# Analysis of Transposition Tables and Replacement Schemes

*Sashi Lazar*

Department of Computer Science and
Electrical Engineering
University of Maryland Baltimore County

E-Mail: slazar1@cs.umbc.edu
WWW: http://www.cs.umbc.edu/~slazar1

December 29, 1995

**Abstract**. A set of experiments were designed to measure the enhancement to alpha-beta search using refutation and transposition tables. Results indicate that the use of transposition tables can reduce the size of the game tree by as much as 95%, significantly improving the algorithmic performance of $\alpha\beta$ search. Seven replacement schemes to handle collisions in transposition tables were proposed and their performance measured. Based on the results of the experiment it can be concluded that for large-sized tables all schemes perform nearly identically. Because of the probability of the collisions is inversely proportional to the size of the table, for smaller sizes two level replacement schemes outperformed the single-level implementations.

**Key Terms**: computer chess, transposition tables, collisions, replacement schemes, alpha-beta search, refutation tables

## I. Introduction

The game of chess is one of the most intellectually challenging games man has invented and played over hundreds of years. The complexity of chess has sparked over 200 hundred years of intensive analysis, still failing to exhaust the possibilities of the billions of game positions and strategies. To measure the complexity of the game, researchers have estimated over $10^{100}$ board configurations [1]. ( In light of the fact that there are $10^{88}$ particles in the known universe, this would qualify the game of chess practically inexhaustible. )

Since the birth of the electronic computer, researchers in artificial intelligence have devoted their timeless efforts to build chess programs capable of defeating the human player. While man and machine compete on comparable levels, their identical performance hardly reflects the way such performance is achieved. Human players are frequently unable to assign numeric scores to various positions, their strength lies in the ability to choose the preferable line of play based on accumulated knowledge[2]. As a result of hundreds of years of research there are libraries of collection of games and positions. Despite the amount of knowledge available from these collections, very little chess knowledge is found in modern computer chess algorithms[3]. Instead, game playing algorithms build and analyze game trees by brut-force search.

Early research has found that while constructing the game tree by generating moves and board positions, a large percent of the configurations resulting from different sequences of moves were identical. It seemed obvious that the evaluation or further expansion of such positions would have resulted in duplicate work. To avoid such repetitions modern computer chess programs use a transposition table, usually implemented as a simple hash table with a fixed size to store the values of positions already encountered during the search. Although this method seems promising, problems may arise when two positions hash into the same slot. To handle such collisions, seven different replacement schemes were proposed by Breuker [4].

In this project, I have modified the chess program AliBaba, written by Dennis Breuker, to measure the true performance of the replacement algorithms. Section 2 briefly discusses the most basic chess algorithms and describes how these components were implemented in the original version of AliBaba. The next section describes the modified version of the software, which includes a sophisticated scripting language to maximize user interaction and performance measurement. The seven replacement schemes used in transposition tables, as proposed by Breuker, are explained in detail in section 4. Section 5 is a short tutorial of the experiment design. The results of this project are reported and analyzed in the final section.

## II. Chess Algorithms and AliBaba

Chess programs analyze the game by building trees. The nodes of the tree are the board configurations and the values associated with them, assigned by the computer. In such game trees, we distinguish leaf nodes from internal nodes. Leaf nodes are found at the bottom of the search tree and are not subject to further expansion. All other nodes in the tree are considered internal nodes. A move generation algorithm produces all the possible moves for all the internal nodes in the tree. Most programs use a variation of $\alpha\beta$ search, because it tends to eliminate a considerable size of the tree. This is based on the principle that some parts of the subtree can never be part of any chosen strategy since the opponent will most likely move to avoid this line of play [1].

The goal of any search algorithm is to reduce the tree size as much as possible. Although many small enhancements to the regular $\alpha\beta$ search are used in modern chess programs, the size of the game tree remains exponential. This size is based on the branching factor, which for practical purposes averages around 36, and the depth to which the search is performed. Because of the memory limitations of modern computers this depth ranges from 10 to 30. It has been observed that the size of the $\alpha\beta$ game tree greatly depends on the order in which the moves are selected for expansion [5]. Under perfect move ordering, which is a possible but impractical method, one can construct a minimal $\alpha\beta$ tree, in which the branch leading to the best positional score is selected first.

AliBaba implements $\alpha\beta$ search with several small enhancements. To obtain a "best guess" on the move ordering, iterative deepening is used as suggested by Scott in 1969. The results of the previous search are used to order the moves for the next iteration. Another enhancement used is called the minimal window. This is based on the observation that it is relatively easy and inexpensive to prove that a move is worse that the best move found so far [4].

During the search, only the leaf nodes are subject to static evaluation. A simple evaluation function is applied to all the positions located at the bottom of the search tree. AliBaba uses a very simple method of assigning numeric scores to various board configurations. The overall score of a position consists of a material sub-score and a positional value. The following algorithm describes the evaluation function used in this experiment:

```
Function Evaluate As Integer

If UseTransPositionTable = TRUE
     value = GetMoveFromTransTable
```

```
if NoMoves
     if In-Check
          return MATE
     else
          return DRAW
value = material[ toMove ]+positionalValue[ toMove ] -
             material[ !toMove ]-positionalValue[ !toMove ]
return value
```

The material and positional values for each side are filled in prior to the call to evaluate. The positional values are derived from the mobility factor which is maintained for every piece on the board. Note, that in the figure above, the function returns the relative score for the position since the opponent's score is subtracted from the position's real score.

The reliability of the static evaluation function also depends on how quiescent the evaluated positions were. Because many of the nodes subject to evaluation are not quiescent, to produce a reliable score these positions are further extended. This is called the quiescence search. AliBaba only considers positions in which the king safety is compromised or a piece can be captured. If a position obtained by the quiescence search is found in the transposition table, its value may be used; quiescence positions are however not written to the table.

AliBaba has a fairly simple move-ordering function. For every internal node (position) all legal moves are generated. This does not include pseudo-legal moves such as allowing the king to stay in check. As it was mentioned above, the ordering of the moves determines the size of search tree generated. To minimize search space, AliBaba uses two common move-ordering heuristics.

- *Refutation tables* are used with iterative deepening, retaining one of the major disadvantages of transposition tables: their size limitations. For every iteration of the search, the principle variation is stored in the refutation table. The paths stored from the *d-1* ply serves as basis to search to depth *d* [5].
- According to Schaeffer, *history heuristics* maintain information on whether there is a correlation between a move and any success that the move has achieving a goal. In [4] he defines a sufficient move to be a move that either causes a cutoff in the $\alpha\beta$ search or yields the best minimax score. When a sufficient move is encountered, its history score is increased. Hence moves that are considered to be good get higher scores. In the interior of the search tree, the moves are then sorted based on their history scores. The position with the highest score is picked for expansion first [4].

Because it is possible to reach the same position by more than one path, if a full search tree would be constructed, its interior nodes would be duplicated. Transposition tables use this observation by storing information about each position ( value, depth, size of subtree ) in case an identical positions is encountered. If a position is found in the transposition table and its value is found reliable, it is retrieved from the table and used for further search[5]. Under ideal circumstances all positions and relevant information could be stored in the table, which, given the memory and storage limitations of available computers, is impossible.

AliBaba implements its transposition table as a fixed sized hash table, with continuos memory allocation. Every object ( position ) is assigned a hash value by a simple hash function. The transposition table is capable of storing $2^n$ entries. The n low-order bits of the hash value are used to define the hash index, while the remaining bits are used to distinguish between several positions that might be mapped on the same hash index by the hash function [4]. Although some implementations of transposition tables include an overflow area[6], AliBaba does not come with such an enhancement.

According to Marsland and Hyatt, an entry in the transposition table should contain the following information:

- **Key**. The bits of the hash value are more significant that those of the hash index. The key is needed to distinguish among the different board positions having the same hash index.
- The best **move** in the position. This is the (sufficient) move which either caused a cutoff, or obtained a highest score.
- The **score** of the best move in the position. This score can be a true αβ value or an upper or lower bound. The table should also contain a flag, which indicates the type of the score.
- The **depth** of the subtree searched. In AliBaba this number is the relative depth, that is, if conducting an m-ply deep search where the position is found at depth n, the relative position m-n is to be stored in the table [4].

It is possible that two different board positions have the same hash index, and hence map into the same entry in the table. When such collisions occur, the choice has to be made which position to keep or evict from the table. In this project we will examine seven different replacement schemes. It should be noted here that by increasing the number of bits in the table hash index, the probability of the collisions can be significantly lowered. This option is impractical in our case given the memory limitation of available computer equipment.

## III. Enhancements to AliBaba

Although AliBaba's search engine was very well written, its interface proved to be inadequate for our experimental purposes. To maximize measurability, I needed a more sophisticated user interface that allowed maximum flexibility. Before the design of the experiment, I decided to rewrite AliBaba's main user interface and add a statistical engine with upgraded display capabilities. As a result of this upgrade AliBaba+ was born.

The complicated command line parameters that the program was originally based on were replaced by interactive user commands. The new user interface, based on a full lexical analyzer and a LR(1) parser, quickly transformed a little command line utility into a user friendly application. Appendix A describes the full syntax AliBaba+ uses. Please note, that for backwards compatibility, all the original command line support was preserved.

With this new user interface, based on AliBaba+'s own script language, run-time parameters such as transposition table scheme, or index sizes can be changed and their effects can be measured immediately. The table below describes the function of the SET command, which allows the user to change search parameters from the command line:

```
The SET Utility

      Usage: SET <option> = <value>

CENTER = <string>|DEFAULT     Sets center control function
Piece  = <value>              Sets relative piece values
VALUES = DEFAULT              Resets relative piece values
COLOR  = <color>              Sets player to move
CONFIRM = <on|off>            If 'confirm' is off, the program will
                              perform any command without user
                              interaction.
DEPTH = <value>               Sets default depth used by STEP
NODES = <count>               Sets max nodes to be examined
STATS = <on|off>              Toggles statistical info display
TABLE = <on|off>              Toggles use of transpos table
REFS  = <on|off>              Toggles use of refutation table
DEBUGLEVEL = <values...>      Turns on and off various debug options.
                              These options can be combined by
                              the | operator.
TABLE PLY = <ply>             TTable keeps "ply' info
TABLE LEVEL = <1|2>           Sets table search level
TABLE SCHEME = <scheme>       Specifies replacement scheme
TABLE INDEX = <val>           Sets the #of bits per item
TBALE MOVES = <on|off>        If ON moves are read form table
TABLE VALUES = <on|off>       If ON values read from table
TABLE STAMP = <on|off>        Use timestamp in table
TABLE FLAG = <on|off>         Mark 'old' entries
TABLE DEBUGDEPTH = <value>    ** Internal Use **
```

The following section describes some of the most important options of the SET command. The options for the SET command can be broken up into two categories. The first category controls the general behavior of the applications, turns on or off various search parameters and specifies statistical recording and displaying methods. The second category controls the behavior of the transposition table, such as the replacement scheme and the hash index size.

- Since I wanted to run this experiment with and without transposition tables, I needed a way to turn the table on or off at run-time. The SET TABLE command serves this purpose. It can also be used to activate the use of the refutation table by typing SET REFS = <on|off>.
- As it was discussed earlier, increasing the number of bits used in the hash index can significantly lower the probability of a collision. For this purpose I included the SET TABLE INDEX command which sets the number of bits in the hash index. The default value is 18 and most of the experiment was performed with this default value.
- The SET CENTER option controls the function used when evaluating center control, which is part of the positional evaluation. Currently three functions are supported: LINEAR, EXPO, and DEFAULT. The first sets the center control table to a linear distribution, while the second type uses an exponential score, that is, pieces closer to the center are scored significantly higher than pieces far from the center. The default function distributes the scores evenly.
- For further experiments I implemented a command that allows the user to set the relative values of the pieces on the board. By using the SET <piece> = <value> command one can assign any material value to any piece and observe how the evaluation function changes. The default values (pawn=100, knight=325, bishop=325, rook=500, queen=900, king=1500) can be restored by the SET VALUES = DEFAULT command.
- The DEPTH option determines the maximum depth AliBaba+ will search to. This limitation does not include the depths reachable by the quiescence search.
- To control the maximum number of nodes that $\alpha\beta$ will expand, use the SET NODES command. This option proved to be useful when forcing a deeper search, but one has to be aware of the limit of the available memory.
- To set the replacement scheme used by the transposition table, I included the SET TABLE SCHEME command. It accepts one of the following strings: new, old, deep, big1, bigAll. Please note that AliBaba+ is case sensitive; therefore the scheme names must match these cases.
- With the deep and big1 methods the user may activate a two level transposition table scheme by using the SET TABLE LEVEL command.
- Normally, after a move is made ( the tree has been fully searched ), positions in the transposition table are cleared for the next search. Positions and their values can be preserved for the next iteration by using the time-stamp method. If position time stamping is active, older table entries are

marked, and while their values can be used at the next iteration, in the case of a collision, these positions are evicted regardless of the replacement scheme in effect.

While designing the experiment, I realized that some of these tests will take several minutes, especially searches to 8 or 10 ply without transposition tables. Typing commands and waiting for the results was just not good enough. AliBaba+ therefore features a recursive script running capability, that is, scripts can be run from the command line, and those scripts can run other scripts and so on. This way I was able to design a few scripts most common to all the experiments. The only thing left to do was set some variables, run one script, then change the settings and run the same script under different scenarios. For a complete description of the experiment, see section 5.

## IV Collisions and Replacement Schemes

As discussed earlier, more than one positions may have the same hash key, causing them to be hashed into the same positions in the transposition table. This is commonly known as a collision[4]. When such collisions occur, a decision has to be made about which of the two positions should be preserved in the table. Such a decision is determined by the replacement scheme used in the transposition table. This section describes seven methods of handling collisions as they were proposed by Breuker and implemented in AliBaba+.

1. The most frequently used method is called *deep*. It has been the traditional way to handle collisions in transposition tables in most chess algorithms. When a collision occurs, the node with the deepest subtree searched is preserved in the table. The reasoning behind this method is that a subtree searched to a greater depth usually yields more information and a larger number of nodes than the shallower ones. And since presumably more time was spent searching the deeper subtree, storing this position in the table can save more work.
2. A fairly common and very simple replacement scheme is called *new*. In this method the newer position is kept whenever a collision occurs. This concept is based on the observation that most transpositions occur locally, within small subtrees of the full $\alpha\beta$ search tree [4].
3. Another replacement method is called *old*. With this scheme, when collision occur, the older positions are preserved in the table. Since this scheme was implemented in the original version of AliBaba, it was included in the experiment for the sake of completeness.
4. If the search tree contains a large number of forcing moves or subtrees extended by the quiescence search, the depth of the search tree can no longer serve as a reliable indicator to the amount of work performed. Instead of selecting the deepest subtree, one may wish to select based on the actual number of nodes contained in the subtree. The disadvantages of storing the actual number of nodes searched under a certain subtree is obvious: this method requires extra amount of storage in the

transposition table. One scheme that counts a position in the transposition table as a single nodes is called *big1*.

5. Another variation of replacement schemes that considers the number of nodes searched in the sub-tree is bigAll. This method however counts a position in the transposition table as N, where N is the number of nodes searched to obtain the score for the particular position.
6. Ebeling in 1986 introduced the use of a two-level transposition table. Two level replacement schemes have two table positions per entry, requiring a bit more space [4]. The following algorithm describes the replacement scheme referred to as *twoDeep*:

```
Procedure TwoDeep_Replace

If New.depth >= first.depth Then
     second = first
     first = new
else
     second = new
```

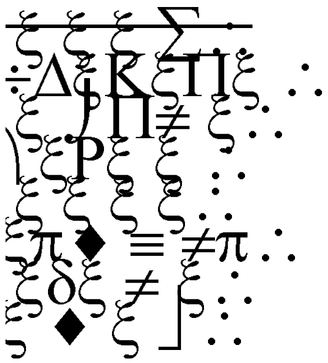
7. The final scheme is called *twoBig*. It is also a two level implementation of the transposition table and it can be view as a combination of new and big1. The replacement algorithm is very similar to the one described above: If the number of nodes searched of the new position is greater than or equal to the first level's number, the first level is shifted down and the new position is stored on the first level. Otherwise, the new position is stored on the second level.

## V. The Experiment

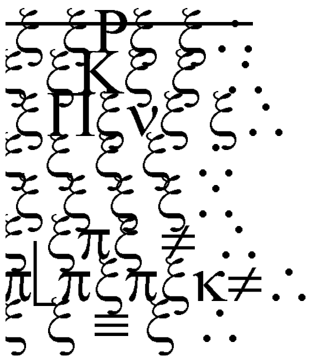
The first part of this project was to develop an easy-to-use multi-purpose interface to AliBaba, and the second was to put it to work. This experiment focused on two questions:

1. How much savings do transposition tables and refutation tables offer in $\alpha\beta$ search in the game of chess, and
2. How well do the seven replacement schemes perform under different conditions.

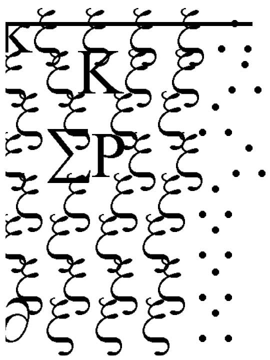
The first part of the experiment was designed to answer the former question. Intuition dictated that we should see a significant improvement by adding refutation tables and later transposition tables to regular $\alpha\beta$ search. The experiment was to collect reliable data to support this intuition. A collection of games was obtained from various Internet sites, including the 1981 Russian championship games, several which had been played by Kasparov vs. Timosjenko in 1981:

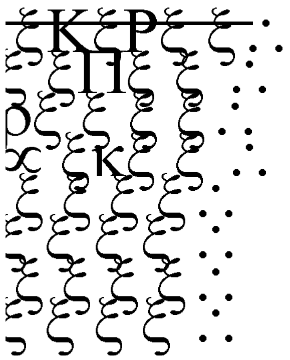


The test programs were designed to run on three different stages of the game:  beginning, middle, and end-games.  The beginning games were not hard to obtain.  I have analyzed some of the opening books used in Crafty, a chess program written by Hyatt, to produce the positions.  The following diagram is, for example, from Ostrich vs. Merlin in move 52, white to move:



I have also tested the program on several classic end games, such as a rook against a knight, two rooks against a queen, two bishops against a rook, etc.





With a collection of over twenty games, several scripts were executed on each board configuration. Every position was subjected to a basic αβ search with and without refutation and transposition tables to different search depths. The depth of the search varied from 3 to sometimes 12. AliBaba+ recorded the number of nodes generated and time it required to produce a move.

The second and more important part of the experiment was designed to produce data that would give us some measurable insights on how well the different replacement schemes handle collisions. To measure their performance, the same collection of games were used. I should note here that although transposition tables are hardly ever used in the first few steps of the game ( modern programs use opening books instead ), for the sake of completeness all tests were performed on opening positions as well.

To ensure the validity of the data, all tests were run under nearly identical circumstances on a Silicon Graphics Indy 100Mhz workstation running UNIX. To measure true performance, all seven replacement schemes were subjected to the same test. This included running several αβ searches to different depths and recording the time required for the program to run to completion, along with the number of nodes examined per second. I also recorded a few more statistics which proved to be a good measure of performance:

1. Total time required to complete the search in milli-seconds
2. The total number of nodes examined during the search
3. I have further broken down the node count to indicate the number of internal nodes,
4. the total number of leaf nodes to which the evaluation function was applied, and
5. the number of nodes extracted by the quiescence search.
6. The number of position filled in the transposition table
7. The load factor of the transposition table after each move
8. The number of reads from the transposition table
9. The total number of writes to the transposition table. This includes positions written during collisions, and

10. the number of hits during the search.
11. The total number of moves generated during the αβ search and
12. the how many times was αβ called recursively.

The final step to complete the experiment was to create a uniform script file. Its purpose was to ensure that all seven replacement schemes were to be tested under the same constrains and conditions, therefore the same script was executed for all the algorithms. The basic structure of the script is described in the figure below:

```
Clear Current Statistics
Set Table Scheme
Set Table Indexing Level
Set Search Depths

Load Game1
Run Test
Load Game2
Run Test
…
Save Statistics
```

This script was then executed for all seven replacement schemes and the information recorded by the statistic engine was saved to a comma delimited text file for further processing. The next section presents the data accumulated as the result of this experiment, and offers an explanatory discussion on the performance analysis of the transposition table replacement algorithms.

## VI. Results and Discussion

AliBaba+ displays both intermediate and final search results in a tabular format. In order to analyze its performance, the user must become familiar with the display format used by the application. When the PRINT BOARD command is issued, the following information is displayed on the terminal screen:

```
WTM> print board
WTM  WS WL BS BL 7200+0 7200+0       (732abd88,db39bcaf)
  +-----------------------+
8 |:r: n :b: q :k: b :n: r |
7 | o :o: o :o: o :o: o :o:|
6 |:::   :::   :::   :::   |
5 |   :::   :::   :::   :::|
4 |:::   :::   :::   :::   |
3 |   :::   :::   :::   :::|
2 |:O: O :O: O :O: O :O: O |
1 | R :N: B :Q: K :B: N :R:|
  +-----------------------+
    A  B  C  D  E  F  G  H
```

Besides the current position, AliBaba+ indicates the side to move in the upper left hand corner, followed by the castling options. The example above shows that both white and black may short and long castle. The following two numbers indicate the material and positional relative scores for both white and black respectively.

When the STEP command is issued, AliBaba+ will search to a specified depth set by the SET DEPTH option or supplied as an argument to the STEP command. The program displays intermediate results after each iteration in the following format:

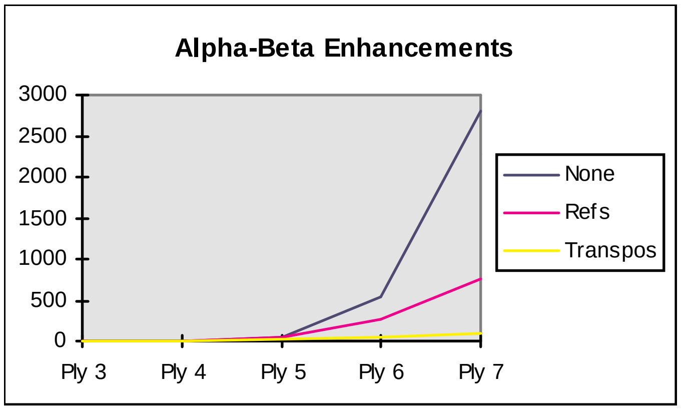
```
WTM> step 4
[0 positions remaining in transposition table]
 1     0.19s  -3.430  g3-g4  .Re8xe6                               2
       0.23s     >>   h2-h4  .Re8xe6                               2
       0.24s  -3.270  h2-h4  .Re8xe6                               2
       0.25s     >>   c2-c4  .Re8xe6                               2
       0.25s  -2.970  c2-c4  .Re8xe6                               2
       0.28s     >>  Ne6-g5  .Re8xe1                               2
       0.28s  -2.690 Ne6-g5  .Re8xe1                               2
       0.30s     >>  Ne6-c5  .Kd7-c8                               2
       0.30s   0.860 Ne6-c5  .Kd7-c8                               2
====================[ 0.35 ]====================
 2     0.36s   0.860 Ne6-c5  Kd7-c8                                2
====================[ 0.45 ]====================
 3     0.57s     <<  Ne6-c5  Kd7-c8   a2-a3  .Qb2xa3               4
       0.87s  -0.460 Ne6-c5  Kd7-c8  Nc5-e4  .Qb2xc2               4
====================[ 1.60 ]====================
 4     1.94s  -0.460 Ne6-c5  Kd7-c8  Nc5-e4  Qb2xc2                4
====================[ 4.15 ]====================
2573 nodes (int:976, leaf:1597, quies:400), 620 n/s
486 filled ( 0.74%), 2586 reads, 193 hits ( 7.46%) 573 writes
6 collisions, 1 expected
16170 moves in 491 times (32.93)
My move: Ne6-c5
```

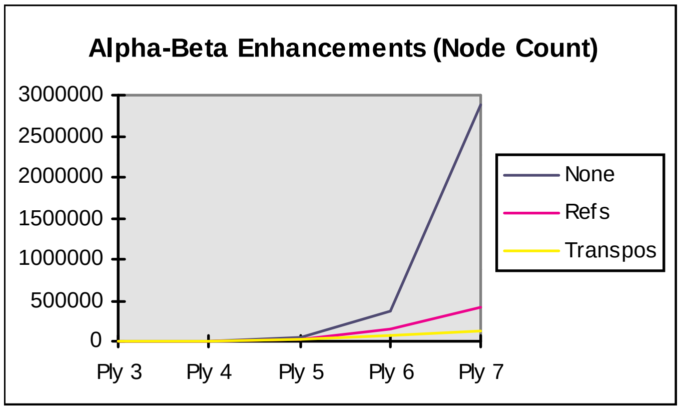
For each depth, the total time in seconds is displayed in the middle of the table and between the iterations. When $\alpha\beta$ finds a new upper bound, it displays the elapsed time along with the move currently examined. Moves preceeded by a dot are results of the quiescence search. On the right hand side of the table the maximum search depth is displayed. The << and >> indicates whether the search has failed low or high respectively. At the bottom of the table the accumulated statistics are displayed.

The information collected from the result of the first part of our experiment supports the argument that both refutation and transposition tables offer significant improvements to the regular $\alpha\beta$ search by providing a better move ordering mechanism and therefore reducing the number of positions to be search by the program. The results of running a 3 through 6 ply search with and without refutation and transpositions tables are represented in the chart below:

| Time (sec) | Node Count | Internal | Leaf | Quies |
|---|---|---|---|---|
| | | **Regular αβ** | | |
| 1.5 | 1046 | 301 | 745 | 209 |
| 11.82 | 8548 | 3686 | 4862 | 3259 |
| 59.77 | 43328 | 18966 | 24362 | 14875 |
| 531.79 | 374527 | 197007 | 177520 | 158887 |
| 2804.42 | 2886728 | 1800864 | 1085864 | 409374 |
| | | **Refutation Table** | | |
| 1.8 | 1011 | 281 | 730 | 191 |
| 6.3 | 3735 | 1382 | 2353 | 1106 |
| 59.01 | 34634 | 15025 | 19609 | 11569 |
| 268.83 | 153780 | 73106 | 80674 | 44748 |
| 745.24 | 421701 | 227649 | 194052 | 198452 |
| | | **Transposition Table** | | |
| 0.83 | 1011 | 281 | 730 | 191 |
| 2.71 | 3550 | 1283 | 2267 | 1026 |
| 18.21 | 23968 | 10067 | 13901 | 7676 |
| 53.61 | 71492 | 27038 | 44454 | 14881 |
| 88.56 | 114969 | 50372 | 64597 | 37429 |

The numbers in the chart presented above clearly indicate a significant improvement when the search engine was enhanced with refutation and transposition tables. The following two graphs compare the running time and the number of nodes examined during the three test phases:





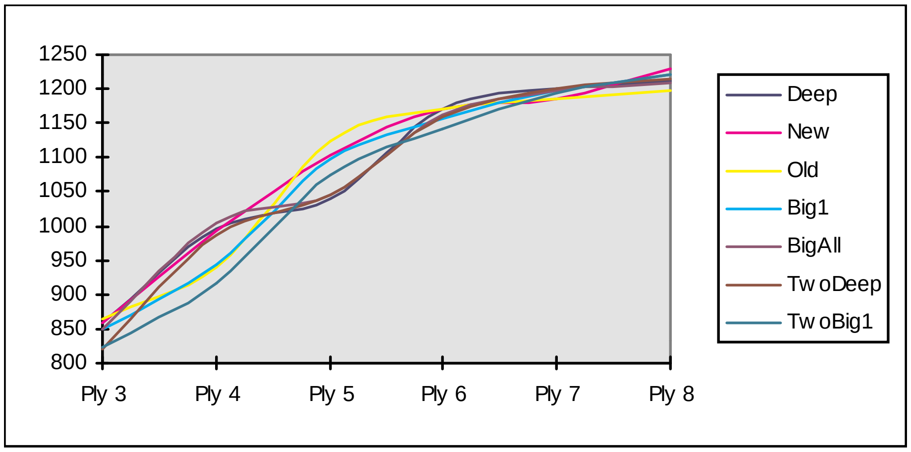
The graphs above indicate that the most significant changes occur when the regular αβ search is enhanced with the use of transposition tables. To explain these findings, I have analyzed the relative number of nodes examined when transposition tables were used.

| None | Transpos | Ratio |
|---|---|---|
| 1046 | 1011 | 3.35% |
| 8548 | 3550 | 58.47% |
| 43328 | 23968 | 44.68% |
| 374527 | 71492 | 80.91% |
| 2886728 | 114969 | 96.02% |

This data suggests that when searched deeper than 6 plies, over 80% of the positions were already in the transposition table. In order to determine how close the generated tree was to the optimal minimum game tree, one would have to first construct such a tree. Because this method proved to be impractical, we can only make approximations. Schaeffer found that with the combination of transposition tables and history heuristics, the tree size can be reduced by as much as 99%, and the trees generated during the search are within a factor of 1.5 times that of the minimal game tree [5].

The second part of the project is unique. To the best of my knowledge no one has designed an extensive experiment to measure the performance of replacement algorithms used in transposition tables. The seven schemes tested were: deep, new, old, big1, bigAll, twoDeep, and twoBig1. The following table shows the results of executing a 3 though 7 ply search on middle games using the different replacement schemes. The values in the table indicate the extracted nodes per second.

| Deep | New | Old | Big1 | BigAll | TwoDeep | TwoBig1 |
|---|---|---|---|---|---|---|
| 858 | 858 | 864 | 850 | 849 | 822 | 822 |
| 995 | 993 | 941 | 944 | 1004 | 988 | 917 |
| 1040 | 1104 | 1123 | 1099 | 1045 | 1044 | 1074 |
| 1170 | 1172 | 1170 | 1156 | 1162 | 1158 | 1142 |
| 1199 | 1186 | 1185 | 1196 | 1197 | 1199 | 1195 |
| 1211 | 1231 | 1198 | 1222 | 1209 | 1216 | 1222 |



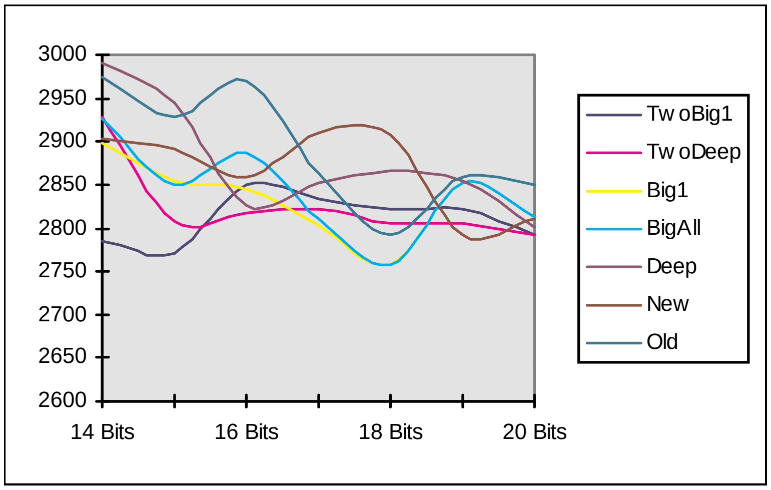
Although at first look the graph indicates that almost all of the algorithms performed identically, summing up the nodes examined per seconds over many games suggests that deep and two deep perform slightly better than the other five. Data collected indicates however, that in middle games choosing, a different replacement schemes to handle collisions in transposition tables is not likely to improve the overall performance.

This can be explained by the size of the transposition table chosen. The test scripts were run on transposition tables with 18 bit indexing, that is the number of entries in the table was $2^{18}$. The following table represents the load factor of the transposition tables on different plies:

| | Deep | New | Old |
|---|---|---|---|
| Ply 3 | 0.18 | 0.27 | 0.216 |
| Ply 4 | 0.65 | 0.975 | 0.78 |
| Ply 5 | 5.01 | 7.515 | 6.012 |
| Ply 6 | 27.78 | 41.67 | 33.336 |
| Ply 7 | 21.11 | 31.665 | 25.332 |
| Ply 8 | 54.73 | 82.095 | 65.676 |

Because of the relatively low load factor of the transposition tables during the search, collisions occur with a low probability, lowering the significance of the replacement algorithm used.

In order to determine which algorithm ( if any ) is the best, I changed the table hash index size from 14 bits to 20 bits. The reduction in the table size increases the probability of collisions, which should magnify the individual performances of the seven replacement schemes. The graph below represents the result of this experiment:



The data represented by the graph indicates that for smaller transposition tables, the two level replacement schemes slightly outperform the single level implementations. This is an important result, because this indicates that while representing positions in the transposition table using a two-level method reduces the maximum number of entries in the table, this disadvantage appears to be negligible on the performance of the algorithm.

## VII. Conclusion

As a part of this project I developed an advanced version of the AliBaba chess program. Its enhanced user interface allowed us to fine tune and measure several enhancements to the regular αβ search and to measure the performance of the seven replacement algorithms used in transposition tables to handle collisions.

Data collected from this experiment indicated that the use of refutation tables and especially transposition tables can significantly reduce the size of the game tree searched by αβ. With the combination of these two enhancements we were able to show a 95% reduction in search time and space.

Several test programs were constructed to measure the performance of the seven proposed replacement schemes: deep, new, old, big1, bigAll, twoDeep, and twoBig. The results of this experiment indicated that slight differences can be observed on smaller size transposition tables, but when the table hash index exceeds 18 bits, differences in performances among the algorithms fade away. We have argued that because the increasing size of the transposition tables reduce the probability of a collisions, the algorithmic performances of different replacement schemes become identical. For small table sizes, the experiment indicated that two level hash tables outperform the single level methods, dispite the reduction in the maximum number of entries in the transposition table.

AliBaba+ was designed to allow the user to change the relative piece values as well as the function that controls the center control evaluation methods.  This is no accident, since for future work I plan to explore how the relative piece values influence the computer’s evaluation and decision making.

## VIII. Acknowledgments

I would like to thank Dr. Alan Sherman and Dr. Jim Mayfield at the University of Maryland for their semester long support with this project.  Special thanks to Dennis Breuker for the implementation of the original AliBaba program, and to Dr. Jonathan Schaeffer at the University of Alberta for his support.

## IX References

1. Jonathan Schaeffer, Experiments in Search and Knowledge, Ph.D. thesis, Dept. of Computer Science, University of Waterloo, 1986
2. Helmut Horacek, Reasoning with Uncertainty in Computer Chess, Artificial Intelligence 1990, pp. 37-56
3. D. Hartmann, Notions of Evaluation Functions Tested Against Grandmaster Games, Advances in Computer Chess, Elsevier Science Publishers, 1989
4. Dennis m. Breuker, Replacement Schemes for Transposition Tables, 1993
5. Jonathan Schaeffer, The History Heuristics and Alpha-Beta Search Enhancements in Practice, University of Alberta, 1989
6. Zobrist A.L. A new Hashing Method with Application for Game Playing.  Technical report #88, Computer Science Department, The University of Wisconsin, Madison, 1990
7. Marsland T.A, A Review of Game-Tree Pruning, ICCA Journal, Vol. 9, No. 1, pp. 3-19, 1986

# Appendix A - AliBaba+ Command Reference

```
List of legal commands.

   #define "<const>" <value>          MOVE <pos-pos>
   #undef  "<const>"  (see below)     NEW
   BACK <moves>                       PLACE <color><piece> ONTO <square>
   CLEAR <BOARD|TABLE>                PRINT ... (see below)
   DELETE <Piece>                     QUIT
   EVALUATE [Color]                   RUN <script>
   ECHO [on|off]                      SAVE ... (see below)
   FORWARD <moves>                    SET ... (see below)
   HELP                               SHOW <BOARD|MOVES> = <value>
   LOAD <filename>                    STEP <depth>

#define and #undef
      These two keywords allow the user to define system wide constants
      and shortcuts to keywords.  NOTE: The type of the 'value' can be
      STRING INT or BOOLEAN.  Quotes around the strings are optional.
  The PRINT Command
      The print command displays information on several optional topic
      on the terminal.  Valid topics are: BOARD, MOVES, GAME, VALUES,
      TABLE, SETTINGS, and VARS.  If the PRINT command is invoked with
      any other string, the specified string will be echoed.
  The SAVE Command
      The SAVE command allows the user to save the board and game state
      to an aux file.  The format of the file might be ASCII or Latex.
      When ASCII is used, both the board and the game (with its moves)
      are stored.  The LATEX type only saves the position.  With the
      save command it is also possible to save the current statistics.
             SAVE <filename> AS <ASCII|LATEX>
             SAVE STATS <filename>
  The SET Utility
      The SET command allow you to set system variables at runtime. This
      gives a great flexibility to measure system performance under
      different settings.
      The SET command can take the following parameters:

      CENTER = <string>|DEFAULT Sets center control function
      Piece  = <value>          Sets relative piece values
      VALUES = DEFAULT          Resets relative piece values
      COLOR  = <color>          Sets player to move
      CONFIRM = <on|off>        If 'confirm' is off, the program
                                 will perform any command without
                                user interaction.
      DEPTH = <value>           Sets default depth used by STEP
      NODES = <count>           Sets max nodes to be examined
      STATS = <on|off>          Toggles statistical info display
      TABLE = <on|off>          Toggles use of transpos table
      REFS  = <on|off>          Toggles use of refutation table
      DEBUGLEVEL = <values...>  Turns on and off various debug
                                options.  These options can be
                                combined by the | operator.
      TABLE PLY = <ply>         TTable keeps "ply' info
      TABLE LEVEL = <1|2>       Sets table search level
      TABLE SCHEME = <scheme>   Specifies replacement scheme
      TABLE INDEX = <val>       Sets the #of bits per item
      TBALE MOVES = <on|off>    If ON moves are read form table
      TABLE VALUES = <on|off>   If ON values read from table
      TABLE STAMP = <on|off>    Use timestamp in table
      TABLE FLAG = <on|off>     Mark 'old' entries
      TABLE DEBUGDEPTH = <value>** Internal Use **
```

## Table of Contents