bound of the window are set to the value resulting from the previous iteration *plus* and *minus* the value of a Pawn, respectively. If the search fails (the value does not lie within the $\alpha\beta$ window), the window is adjusted to either ($-\infty$, value) when failing low, or (value, $+\infty$) when failing high.

Leaves in the search tree should be "relatively quiescent" when evaluated (Shannon, 1950). Not all leaves are quiescent, so they should be further investigated by a *quiescence search*. In this search only capturing moves and promotion moves are considered, except if the King is in check, when all moves must be searched. We note that in the former case a quiescence search may be terminated early, viz. as soon as it becomes clear that all moves to be generated will be disadvantageous (Schrüfer, 1989). No other search extensions are used in the experiments in order to avoid possible search anomalies.

Before executing the principal-variation search at a node in the search tree, it is checked whether the position represented by the node is a draw by stalemate, by three-fold repetition, or by the 50-move rule, or whether it is a win by checkmate.

---

[14] The full C source code is available by anonymous FTP.
The URL is `ftp://ftp.cs.unimaas.nl/pub/software/breuker/alibaba.tar.Z`

[15] We note that the version of principal-variation search as mentioned by Marsland (1986) is identical to the version of negascout as mentioned by Reinefeld (1989). We use the 1989 reference instead of 1983, which was the first source of this algorithm, since the algorithm described in Reinefeld (1983) contains minor errors.

The evaluation function

The evaluation function used is simple. It consists of a material part and a positional one. The material part counts the difference of material between sides. The positional part is restricted to summing piece-square-table values. During a game, for every type of piece a 64-square table is maintained. Each table contains positional values for that piece on every square on the board. Again, we tried to keep things as simple as possible for the reproducibility. Therefore the positional values are independent of the position at the root. The positional part of the evaluation function is updated incrementally: whenever a move is investigated during the search process, the positional value of the piece-fromSquare table entry is subtracted from it, and the value of the piece-toSquare table entry is added to it. Finally, the evaluation function also serves to detect draws by stalemate, by three-fold repetition and by the 50-move rule as well as checkmate.

The move-ordering heuristics

In any position, AliBaba generates only legal moves, excluding pseudo-legal moves, such as placing or leaving its own King in check. Since the move ordering is important for the efficiency of the $\alpha\beta$ algorithm the following ordering heuristics are implemented.

Refutation tables (Akl and Newborn, 1977). For every move in the root position, the main variation is stored. In the next iteration, moves out of these refutation lines are tried first.

History heuristic (Schaeffer, 1983; Schaeffer, 1989b). A score for every legal move encountered in the search tree is maintained. Every time a move is found to be best in a search, its score is adjusted by an amount proportional to the depth of the subtree investigated. When ordering moves using this heuristic, moves with a higher score are considered before moves with a lower score.

In AliBaba, the moves are ordered in the following way. The first move to be considered is the move from the refutation table (if present). Then, if the position is found in the transposition table (see page 24), the transposition-table move is the next move to be considered. These moves are followed by capture moves (the highest-valued piece to be captured first; if equal, then the lowest-valued capturing piece first). Thereafter follow the promotion moves (ordered by promotion piece; the highest-valued promotion piece first). The remaining moves are ordered according to their descending history-heuristic scores. In addition to the move-ordering heuristics mentioned above, applied immediately after move generation, the root moves are also ordered during the iterative-deepening search processes.

$\alpha\beta$ search combined with a transposition table

Whenever a move is investigated in the $\alpha\beta$ search, the resulting position is looked up in the transposition table. If the position is present, and the depth of the examined subtree is greater than or equal to the depth still to be searched, the information in the table is considered reliable. Therefore, if the score is an exact value, it can immediately be used; otherwise, it can be used to update the window bounds (possibly causing a cut-off). The transposition-table move is always used to order moves (see page 23).

After a position has been investigated to a certain depth, it is stored in the transposition table together with the best move (i.e., the move which caused a cut-off, or the move with the highest score), its score, a flag (denoting whether the score was an exact value, a lower bound, or an upper bound), and the search depth. During quiescence search, a position is never stored in the transposition table.

The results of a transposition-table look-up are used at *all* nodes in the tree. If a leaf position is present in the table, the transposition-table score is used for the evaluation. If the score was an exact value, this score is used as evaluation value. Otherwise, the position is evaluated using the evaluation function. If the evaluation value is higher than the transposition-table score and the bound is an upper bound, the evaluation value becomes equal to the transposition-table score (analogously for the lower-bound score). Since the evaluation function is also used in the quiescence search, the transposition table is used in the quiescence search as well. Note, however, that since positions are only retrieved and not stored during quiescence search, their usefulness is limited during that phase.

In our experiments the transposition table is implemented as a linear array with one or two table positions per entry. No overflow area is used (see also subsection 2.4.1). Furthermore, a 64-bit hash value is used[16]. More details of the implementation of a transposition table in plain $\alpha\beta$ search are given in Marsland (1986).

The pseudo-code (based on Marsland, 1986) for the implementation of a transposition table in plain $\alpha\beta$ search (in a negamax framework) is given in Figure 2.7. Details concerning enhancements, move-ordering techniques and quiescence search are omitted for clarity. The parameters of the function are the current position under investigation (position), the depth to be searched (depth), and the α and β bounds of the search window, respectively. We note that the function Evaluate needs as parameters the position and the transposition-table information. If a leaf position is present in the table, the transposition-table score is used for the evaluation (see above). Furthermore, the function TryToStore attempts to store the search information in the transposition table, using a *replacement scheme* (see Section 2.7.1) when encountering a collision. The function AlphaBeta returns the best value of the position under investigation.

---

[16] In the experiments the size of the transposition table ranges from 8K to 2048K entries. For these transposition-table sizes the hash index ranges from 13 to 21 bits.

```
function AlphaBeta( position, depth, $\alpha$, $\beta$ )
    old$\alpha$ := $\alpha$
    Retrieve( position, ttMove, ttScore, ttFlag, ttDepth )
    /* If the position is not found, ttDepth will be $-1$ and ttMove 0 */
    if ttDepth$\geq$depth then begin
        if ttFlag=ExactValue then return ttScore
        elseif ttFlag=LowerBound then $\alpha$ := max( $\alpha$, ttScore )
        elseif ttFlag=UpperBound then $\beta$ := min( $\beta$, ttScore )
        if $\alpha \geq \beta$ then return ttScore
    end
    if depth=0 then /* Leaf */
        return Evaluate( position, ttScore, ttFlag, ttDepth )
    if ttDepth$\geq$0 then begin /* Examine tt-move first */
        newPos := DoMove( ttMove, position )
        bestValue := $-$AlphaBeta( newPos, depth$-1$, $-\beta$, $-\alpha$ )
        UndoMove( ttMove, newPos )
        bestMove := ttMove
        if bestValue$\geq\beta$ then goto Done
    end
    else bestValue := $-\infty$
    GenerateMoves( moveList, nrMoves )
    if nrMoves=0 then
        return Evaluate( position, ttScore, ttFlag, ttDepth )
    for i:=1 to nrMoves do begin
        if moveList[ i ]$\neq$ttMove then begin
            $\alpha$ := max( bestValue, $\alpha$ )
            newPos := DoMove( moveList[ i ], position )
            value := $-$AlphaBeta( newPos, depth$-1$, $-\beta$, $-\alpha$ )
            UndoMove( moveList[ i ], newPos )
            if value>bestValue then begin
                bestValue := value
                bestMove := moveList[ i ]
                if bestValue$\geq\beta$ then goto Done
            end
        end
    end
Done:
    if bestValue$\leq$old$\alpha$ then ttFlag := UpperBound
    elseif bestValue$\geq\beta$ then ttFlag := LowerBound
    else ttFlag := ExactValue
    TryToStore( position, bestMove, bestValue, ttFlag, depth )
    return bestValue
end /* AlphaBeta */
```

Figure 2.7: The $\alpha\beta$-search function with a transposition table.

2.5.2 The game of domineering

Like chess, domineering is a two-player zero-sum game with perfect information. The game is also known as crosscram, and as dominoes. It was proposed by Göran Andersson around 1973 (Gardner, 1974; Conway, 1976). In domineering the players alternately place a domino[17] (2×1 tile) on a board, i.e., on a finite subset of Cartesian boards of any size or shape. The game is usually played on rectangular boards. The two players are denoted by Vertical and Horizontal. In standard domineering the first player is Vertical, who is only allowed to place its dominoes vertically on the board. Horizontal may play only horizontally. Of course, dominoes are not allowed to overlap. As soon as a player is unable to move the player loses. Although domineering can be played on any board and with Vertical as well as Horizontal to move first, the original game is played on a (8×8) checker-board with Vertical to start, and this instance has generally been adopted as standard domineering. According to West (1996) this size is sufficiently large to be beyond the range of human analysis, and hence the size is fit for an interesting game.

For the domineering replacement-scheme experiments we have developed the program Domi. The search engine is plain $\alpha\beta$ search. The evaluation function is a two-valued function, only returning the values *win* and *loss*.

The move-ordering heuristics

In Domi a distinction is made between (1) the mobility, (2) the number of real moves, and (3) the number of safe moves. Mobility is defined as the number of distinct moves that a player can make in a position. The number of real moves is defined as the maximum number of moves that a player can make in a position, provided that the opponent does not make any move. The number of safe moves is defined as the maximum number of moves that a player can make from a given position in the remaining part of the game, irrespective of the moves that the opponent will make.

The mobility, the number of real moves, and the number of safe moves are updated incrementally. During the search, the decrements δ of the number of real moves and the number of safe moves are continuously updated for both players. The four values are instrumental for a move ordering within the $\alpha\beta$ search, the heuristic being: the higher the ordering value, the better the move likely is. The formula for the ordering value is

$$\begin{aligned}\textit{ordering value} = \; & \delta(\textit{real moves opponent}) - \delta(\textit{real moves player to move}) + \\ & \delta(\textit{safe moves opponent}) - \delta(\textit{safe moves player to move}).\end{aligned}$$

Forward cut-offs

The number of real moves indicates an upper bound of the search-tree depth, and the number of safe moves indicates a lower bound of the search-tree depth. If the number of safe moves of the player to move is greater than or equal to the number of real moves of the opponent after the player has made its move, the move is called

---

[17] The markings on the dominoes are irrelevant.

a *winning* move. In this case, no further moves are generated and the search at this position will be terminated, resulting in a win for the player to move. If the number of safe moves of the opponent is greater than the number of real moves of the player to move after the player to move has made its move, the move is called a *losing* move. In this case, the move is discarded and the next sibling, if any, will be generated.

$\alpha\beta$ search combined with a transposition table

The implementation of the transposition table is similar to the implementation given in Figure 2.7, with two exceptions: (1) `ttFlag` is always equal to `ExactValue` (since only the values *win* and *loss* are used and therefore no bound values are possible), and (2) only the best value and not the best move is stored in the table[18]. All symmetries of the rectangular board are used in DOMI. Whenever a node is investigated in the search, the resulting position is looked up in the transposition table. If it is not present, any of the three symmetrical positions (a horizontal, and/or a vertical reflection) is looked up. In the latter case, if present, the information of the symmetrical position is used[19].

2.6 The test domains

In this section we describe the test domains in which the experiments are performed.

2.6.1 Chess test sets in the literature

Several methods have been used to test the strength of a chess-playing program. In many cases a *test set* is used. Previous test sets mentioned in the literature are:

- the *Win-at-Chess* set of 300 tactical positions from Reinfeld (1958). These positions serve well to test the tactical ability of chess programs, although the strongest programs have outgrown the test (Anantharaman *et al.*, 1988);
- the Bratko-Kopec set of 24 positions (Kopec and Bratko, 1982). These positions are divided into two categories: twelve tactical and twelve positional positions. The positional positions all have a *pawn-lever* move (described by Kmoch, 1959) as their solution. This test suite has two disadvantages: (1) 24 positions are too few, and (2) the test is highly specialized in what it tests;
- a test set consisting of 86 positions, devised by Nielsen (1991). The main purpose of this test set is to estimate the ELO rating (Elo, 1978) of the program;

---

[18] If a position is present in the table, its game-theoretic value (*win* or *loss*) is known and no further search is needed at this point.

[19] We note that we do not make use of rotation symmetry, because that exchanges the concepts of horizontal and vertical.

- a large test set of 5551 positions, described by Lang and Smith (1993). It consists of roughly 2530 tactical positions, 800 positional positions, 2100 endgame positions and 110 opening positions. The test set seems very good, but it will take a long time to run a program on all the test positions. Even if the program is allowed to analyze each position for only three minutes (tournament speed), it will take more than 11 days of computing time. Considering that a programmer needs to test every modification of the program, we have not adopted this test set for our research.

Berliner *et al.* (1991) give a taxonomy of chess positions and have tried to devise a representative test set. Private communication between Lang and Berliner shows that great difficulties were encountered in creating such a set and only some twenty positions have been produced so far (Lang and Smith, 1993).

Finally, it is known (Lang and Smith, 1993) that many commercial companies, such as Fidelity Electronics and Heuristic Software, and many professional programmers, have created their own test sets, but they have rarely published these positions. Most of these tests are devised to test only one aspect of a chess program. Some of these tests are published in computer chess magazines, such as *Computerschaak*, *Modul*, and *ComputerSchach und Spiele*. With the popularity of the Internet nowadays, many more test sets (including the ones mentioned above) are available at several FTP sites. See, for instance, URL `ftp://external.nj.nec.com/pub/wds/`.

As already evident from above, test sets always have a disadvantage: either the number of positions is too small to be representative of positions in high-level chess games, or the number of positions is so large that it will take too much time to test a program on every position. Anantharaman (1991) mentions three other methods to test the strength of a chess program.

1. Play a large number of tournament games. The disadvantage is the time it will take to play a sufficient number of games to obtain a good impression of the strength of the program.

2. Play matches between two computers, starting from a set of chosen positions, playing both sides. This approach has been used by, amongst others, Gillogly (1978) and Schaeffer (1986). According to Anantharaman this will take much time too, because about 1,000 games are necessary to spot a rating difference of ten points[20].

3. Marsland and Rushton (1973) have taken 760 positions from a collection of games between human masters from several strong tournaments. They test the program using all these positions. Conclusions on the strength of a chess program are based on the average rank of the move the human master played. One of the disadvantages is that there is no distinction between minor mistakes

---

[20] This is only important if the versions tested do not differ much in strength. If one version is much stronger (say about 250 points) than the other version, it is not interesting to know whether it is 240, 250, or 260 points stronger.

and major blunders. Another disadvantage is that the possibility that the move played by the program is better than the human move is not taken into account.

Anantharaman (1991) describes another approach to test chess programs. The approach was designed to test search heuristics, but can equally well be applied to test other enhancements of a chess program. The method is used in testing DEEP THOUGHT and its successor DEEP BLUE. The quality of the test program is measured using a *deeper searching* reference program. This reference program is about 300 rating points stronger than the test program. Anantharaman used circa 3,600 positions to evaluate the test program. He concluded that comparing the move chosen by the test program with the move chosen by a human expert is not a reliable method for evaluating the test program. He showed further that comparing the move chosen by the test program with the move chosen by the reference program is a better way for evaluating the test program, correlating well with USCF ratings. Anantharaman reports that with the described technique the same reliability can be reached within only 6% to 16% of the time required when using matches between computers.

2.6.2 Our chess test set

Our testing method for chess differs from the methods discussed above. We have opted to use a sequence of positions derived from actual games as the test set. One advantage is that the chosen positions will not be biased towards tactical issues, but will automatically incorporate positional ones. Moreover, the choice also meets the requirement that successive positions should be related, which is essential when investigating the effects of clearing the transposition table between moves (see subsection 2.7.1). Finally, our goal is not to investigate the strength of the test program, but to investigate the sizes of the search trees involved.

The chess experiments have been divided into two parts. The first part concerns middle-game experiments, and the second part endgame experiments. The middle-game experiments and the endgame experiments are separated to see whether the results are different, since it is known that the benefits from the use of transposition tables are greater in endgame positions than in middle-game positions (Slate and Atkin, 1977).

For the middle-game experiments we have chosen positions from all six Kasparov games of the Euwe memorial VSB tournament 1994 as our test set. Clearly, Kasparov, being the World Champion, is a good player, so his games are of high quality. The opening phase is omitted. We shall only consider middle-game positions, defined as positions from move 15 onwards where both sides have at least 18 points of material[21]. We note that games 1, 2, and 6 terminate when they are still in the middle game according to this definition. Our final restriction is that only positions where Kasparov is to move are investigated[22], resulting in 94 positions as a middle-game test set. The positions are given in Appendix A.

---

[21] Pawn=1, Knight=3.25, Bishop=3.25, Rook=5, Queen=9. Kings do not contribute.

[22] This could be interpreted as a bias in the test positions.

For the endgame experiments we have chosen positions of five games, taken from four instructive endgame books (Fine, 1941; Bouwmeester, 1966; Levenfish and Smyslov, 1971; Averbakh, 1987). An endgame position is defined as a position where at least one side has less than 18 points of material. Only the WTM positions are considered. This results in an endgame test set, consisting of 112 positions. The positions are listed in Appendix B. The test set includes many different types of endgame, such as pawn endgames, bishop endgames, rook endgames and queen endgames. The number of blocked-pawn pairs ranges from zero to four.

2.6.3 The domineering test set

The domineering experiments have been divided into two parts. For the first series of experiments we have taken the empty standard (8×8) board as the test position. Next to the goal of finding the game-theoretic value of the test position, we have set as research goal: deciding which replacement scheme is best.

The second series of experiments concentrates on establishing the game-theoretic value of domineering, played on non-standard boards. We have investigated rectangular board sizes $m \times n$, with m ranging from 2 to 8, and n from m to 9. The variable m denotes the number of rows and the variable n denotes the number of columns of the rectangular board. Contrary to so-called *impartial* games, such as tic-tac-toe, were both players always have the same options, domineering is a game in which the options for both players are not alike. These games are called *partizan*. For partizan games it can matter which player starts the game. In the case of domineering, for square boards (including standard domineering) it is irrelevant whether Vertical or Horizontal starts, but for non-square boards it does matter. We explicitly refrain from the rule that Vertical always starts. Of course an $m \times n$ game started by Horizontal is equivalent to an $n \times m$ game started by Vertical. It thus makes sense to distinguish four possible outcomes for the various domineering games, denoted by '1', '2', 'V', and 'H'. The meanings are as follows:

1: a first-player win, independent of whether Vertical or Horizontal starts;

2: a second-player win, independent of whether Vertical or Horizontal starts;

V: a win for Vertical, independent of whether Vertical plays first or second;

H: a win for Horizontal, independent of whether Horizontal plays first or second.

2.7 Experiments and results

The literature on transposition tables is mainly tutorial in nature (e.g., Marsland, 1986), with only a few detailed discussions of performance (e.g., Ebeling, 1986; Schaeffer, 1989b). One frequently cited performance observation is that doubling the number of positions in the table reduces the size of the search tree. This is an obvious result, since the more information in the table, the greater the probability of finding a transposition. Performance analyses of other aspects of transposition tables, such as which positions to replace, have not, as far as we know, been published in the literature. This section lists three of our experiments concerning transposition tables. In subsection 2.7.1 experiments on using replacement schemes are described. The results have been published before in Breuker *et al.* (1994a), Breuker *et al.* (1996), and Breuker *et al.* (1998b). Subsection 2.7.2 quantifies the merits of using the move information and the score information of the transposition table. In subsection 2.7.3 several ways of using the additional memory are examined. The results of the last two sections have been published before in Breuker and Uiterwijk (1995) and Breuker *et al.* (1997b).

2.7.1 Comparing replacement schemes

The most common implementation of a transposition table is a large hash table. Even though this table is usually made as large as possible, subject to memory constraints, and an overflow area is used, collisions (for which see subsection 2.4.2) are bound to occur. When a collision occurs, a choice has to be made whether to replace or to retain the position in the table. This choice is governed by a *replacement scheme*. From the literature and from discussions with computer-chess practitioners, it appears that the most common form of collision resolution is to prefer the results of deeper searches over shallower ones (Greenblatt *et al.*, 1967; Slate and Atkin, 1977; Marsland, 1986; Hyatt, 1994; Stanback, 1994). This has an intuitive appeal, but has not been supported empirically. This subsection compares the performance of seven collision-resolution schemes, the impact of clearing the transposition table between searches, and the effect of changing the number of positions in the table.

Replacement schemes

Whenever a collision is detected, a choice has to be made whether to replace the existing position in the transposition table. We examine seven different *replacement schemes*, viz. DEEP, NEW, OLD, BIG1, BIGALL, TWODEEP, TWOBIG1. They are based on five concepts, as numbered below.

1. Concept *Deep* (used in scheme DEEP).
   The concept *Deep* is traditional. It is based on the depths of the subtrees examined for the positions involved. In scheme DEEP at a collision, the position with the *deepest* subtree is preserved in the table (Marsland, 1986; Hyatt *et al.*, 1990). The rationale behind this scheme is that a subtree searched to a greater depth usually contains more nodes than a subtree searched to a shallower depth. Therefore, more time was invested in searching the larger tree. Hence, this value, if retrieved from the table, saves more work (i.e., eliminates a larger tree).

2. Concept *New* (used in scheme NEW).
   The concept *New* prefers the last examined position over earlier ones. 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).

3. Concept *Old* (used in scheme OLD).
   The concept *Old* prefers the earliest examined position over later ones. The replacement scheme OLD (the opposite of the scheme NEW) *never* replaces an existing position with a newer position. This scheme has only been included for the sake of completeness.

4. Concept *Big* (used in schemes BIG1 and BIGALL).
   The concept *Big* is based on the number of nodes of a subtree. Sometimes a subtree contains many forcing moves. It also may be potentially well-ordered (in which case many cut-offs have occurred). In such cases, 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. A drawback then is that the number of nodes must be retained as part of each transposition-table entry, reducing the effective number of positions possible for a given amount of storage.

   This concept is used in two schemes: BIG1 and BIGALL. The former counts a table position in a transposition table as a single node, the latter as N nodes, where N is the number of positions searched in order to obtain the information of the table position stored.

5. Concept *Two-level* (used in schemes TWODEEP and TWOBIG1).
   The concept *Two-level* uses a two-level transposition table (Ebeling, 1986; Schaeffer, 1994). Such a transposition table has two table positions per entry[23]. For the scheme TWODEEP the subtree of the first table position is larger than the subtree of the second table position. Upon a collision:

   - if the candidate position has been searched to a depth greater than or equal to the depth of the extant first table position, the first table position is shifted to the second table position, and the candidate position is stored in the first table position;
   - otherwise, the candidate position is stored in the second table position (possibly overwriting an existing position).

   Thus, the candidate position is always stored, and the less important of the remaining two positions (in terms of depth of search) is overwritten. We have also tested the analogous combination of the schemes NEW and BIG1 (further denoted as TWOBIG1).

---

[23] Ebeling (1986) implemented the two-level transposition table in a slightly different way.

We note that in all replacement schemes in our experiments the decision to overwrite an entry does not depend on the type of the score (exact value, lower bound, or upper bound) of the positions involved.

Time stamping

When playing a game, a choice must be made about what to do with the positions stored in the transposition table during the search from a previous position in the game. Successive positions in a game are related to one another, and it therefore may seem best to retain *all* positions in the transposition table[24]. However, these positions are subject to aging, and will be of little use after a few moves in the game. Consequently, clearing the transposition table between searches may also seem attractive, e.g., when the evaluation function between searches is changed.

Instead of physically clearing positions in the transposition table, it may be preferable to time-stamp them after the completion of each search. A time-stamped position remains stored in the table until a collision occurs, when it is unconditionally overwritten. While time-stamped but not overwritten, it will still be used for retrieving information. A position not time-stamped holds information more recent than any previous search.

Table sizes

Undoubtedly, many experiments have been conducted to test the effect of the transposition-table size on the number of nodes investigated. In spite of this, there are few reports in the literature. Ebeling (1986) states: "each doubling in the hash table size yields only a 7% decrease in the search size."

Schaeffer (1994) reported a 5% decrease in the number of nodes searched when doubling the number of positions in the transposition table. It is remarkable that both authors arrive at effects of the same order of magnitude in spite of employing different move-ordering techniques.

We have tested the effect of doubling the number of positions in the transposition tables by conducting the experiments for chess with eight different table sizes, ranging from 8K to 1024K positions and for domineering with four different table sizes, ranging from 256K to 2048K positions, each time doubling the number of positions[25].

The chess experiments

To test the ideas mentioned, the following chess experiments were conducted. The first series of experiments concerned middle-game positions only. It observed the performance of every combination of the seven replacement schemes (with and without

---

[24] We note that if the evaluation function depends on the position at the root of the search tree, search anomalies can occur if the values of positions from a previous search are retrieved from the transposition table. In our chess experiments we did not encounter that problem, since in ALIBABA the evaluation values are independent of the position at the root (cf. page 23).

[25] We use K as an abbreviation for 1024.

time stamping) and the eight table sizes. The middle-game tests have been conducted on 94 middle-game positions, taken from six games between chess experts (see Section 2.6). Each position was searched for 3 to 7 ply. For table sizes of 16K, 64K, 256K and 1024K positions 8-ply searches were performed on 44 middle-game positions taken from the first three games given in Appendix A.

The second series of experiments concerned endgame positions only. It observed the performance of every combination of the seven replacement schemes (only with time stamping) and eight table sizes (ranging from 8K to 1024K positions). These tests have been conducted on 112 endgame positions, taken from five games between chess experts (see Section 2.6). Each position was searched to a depth of 10 ply.

The domineering experiments

In the first series of experiments five replacement schemes have been compared. From the chess experiments it will be evident (as expected) that the scheme OLD is not a good candidate for practical use (cf. page 36), since it uses by far more nodes than all other replacement schemes considered. Therefore, scheme OLD is not considered for the domineering experiments. Further, it will be shown that the differences between schemes BIG1 and BIGALL are marginal in chess (cf. page 36). Therefore, for the domineering experiments we decided to drop scheme BIGALL. Thus, the following five replacement schemes are considered: TWOBIG1, TWODEEP, BIG1, DEEP and NEW. As mentioned before, the experiments are performed with four different transposition-table sizes, ranging from 256K to 2048K positions.

For the second series of experiments we have used the best replacement scheme (TWOBIG1) found from the first series of experiments together with a transposition table of 2048K positions. All boards with $m \neq n$ are investigated twice: (1) with the first player moving vertically, and (2) with the first player moving horizontally.

The performance metric

As the measure for quantifying the search effort in the chess and domineering experiments we use the number of *all* nodes investigated, i.e., the sum of the interior nodes and the leaves. The complete results of all experiments are listed in Appendix C. A few typical results are graphically illustrated in this section. When comparing the replacement schemes for each table size the number of *positions* has been kept constant. This implies that the three BIG schemes (BIG1, BIGALL, TWOBIG1) use slightly more memory than the other schemes because each table position has one additional field (to store the information about the size of the subtree searched). It is claimed that these minor differences do not affect the interpretation of the results. Further, we note that the two-level schemes (TWOBIG1, TWODEEP) have half the number of entries compared to the other five schemes.

The chess middle-game experiments without time stamping

Figure 2.8 shows the middle-game results for the seven replacement schemes using 7-ply searches without time stamping. The graph plots the number of nodes investigated (in millions) as a function of transposition-table size. The number of nodes is the sum of the nodes investigated for the 94 test positions.

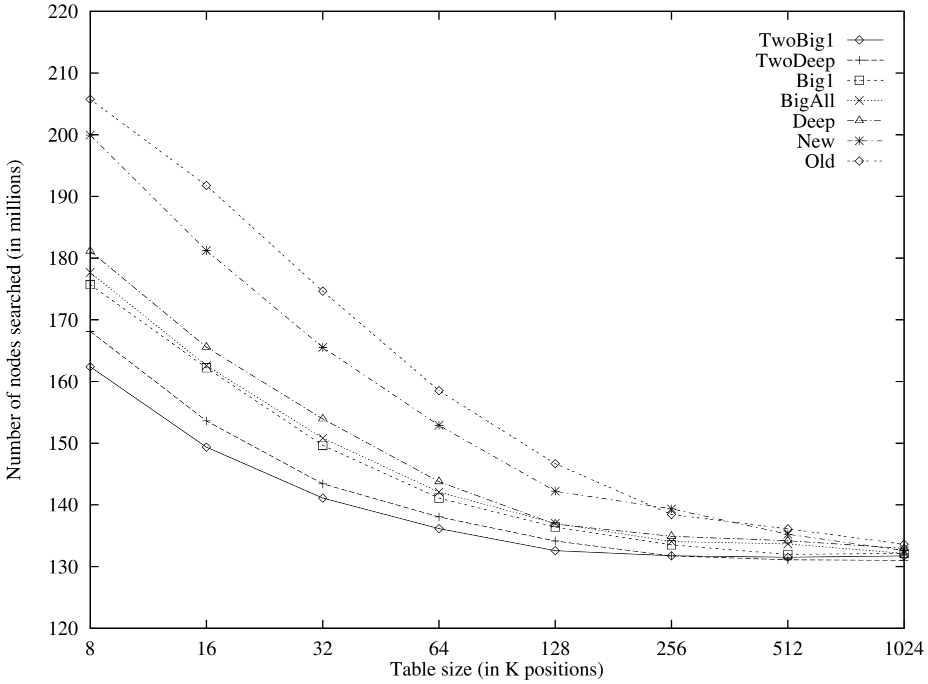

Figure 2.8: Comparing replacement schemes in the chess middle game (without time stamping, 7-ply searches).

The following trends seem to be evident.

- As the table size increases, the number of nodes searched tends to level out to a constant. In other words, at some point, possibly before 1024K in our case, no significant gains may be hoped for by increasing the table size. This is caused by the larger percentage of tree nodes that can be retained in the transposition table: the probability of harmful collisions (i.e., collisions that cost many nodes) then greatly decreases. At a certain point the transposition table is sufficiently big to hold the entire search tree.
- As the table size increases, the spread between replacement schemes shrinks. For table sizes from 512K upwards, the spread is only around 3%, whereas the smallest practicable size, 8K, suggests a spread of no less than 21% between

the best (TwoBig1) and worst (Old) scheme. This is a consequence of the argument above.

- The two-level-table schemes outperform those with one level only. For most data points, TwoBig1 is better than TwoDeep.
- The schemes Old and New are worse than the other three one-level-table schemes. This can be explained by observing that Old and New do not take into account the amount of work done to investigate a position.
- There is hardly any difference between the schemes Big1 and BigAll.
- Our data for small table sizes (8K to 64K) confirm Ebeling's (1986) statement, based on 10 positions, that TwoDeep "reduces search times by 5 to 10% for middle game positions" when compared with Deep.

It is important to observe that the deeper the search performed, the larger the transposition table should be. Beyond 256K positions for a 7-ply search, performance levels off; there is little further to gain. However, some programs can search considerably deeper than 7 ply. They may not have sufficient memory to allow a transposition-table size large enough to reach the point where doubling the number of positions in the table has a limited benefit. The shape of the lines in Figure 2.8 may provide some insight into the effect of transposition-table performance for deeper searches. For example, assuming that searching one ply deeper increases the tree size by a factor of about 4 (Thompson, 1982; Junghanns *et al.*, 1997) a 9-ply search might build a 16 times larger tree than a 7-ply search. The 9-ply results for 256K positions can be approximated by using the 16K ($\frac{256K}{16}$) data point of the 7-ply results. This shows TwoBig1 to be a clear winner.

If we use a 1% reduction in node counts as a criterion for the usefulness of doubling the number of positions in the transposition table, then we obtain from Figure 2.8 for 3, 4, 5, 6, and 7-ply searches in the middle game the following suggested table sizes: $\leq$8K, 16K, 32K, 32K, and 256K positions, respectively.

The chess middle-game experiments with time stamping

The same experiments as above were performed, the only difference being time stamping. This means that each time after a search was completed, the table positions were given a time-stamp, as opposed to clearing the table positions. Thereafter, the next position in the game was searched. Thus the results of a previous search could still be used. Figure 2.9 shows the results of these experiments.

Comparing this figure to Figure 2.8, the following trends seem to be evident.

- The shapes of all graphs are similar in the Figures 2.8 and 2.9.
- The relative order of merit of the replacement schemes seems to be invariant for time stamping; whether one time-stamps or clears the transposition tables between moves, TwoBig1 appears to have a persistent edge.

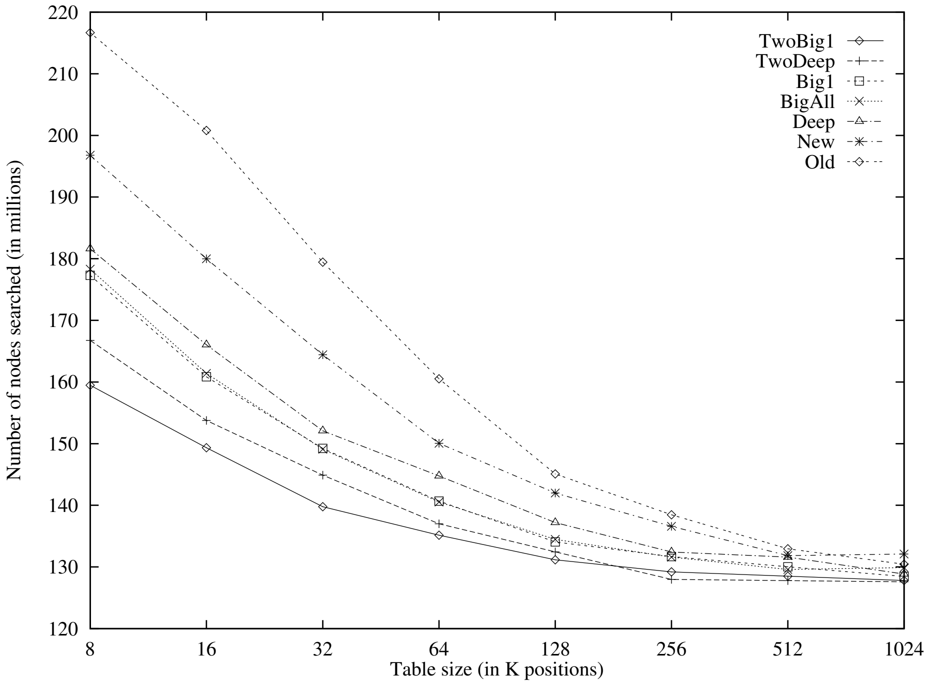

Figure 2.9: Comparing replacement schemes in the chess middle game (with time stamping, 7-ply searches).

Time stamping has a slight performance benefit. The savings with time stamping are some 2%. Therefore, it can be recommended since it only requires one additional bit per table position and requires little additional computation.

As mentioned on page 33, 8-ply searches have been performed on middle-game positions for table sizes of 16K, 64K, 256K and 1024K positions, again with and without time stamping. The results are given in Appendix C. Assuming a ratio of four in search size between subsequent ply depths, the 7-ply results for table sizes of 16K, 64K and 256K positions should be scalable to the 8-ply results for table sizes of 64K, 256K and 1024K positions, respectively. Inspection of the results verifies this. In other words, the 7-ply search conclusions given above are confirmed by the 8-ply search results, in particular the conclusion that the two-level schemes outperform those with one level.

The benefit of a transposition table in chess middle games

Figure 2.10 shows the relation between the benefit of using a transposition table and the search depth for all 94 middle-game positions. The data are shown for a transposition table of 1024K positions and replacement scheme TwoBig1, using time stamping. The search size without a transposition table is 1.

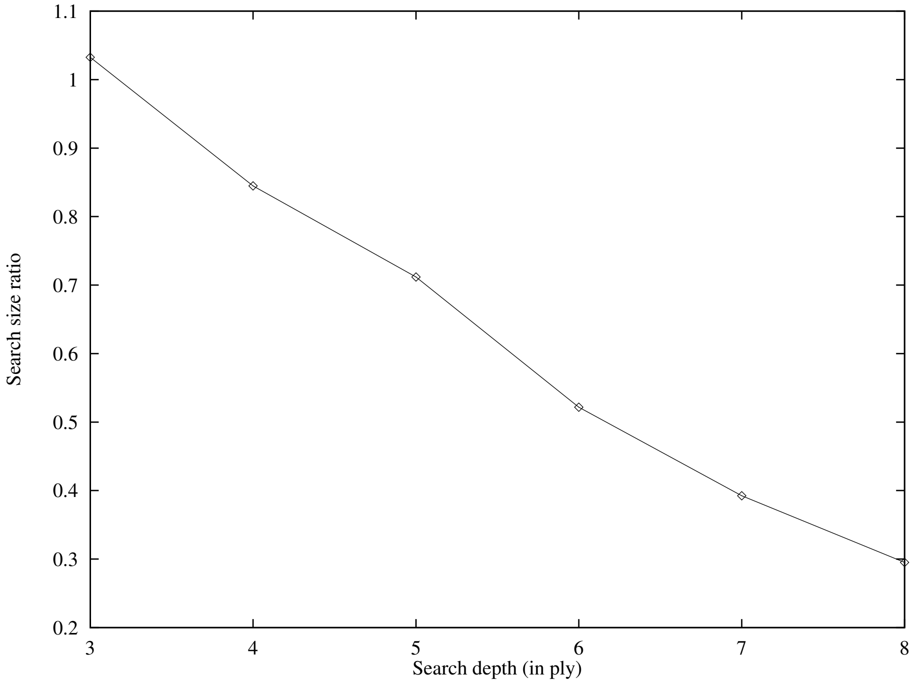

Figure 2.10: Using a transposition table in the chess middle game (with time stamping, scheme TwoBig1, 1024K positions).

For this example we see that, limiting ourselves to a 3-ply search in middle-game positions, the use of a transposition table with time stamping is even counterproductive in that it prolongs the search. The probable cause is an unfavourable move ordering, caused by a poor best-move suggestion from the transposition table. However, it is reassuring that the use of transposition tables is definitely advantageous at more realistic search depths of over 3 ply.

Ebeling (1986) concludes that "not using the hash table for moves affects the search size by at least a factor of two." The graph confirms this factor for searches of 6 ply and deeper. It is noted that transposition tables reduce the search considerably in many other domains, such as domineering (cf. page 40) and also in single-agent-search problems, such as sokoban (Junghanns and Schaeffer, 1997).

The chess endgame experiments with time stamping

Figure 2.11 shows the endgame results for the seven replacement schemes using 10-ply searches with time stamping. The graph plots the number of nodes investigated (in millions) as a function of transposition-table size. The number of nodes is the sum of the nodes investigated for the 112 test positions.

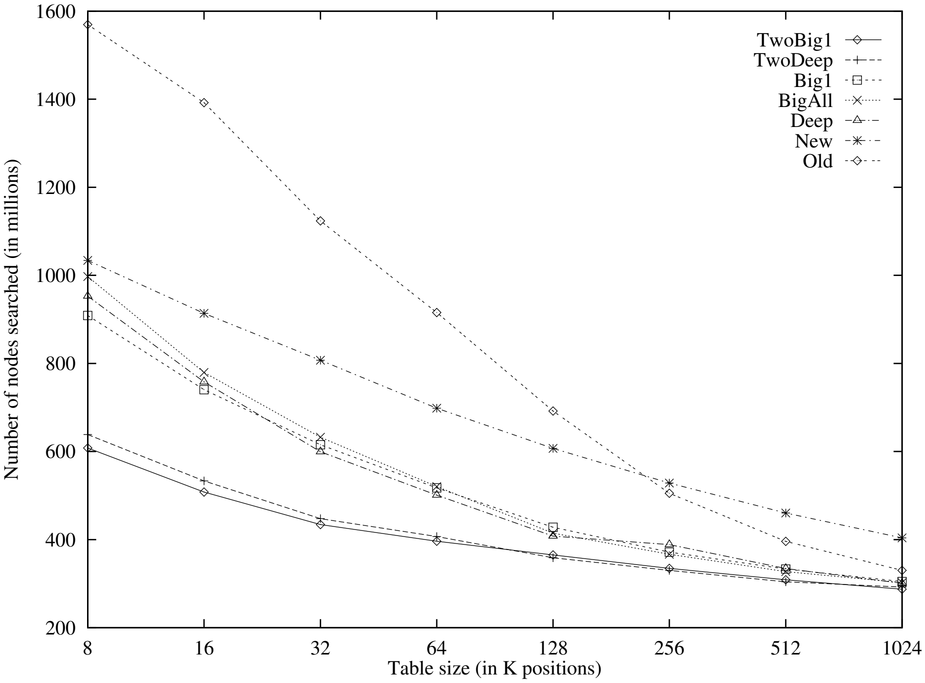

Figure 2.11: Comparing replacement schemes in the chess endgame (with time stamping, 10-ply searches).

From this graph it follows that the conclusions given for the middle-game experiment also hold for the endgame, with one exception. In middle-game positions it is clear that the concept Big works better than the concept Deep: schemes Big1 and BigAll search fewer nodes than scheme Deep. and scheme TwoBig1 fewer nodes than scheme TwoDeep. The difference between the two concepts has disappeared in the endgame. This is explained as follows. If a subtree contains many forcing moves or is well-ordered, cut-offs occur. Since in the middle game the mobility of each player is higher than in the endgame, such pruning will on average cause larger savings in middle-game positions than in endgame positions. Therefore, the size of search trees of equal depth will vary more in middle-game positions than in endgame positions. The concept Deep does not have a preference for any of two such subtrees, whereas the concept Big has a preference for the largest subtree. Thus, in the middle game the size (as compared to the depth) of the search tree investigated will be a better characteristic measuring the work performed than it is in the endgame.

If we again use a 1% reduction in node counts as a criterion for the usefulness of doubling the number of positions in the transposition table, then we obtain for 3, 4, 5, 6, 7, 8, 9, and 10-ply searches in the endgame the following suggested table sizes: ≤8K, ≤8K, ≤8K, 32K, 64K, 512K, ≥1024K, and ≥1024K positions, respectively.

Solving domineering

From preliminary experiments it was obvious that standard 8×8 domineering could not be solved in a reasonable amount of time without using a transposition table (Fotland, 1997). Using a transposition table, we solved the game. It appeared to be a first-player win. Later on, we were informed that this result was independently found by Morita (1997).

In Figure 2.12 the results for the five replacement schemes in domineering are given. Detailed results are listed in Appendix C. The graph plots the number of nodes investigated (in millions) to solve the standard game as a function of transposition-table size.

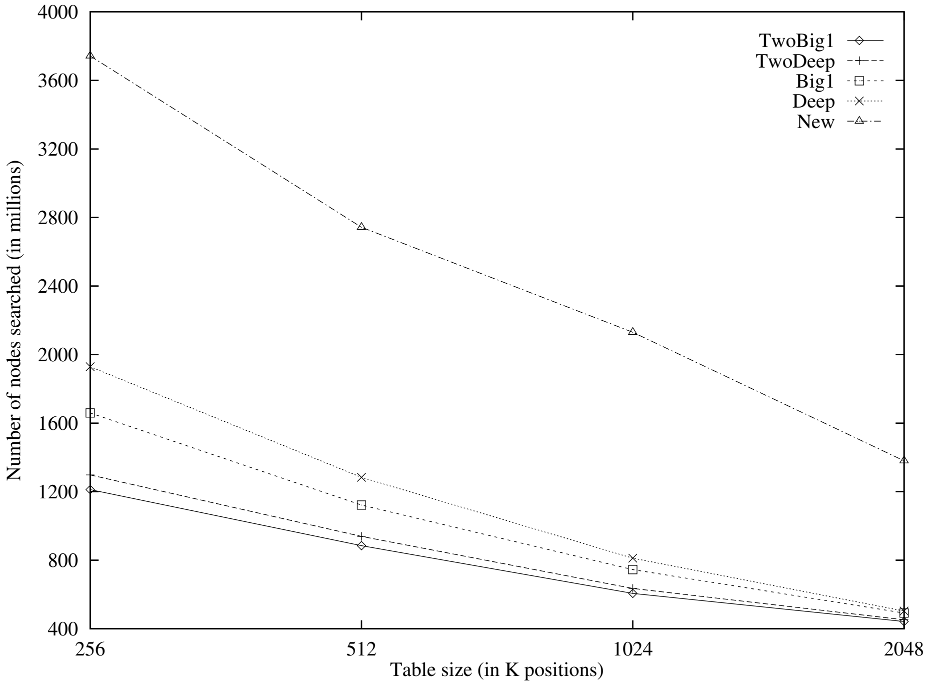

Figure 2.12: Comparing replacement schemes in domineering.

It is noted that the conclusions from the domain of chess also hold in the domain of domineering and are even more pronounced: two-level replacement schemes work much better than one-level schemes. Furthermore, the concept *Big* shows more improvement over the concept *Deep* than in chess.

Solving domineering for non-standard boards

Table 2.1 gives the results for the second series of experiments. The numbers indicate the real number of nodes investigated. The scheme used is TwoBig1 with 2048K positions. In the first column the board size is depicted. The second column gives the game-theoretic value, with '1', '2', 'V', and 'H' as defined in subsection 2.6.3.

Size	Res	Nodes
2×2	1	1
2×3	1	2
2×4	H	13
2×5	V	15
2×6	1	14
2×7	1	17
2×8	H	67
2×9	V	126
3×3	H	1
3×4	H	10
3×5	H	19
3×6	H	40
3×7	H	77
3×8	H	74
3×9	H	99
4×4	1	40
4×5	V	87
4×6	1	1,327
4×7	V	1,984
4×8	H	12,024
4×9	V	45,314
5×5	2	604
5×6	H	1,500
5×7	H	13,584
5×8	H	30,348
5×9	H	177,324
6×6	1	17,232
6×7	V	302,259
6×8	H	3,362,436
6×9	V	18,421,911
7×7	1	408,260
7×8	H	12,339,876
7×9	H	320,589,295
8×8	1	441,990,070
8×9	V	70,918,073,509

Table 2.1: Game-theoretic results of domineering for various board sizes.

Our results fully agree with the results published earlier by Berlekamp and coworkers as far as investigated by them (see Berlekamp *et al.*, 1982b; Berlekamp, 1988; Guy, 1991). They provide complete analyses for boards with a size of $2\times n$ $(2\leq n\leq 7)$, $3\times n$ $(3\leq n\leq 5)$, and 5×5. We remark that games with a game-theoretic value '1', '2', 'H' and 'V' match their characterizations of fuzzy, zero, positive and negative games, respectively. By using a straightforward $\alpha\beta$ algorithm, returning only whether a position is a win or a loss, we did not keep track by what difference a position is won or lost. Hence, it is impossible to provide a detailed comparison with their analyses.

Another subset of our results coincide with the results obtained previously by Fotland (1997), who did his investigations several years ago. Fotland also used a straightforward $\alpha\beta$ algorithm plus a large transposition table. He did not solve the 8×8, and the $m\times 9$ $(5\leq m\leq 8)$ boards. Our program Domi never investigated more nodes than Fotland's program; Domi has a more efficient node investigation than Fotland's program by a ratio of up to 10 for the larger boards.

In Table 2.1 we may discern several patterns of exponential growth with the board size, e.g., the $n\times n$ series, the $m\times n$ series with fixed m, *etc.* The results suggest that the ratio always grows exponentially with the board size. Since the 8×9 board took more than 600 hours to be solved, we did not investigate the 9×9 board. It is interesting to note that of all boards considered the 5×5 board is the only one in which the second player wins.

2.7.2 Quantifying the merits of move and score

Although it is evident that the use of a transposition table reduces the search effort, two open questions still exist. First, how big is the overall reduction? And second, which information has the largest impact on the reduction? This subsection consists of two parts. The first part compares storing the best *move* with storing the *value* of the best move. The second part compares storing the *bound* values for minimal-window search with storing the *exact* values[26].

From the components mentioned in subsection 2.3.2 it follows that a transposition table is used for two reasons: (1) the score is used for establishing the value of the position, and (2) the retrieved move is used for move ordering. In the first case the value is either an exact value, and this position does not have to be re-searched, or a bound value, in which case either the α value or the β value might be adjusted[27].

We have investigated the merits of these individual components in order to obtain more insight into the way a transposition table helps to reduce the search effort. This information may help in devising more efficient transposition-table schemes and may deliver guidelines about what additional information can be useful. For investigating the merits of *move* and *score* we have performed six experiments.

1. Search without a transposition table.
2. Search with a traditional transposition table, without *score*.
3. Search with a traditional transposition table, without *move*.
4. Search with a traditional transposition table, without *move*, only storing and using the score information if the score is an *exact value*.
5. Search with a traditional transposition table, without *move*, only storing and using the score information if the score is a *bound value*.
6. Search with a traditional transposition table, with *move* and *score*, storing and using the score information both if the score is an *exact value* or a *bound value* (i.e., use the transposition table fully).

The experiments 1 and 6 are performed to obtain upper and lower bounds.

Results of the merits of move and score

As the measure for quantifying the search effort we use the number of *all* nodes investigated, i.e., the sum of interior nodes and leaves. The test set used for the experiments consists of 18 consecutive WTM middle-game positions taken from the game Kasparov-Short, Amsterdam 1994, and 21 consecutive WTM endgame positions taken from the game Rabinovich-Romanovsky, Leningrad 1934 (see Appendix A and

---

[26] We note that the experiments are only performed in the chess domain, since in the domineering experiments no moves and no bound values are stored.

[27] Obviously, when the depth still to be searched is greater than the depth in the transposition table, the score from the transposition table is not used.

B). Both games were played by human experts. The 18 middle-game positions have been searched to a depth of 8 ply, and the 21 endgame positions to a depth of 10 ply. The replacement scheme used for all experiments is TwoBig1, the scheme which performs best (see subsection 2.7.1). All experiments have been performed with a series of transposition tables, ranging from 8K positions to 256K positions, since beyond 256K positions there is little further to gain, as is shown in subsection 2.7.1. Time stamping (see page 33) is used. The complete results can be found in Appendix C. The number of nodes are the cumulative results of all 18 and 21 positions, respectively. The merits of the best move and its score stored in a transposition-table entry have been examined separately.

Middle-game experiments

In Figure 2.13 the results of the use of a traditional transposition table for the middle-game positions are depicted. The figure shows the number of nodes investigated as a function of the transposition-table size. The numbers in the legend refer to the experiments mentioned on page 42.

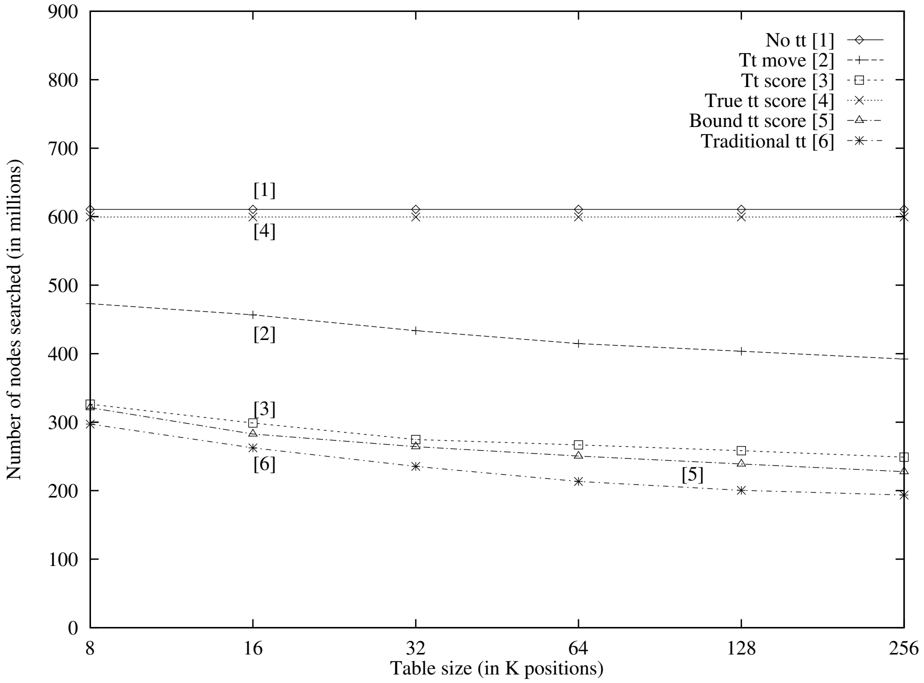

Figure 2.13: Comparing move and score in the chess middle game (8-ply searches).

Figure 2.13 clearly shows that the use of a transposition table (experiment 6) is very profitable in terms of number of nodes searched compared to searching without a transposition table (experiment 1); a result which was already evident from the results of subsection 2.7.1. Further, using the field *score* of a transposition table (experiment 3) is more important than using the field *move* (experiment 2)[28]. This is caused by the minimal-window search: whenever one of the bounds of the minimal window is updated, its lower bound will be greater than its upper bound, thereby causing a cut-off. Experiments show that whenever a position is found in the transposition table, the retrieved value causes a cut-off in about 50% of the cases[29]. However, this effect stems fully from bound values (experiment 5). Exact values (experiment 4) hardly have any effect in this respect. Upon closer investigation it becomes clear that exact values are used only a few times. Typically, an exact value is encountered tens of times in the transposition table, while a bound value is encountered tens of thousands of times[30].

Endgame experiments

The results of the experiments on the endgame positions are analogous to the results of the experiments on the middle-game positions, but they are more pronounced, as can be seen in Figure 2.14. Moreover, the use of a transposition table is more profitable in endgames than in middle games. We see that the largest (256K positions) transposition table used in middle games with only the field *move* (experiment 2) results in about a 36% node decrease, whereas in the endgame the decrease is about 65%. If in addition *score* is used (experiment 6), a total decrease of about 68% in the middle game and about 89% in the endgame is obtained.

2.7.3 Using additional memory

A *collision* (Knuth, 1973) occurs when two different board positions map onto the same entry in the transposition table (i.e., they have an equal hash index, but a different hash value). Regardless of whether the old entry is replaced by the new one, collisions will have a negative effect on the efficiency of a transposition table, since one of the two positions will not be present in the table. The probability of the occurrence of collisions can be lowered by increasing (doubling) the number of positions in the transposition table. However, at a certain point the doubling is not profitable any more (cf. subsection 2.7.1). This subsection looks at other ways to use additional memory, by comparing the use of more information per entry position with the use of more positions in the table.

---

[28] We note that the results of the experiments depend on the move-ordering mechanism used (for which see page 23).

[29] The minimal window causes the retrieved value to be either a fail low, or a fail high.

[30] All nodes (except nodes on the principal variation and fail-high nodes) are searched with a minimal window. Therefore, no exact value is known for these nodes.

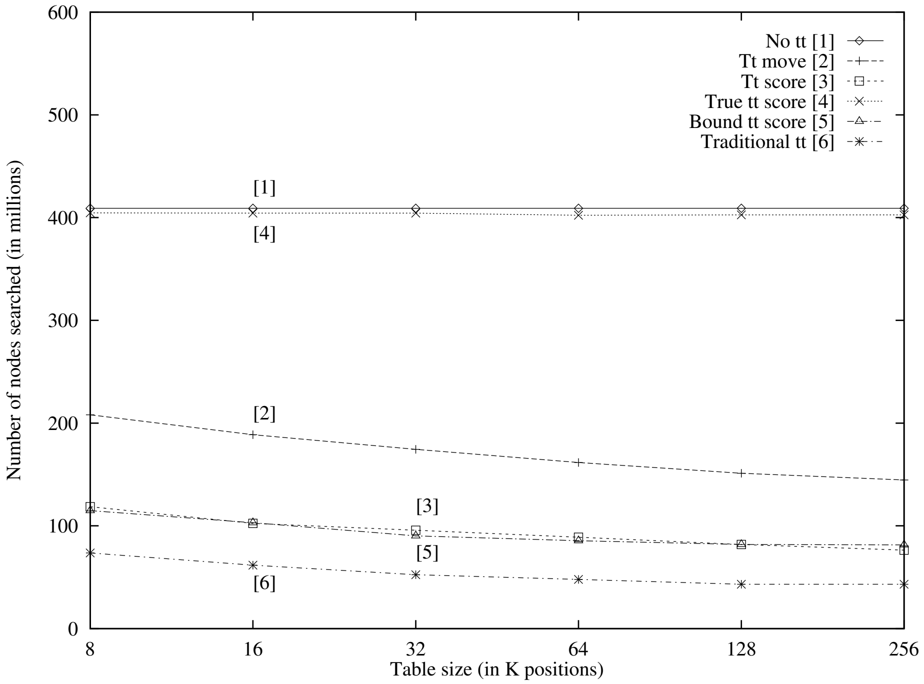

Figure 2.14: Comparing move and score in the chess endgame (10-ply searches).

Additional components

In our search for storing additional information in a transposition-table entry we have found several suggestions, amongst others made by Schaeffer (1994), Stanback (1994), and Thompson (1996a). From their suggestions, we mention six additional components.

date : contains the root's ply number in the game at the time when the position was stored[31]. Sometimes only a 1-bit date flag is used, stating whether the position is from an 'old' search or not. The date is used for time stamping. A position will be overwritten by a position with a newer date.

depth : contains the number of ply seen from the root. A position is more important if it is nearer to the root, since there it has a higher probability of being re-searched; possible savings are then most likely larger than savings for positions deeper in the tree.

---

[31] Feldmann (1996) defines *date* as the number of conversion moves (irreversible moves) made in the game.

extension : contains a Boolean value, denoting if a search extension was done at this position. The extension criteria of a node may vary (e.g., because the extension is dependent on the $\alpha\beta$ window), resulting in an extension one time and *not* in an extension the other time. The Boolean extension helps to overcome this problem (which is especially important when doing a re-search).

principal : contains a Boolean value, denoting if this position is part of the principal variation of a child of the root[32]. Positions which are part of the principal variation of a root's child are important positions, and may not be overwritten by other positions.

draw : contains a Boolean value, denoting if the backed-up score of this position is a proved draw. This is useful for distinguishing between variations resulting in positions which are real draws, and variations resulting in balanced positions (which obtain a draw value).

additional bound : instead of storing only a lower bound or an upper bound of the score, both bounds can be stored in an entry, with separate search depths for each. This is done by Truscott (1981) in the program DUCHESS.

Presumably, the information contained in these six components will have an impact on the number of nodes searched. However, only very few researchers have published even provisional results about experiments on these additional components. In subsection 2.7.1 we mentioned an experiment testing the use of a 1-bit date flag (time stamping), concluding that time stamping has a slight edge. In general it seems that adding these new components to an entry is not very profitable (Schaeffer, 1996b).

Storing the additional information described above does not take up much memory. Most fields need one bit of storage only, since they are Booleans. The choice for small additional components is made on purpose, since a larger entry results in a transposition table with fewer entries (assuming the same amount of memory is available). However, once a critical transposition-table size has been reached not much is to be gained from doubling the number of positions. Moreover, if the available memory is less than the memory needed for doubling the number of positions in the table, it still can be used for storing more information in an entry.

The above considerations have led to the question of how to use additional fields, taking up more memory than only one bit. Instead of storing the best move (which can be seen as a 1-ply principal variation) in a transposition-table entry, it may be interesting to investigate the effects of storing a deeper *principal variation* in an entry (Schaeffer, 1996b). This principal variation (PV) can be used to guide the search. If a position is not present in the transposition table, a good move may still be available from the n-ply PV information of an ancestor position.

---

[32] Note that this is a way to implement the refutation table using the transposition table.

Additional memory

Below we describe a limited set of experiments investigating the effects of storing an n-ply PV in a transposition-table entry[33]. The PV information is used as follows. If a position is found in the transposition table, the corresponding PV is retrieved from the table. The first move in the PV is used for move ordering and the remainder of the PV is used in further search. If a position is not found in the transposition table, and a good move is available from the PV of an ancestor position, then this move is used for move ordering.

The conditions for the experiments are the same as the conditions mentioned in subsection 2.7.2. Again, the number of nodes in the Figures 2.15 and 2.16 are the cumulative results of all 18 and 21 positions, respectively. We have tested the results of storing an n-ply PV ($n = 2...5$) in an entry versus storing only the best move (a 1-ply PV). The complete results of the experiments are presented in tabular form in Appendix C.

Middle-game experiments

In Figure 2.15 the results of the PV experiments on middle-game positions are depicted. The number of nodes investigated are shown as a function of the transposition-table size.

Our first observation is that storing an n-ply PV seems hardly worthwhile: the effects are small and severely dependent on the size of the transposition table. The explanation for this is that for less than 0.1% of the nodes investigated a position appears to be absent in the transposition table, whereas a PV from an ancestor still is available. To give some quantification, it can be seen that with the largest transposition table (256K positions), storing a 5-ply PV instead of a 1-ply PV wins roughly 5%, outperforming the 1% gain by simply doubling the number of positions in the table to 512K (see subsection 2.7.1).

Endgame experiments

The results of the experiments on the endgame positions are analogous to the results of the experiments on the middle-game positions, as can be seen in Figure 2.16. Here again, for the largest transposition-table size, the 5-ply PV outperforms the 1-ply PV, this time by some 12%.

2.8 Chapter conclusions

This chapter has shown that a transposition table (memorizing the outcome of positions previously analyzed in games, such as chess and domineering) is a useful technique. The technique has enabled us to solve a large number of different-sized

---

[33] We note that these experiments are solely performed in the chess domain, since no moves are stored in the domineering experiments.

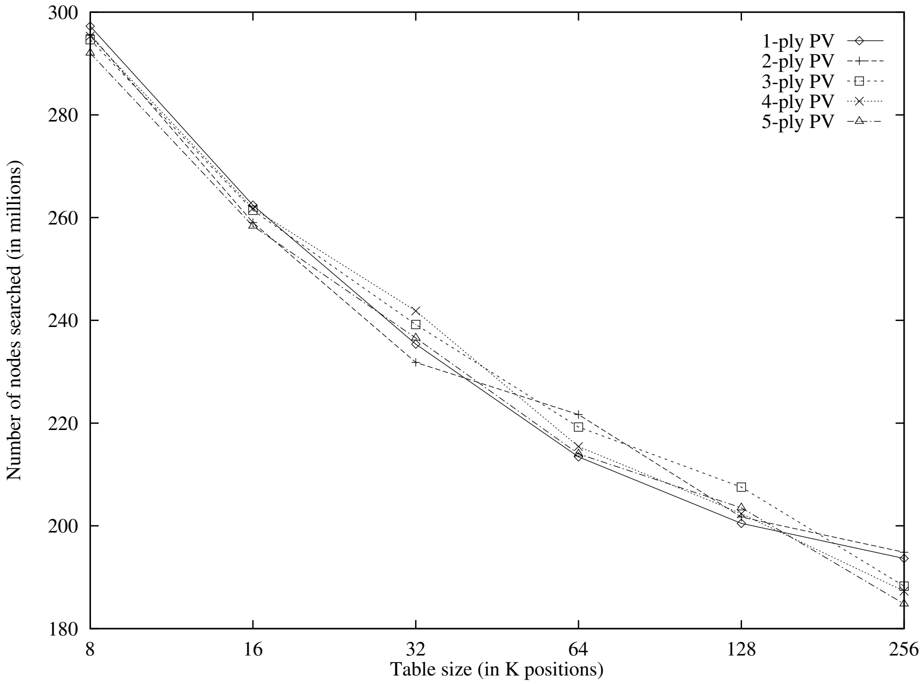

Figure 2.15: Storing an n-ply PV in the chess middle game (8-ply searches).

domineering games, including the standard 8×8 game. Without a transposition table this takes a much longer time and can therefore be considered practically impossible. We have described three series of experiments on the use of a transposition table. The goal of these experiments was to obtain more insight into the first problem statement: which methods exist to improve the efficiency of a transposition table?

First, we have tested which replacement scheme performs best. On logical grounds, one is tempted to conclude that the *number of nodes* of a subtree (used in schemes BIG1 and BIGALL) is a better estimate of the work performed (and therefore potentially to be saved) than the *depth* of that subtree (used in scheme DEEP), especially in positions with a large mobility. The experiments support this logic. In chess middle-game positions and in domineering the schemes based on the concept BIG perform better than the schemes based on the concept DEEP. In chess endgame positions this difference disappears, since the lower mobility then diminishes the differences in effects of the two measures. Based on the 7-ply and 8-ply results in chess middle games, the 10-ply results in chess endgames and the domineering results, we conclude that a two-level scheme is better than any one-level scheme. Thus it follows that the most widely used scheme, DEEP, is not best. Based on the conclusions

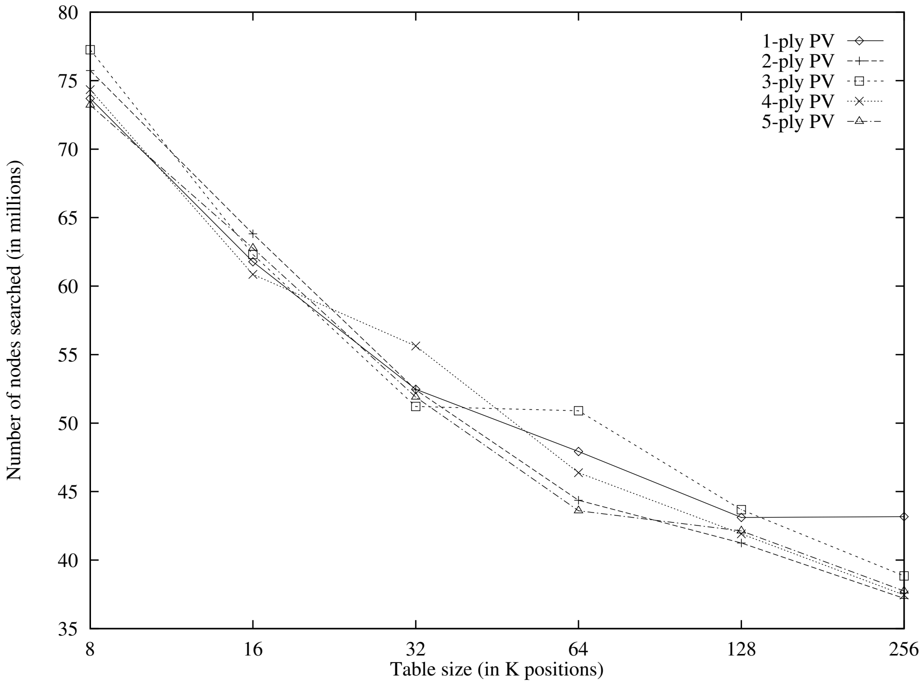

Figure 2.16: Storing an n-ply PV in the chess endgame(10-ply searches).

we recommend using the scheme TwoBig1 as the best replacement scheme for a transposition table.

Second, it is examined which information is more important to store in a transposition-table entry: the best move in a position, or the score of that move. It follows that storing the score of a position is more profitable than storing the best move. This result holds for chess middle-game positions as well as endgame positions. It was also found that for minimal-window search bound values have a much larger effect than exact values. This effect, although nowadays expected, contrasts with the idea for which transposition tables originally were devised, i.e., avoiding the re-search of positions searched before.

Third, we have tested the effect of storing an n-ply PV ($n = 2...5$) in an entry, instead of only the best move (a 1-ply PV). Preliminary results show that a 5-ply PV may win roughly 5% for the chess middle game, and 12% for the endgame, though more experiments are necessary to validate the conjecture that it really is profitable to use additional memory by storing a 5-ply PV instead of increasing the number of positions in the transposition table.

From the experiments it follows that it is important to choose a good replacement scheme. Further, the available memory can be used to make the transposition table as large as possible. However, once a critical transposition-table size has been reached not much is to be gained from doubling the number of positions in the table. In that case, better ways exist for using the available memory. Instead of doubling the number of positions in the transposition table, it is better to use the additional memory by storing more information in an entry, thereby enlarging the entry size. Based on the above experiments it is recommended to concentrate on storing additional information which affects the number of cut-offs generated by bound values.