

BAR-ILAN UNIVERSITY

An Orthodox k-Move
Problem-Composer for Chess
Directmates

Fridel Fainshtein

Submitted in partial fulfillment of the requirements for the Master's Degree in
the Department of Computer Science, Bar-Ilan University

This work was carried out under the supervision of

Dr. Yaakov HaCohen-Kerner

Department of Computer Science

Jerusalem College of Technology (Machon Lev),

Jerusalem, Israel

and

Prof. Nathan Netanyahu

Department of Computer Science

Bar-Ilan University,

Ramat Gan, Israel

Acknowledgements

I would like to thank Dr. Yaakov HaCohen-Kerner, my supervisor. His generous help, constant advises, and experience highly contributed to the completion of this thesis. Dr. HaCohen-Kerner spent a lot of time during the entire process. In my opinion, he is the best supervisor and teacher that a student would wish.

I would also like to thank Prof. Nathan S. Netanyahu for his fruitful comments.

I would like to thank Dr. Yair Wiseman for his help concerning computer architectures.

I would like to thank Josef Retter and Jean Haymann, both of whom are international masters of the FIDE for chess composition. These two international masters supplied us with relevant background concerning composition of chess problems.

I would like to thank Bar-Ilan Computer Science Thesis Committee: Prof. Sarit Kraus, Prof. Amihood Amir, and Prof. Amir Herzberg, for the valuable comments.

I would also like to thank my mother, Olga Fainshtein, and her friend, Batya Yakobi, for their help with the English language.

I would like to thank my wife, Tanya, for her generous support during the work on this thesis.

Abstract

Computerized composers of chess problems are very rare. Moreover, they produce neither impressive nor creative new mate problems. A previous model, called an Improver of Chess Problems (ICP) improved slightly the quality of 10 out of 36 (about 28%) known 2-movers (problems in which White has to mate Black in two moves against any defense of Black). ICP was based on hill-climbing search and a quality function for evaluating the problems.

Such a quality function can help to composers to achieve better problems with the computer help. The quality function was built using chess problems literature and consulting two international masters in chess composition. Comparing newly generated problem to the original problem using the values of both, we can see immediately which is better. Automating the process, we can do the whole process fully automatic.

In this thesis, two improved models for 2-movers are described. The first is called Deep Improver of Chess Problems (DICP). This model uses an improved version of Bounded Depth First Search (BDFS) and an improved version of a quality function. The experiment we carried out on the ICP's database showed that the quality of 32 problems (about 89%) was improved. Moreover, DICP's improvements are better in their quality in comparison with those of ICP.

The second model is called Chess Composer. It uses an ordered version of Depth First Iterative Deepening (DFID) and has the same quality function as in DICP. The results of the experiment we carried out on 100 known problems show that the quality of 97 of the problems (97%) was improved. Some of the improvements are rather impressive, considering that most of the tested problems were composed by very experienced composers.

These new improved problems can be regarded as creative because they are better, and they are often not too similar to the original problems. Moreover, some of the improvements are meaningful and impressive from the viewpoint of chess composition.

Finally, a general theoretical model, the k -move Chess Composer, is proposed. Practically, it uses the same algorithm as in Chess Composer for 2-movers. To detect whether or not a problem has a solution, Depth First Search (DFS) with pruning was used on AND/OR trees. A special metric has been built in purpose to improve solution trees of k -movers (A solution tree is a tree developed for all possible moves of Black and the right responses of White). This metric can be used in a future evaluation function.

All proposed models use a 64-bit board representation. In general, 64-bit board representation means a flexible structure for representation a board using 64-bit computer words. This representation slightly improves the search speed. Experiments show that move generator of our composer is faster by about 25% than the move generator of professional programs that use only a 32-bit board representation.

Keywords: Computer Chess Composition, Automatic Problems Construction and Improvement, DFID, 64-bit Board Representation

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 8 |
| 2 | Background | 11 |
| 2.1 | Chess and its Complexity | 11 |
| 2.1.1 | Chess | 11 |
| 2.1.2 | The Complexity of Chess | 11 |
| 2.2 | Classification of chess problems | 12 |
| 2.2.1 | Rules of Orthodox Directmates. | 13 |
| 2.2.2 | Composition Themes | 15 |
| 2.3 | Chess Problems and Computer Chess Programs. | 16 |
| 2.3.1 | Problem Solving | 16 |
| 2.3.2 | Conspiracy Numbers | 17 |
| 2.3.3 | Proof Numbers | 17 |
| 2.4 | Endgames | 18 |
| 2.5 | Composing of Chess and Chess-like Problems | 19 |
| 2.5.1 | Composition of Two-Move Mate Problems. | 19 |
| 2.5.2 | A Global Scheme for Problems Composing | 21 |
| 2.5.3 | Composing in Tzume-Shogi | 22 |
| 3 | DICP, 2-move Chess Composer, and <i>K-move</i> Chess Composer Models | 24 |
| 3.1 | General Description | 24 |
| 3.2 | Heuristic Quality Function. | 29 |
| 3.3 | Complexity of DICP and Chess Composer | 30 |
| 3.4 | 64-bit Representation | 37 |

| | | |
|----------|--|------------|
| 3.5 | Comparison to Other Models | 40 |
| 3.6 | <i>K-movers</i> Model ($k > 2$) | 42 |
| 3.6.1 | The <i>K-movers'</i> Improver Algorithm ($k > 2$) | 48 |
| 4 | Experimental Results. | 57 |
| 4.1 | DICP. | 57 |
| 4.2 | Chess Composer | 62 |
| 4.3 | Illustrative Examples | 68 |
| 4.3.1 | Example 1 | 68 |
| 4.3.2 | Example 2 | 71 |
| 4.3.3 | Example 3 | 75 |
| 4.3.4 | Example 4 | 77 |
| 5 | Summary and Future Research | 82 |
| | References | 85 |
| | Internet Links | 91 |
| | Appendix A - The Game of Chess. | 92 |
| | Appendix B -The ICP Heuristic Function | 98 |
| | Appendix F - Definitions of New Themes | 100 |
| | Appendix D - 64-bit Representation | 105 |
| | Appendix E - Results of the Improvement Processes | 111 |
| | Appendix F - Example of applying order on additions of White pieces | 121 |

List of Figures

| | |
|---|-----------|
| Figure 1: Simple Bounded DFS used in DICP | 25 |
| Figure 2: Optimized Depth First Iterative Deepening Composer. | 27 |
| Figure 3: The additions-tree up to depth $d = 2$ | 31 |
| Figure 4: An example of a solution tree for 3-movers | 42 |
| Figure 5: An example of an "ideal" tree for 3-movers | 44 |
| Figure 6: A metric for measuring k -move trees | 45 |
| Figure 7: The general algorithm for improving k -movers | 49 |
| Figure 8: The main algorithm for solving a k -mover | 49 |
| Figure 9: An algorithm for checking if a problem has a unique solution in k moves | 51 |
| Figure 10: Solved-unsolved recursive pruning procedure | 53 |
| Figure 11: An algorithm for k -move metric calculation | 55 |
| Figure 12: Improved problems as a function of the change in number of themes | 61 |
| Figure 13: Improved problems as a function of the change in number of pieces | 61 |
| Figure 14: Distribution of transformed new positions for all levels | 62 |
| Figure 15: Change in scores | 63 |
| Figure 16: Average growing in themes | 64 |
| Figure 17: Almost half of the problems achieve additional themes | 65 |
| Figure 18: Number of improved problems as function of the change in pieces at the best-scored problems | 66 |
| Figure 19: Distribution of transformations | 67 |
| Figure 20: Run-time as a function of the number of pieces | 67 |

List of Tables

| | |
|--|------------|
| Table 1: Average number of generated positions and running time by <i>Chess Composer</i> for all 100 original problems | 36 |
| Table 2: Generation of all possible positions from the initial chess position up to 8 plies with different configurations | 38 |
| Table 3: Number of all generated improvements by DICP for the 36 original problems | 58 |
| Table 4: Rate of improved problems in ICP and DICP (best improvement for each problem) | 58 |
| Table 5: Statistics concerning the performance of DICP for all 32 improved problems. | 59 |
| Table 6: Distribution of improvement transformations. | 60 |
| Table 7: Number of all improvements generated by <i>Chess Composer</i> for all 100 original problems | 64 |
| Table 8: The pieces in chess | 92 |
| Table 9: ICP's themes | 98 |
| Table 10: ICP's bonuses | 99 |
| Table 11: ICP's penalties. | 100 |
| Table 12: Shannon's relative values for pieces. | 100 |
| Table 13: Scores for the new definitions. | 103 |
| Table 14: Primitives for the board representation. | 105 |
| Table 15: Helpful variables | 106 |
| Table 16: Pawns' moves and captures | 107 |
| Table 17: Use of masks for knights and kings | 108 |
| Table 18: Moves and captures by sliding pieces via example of White rooks. | 109 |

List of Diagrams

| | | |
|--------------------|--|-----------|
| Diagram 1: | H. Weenink, Good Companion, 12/1917. Original Problem | 69 |
| Diagram 2: | The best improvement by Chess Composer. | 69 |
| Diagram 3: | Unknown source 1. Original problem. | 71 |
| Diagram 4: | The best improvement by ICP | 72 |
| Diagram 5: | The best improvement by Chess Composer | 73 |
| Diagram 6: | Geoffrey Mott-Smith. The Chess Review, December, 1937.. . . . | 75 |
| Diagram 7: | The best improvement by Chess Composer | 76 |
| Diagram 8: | Werner Speckmann, Neueste Kieler Nachrichten, 1939. Original problem | 78 |
| Diagram 9: | The best improvement by ICP | 78 |
| Diagram 10: | The best improvement by Chess Composer | 80 |
| Diagram 11: | The initial position | 92 |
| Diagram 12: | King's moves | 93 |
| Diagram 13: | Knight's moves | 93 |
| Diagram 14: | Pawn's moves | 93 |
| Diagram 15: | Rook's moves | 93 |
| Diagram 16: | Bishop's moves | 94 |
| Diagram 17: | Queen's moves | 94 |
| Diagram 18: | Castling | 95 |
| Diagram 19: | Pawn's captures | 96 |

1. Introduction

Why computer chess composition? Why chess? The question "can a machine think?" has been asked by many researchers, e.g., Shannon (1950), Turing (1950, 1953), and McCarty (1997). To answer this question, all of them used chess as a model. Chess was chosen because it has simple definite rules, well understandable goals, and clear-cut problems. In addition, chess complexity is relatively high. These facts make chess a very useful instrument for researches in Artificial Intelligence. McCarty (1997) quotes the words of the Russian mathematician Alexander Kronrod: "Chess is the *Drosophila* of Artificial Intelligence", which means that chess is as fundamental for Artificial Intelligence as is the fruit fly, *Drosophila*, for genetics.

Talking about computer chess, one may mean a situation where a computer plays against a human being or against another computer. Historically, this situation has been well studied. The most famous chess artificial player was IBM's Deep Blue. This machine is the only one known that defeated a world champion in a full match. The machine beat Garry Kasparov 3.5:2.5 in 1997 (Campbell et al. 2002, Schaeffer et al. 1997, Seirawan 1997). Nowadays top machines are stronger than Deep Blue, rather because of smarter algorithms than because of the wide use of processors (see last results of ICGA computer chess competition [il-2]).

The sub-domain of chess problems composition might be even more appropriate for AI research because: (1) compared to chess, it is relatively an uninvestigated research domain, and (2) the branching factor in chess composition is about ten times higher than the branching factor in chess.

1. INTRODUCTION

Chess problems solving and composing are different from chess playing. Chess is a two-player game, while composing and solving chess problems are considered as a one player game.

A chess problem can be viewed as an art work, while it is not true for most of played chess games. A chess problem usually contains one or more ideas that are called "themes". In our model, we would like to automatically create problems containing themes using a computer. Thus, following Shannon's thoughts, our question is: "Can a machine be an art creator?" This is the main contribution of the research to Artificial Intelligence.

Chess problems have at least 700 years of history. Harley (1931), in his famous book about 2-movers (i.e., chess problems in which White has to mate Black in two moves against any defense of Black), brings as an example a 13th century composition (problem #1). However, modern chess problem composition started in the middle of the 19th century. Sam Loyd, the inventor of the famous 15-puzzle, was also one of the best chess problem-composers in the 19th century. Some of his compositions are presented in Harley (1931) and in Howard (1943).

Since the Middle Ages lonely authors, the modern community of chess problem solvers and composers has grown. Searching for "chess problem" on Google (www.google.com), one will be surprised to find many different groups of interest in the majority of civilized countries.

Modern chess problems are different from ancient chess problems. Modern problems are usually seen as art compositions rather than just chess puzzles. To be a modern problem, it must have an *artistic value*.

While almost every chess player has had experience as a problem solver, there are only a few composers of chess problems. To compose a problem is much harder for a human than to solve it.

1. *INTRODUCTION*

The goal of this research is to build a model that will aid with composing new chess problems in a desired number of moves. Problems with a high artistic value are the one interesting to the community of modern chess problems composers and solvers. The models proposed in this research also answer the needs of modern chess composition.

Harley (1931, 1944), Howard (1943), and Rice (1996) explained in their classic books the composing process from the viewpoint of human composers. For a human, composing a chess problem is much harder than solving it. For a computer, the same is true too. To solve a problem, a computer need just develop and check all possible variants to the desired depth. To compose a problem is not that simple. The new problem should be both legal according to chess composition rules and containing themes.

In this thesis, as a starting point, various chess problems were collected from different chess composition books. Applying series of transformations, these problems were changed with the purpose of improving them in terms of themes. The results are impressive. In case of 2-movers, 97% of problems were improved. Some of the improvements are regarded as impressive from the chess composition viewpoint. Our evaluation function is built mainly due to inventors of chess composition theory, Harley and Howard. Also, two chess experts were involved during the creating of the evaluation function.

The thesis is organized as follows. Chapter 2 contains background and related research. Chapter 3 describes our proposed models. Chapter 4 presents experimental results and their analyses. Finally, chapter 5 contains conclusions and a discussion on future research.

2. Background

2.1. Chess and its Complexity

2.1.1. Chess

Chess rules are very simple and well-defined. Appendix A gives a full description of these rules, which belong to orthodox chess (i.e., chess with regular board, pieces, and rules). Orthodox chess playing is not the only chess-related domain. Fairy chess is a good example for a chess-related game, where some rules are changed. For example, a board may be cyclic or with restricted areas. There may be additional pieces. Rules may be different. For instance, White and Black may wish to cause the opposite side to mate their own king (self-playing).

A Japanese game, Shogi, is an excellent example for a chess-like game with different rules. There are 81 squares on the board. There are additional pieces. Shogi's initial position is also different. Captured pieces may be added on the board by the capturing side. Other differences are described in Matsubara et al. (1997).

2.1.2. The Complexity of Chess

Shannon (1950) estimated the number of different legal chess positions to be about 10^{43} . Shannon reached his estimation considering the following number of combinations,

$\frac{64!}{(32! \times (8!)^2 \times (2!)^6)}$, which is roughly 4.63×10^{42} . The explanation for this combinatorial

value is as follows. There are 64 squares on the chessboard, 32 different pieces, 8 White

2. BACKGROUND

pawns, 8 Black pawns, 6 different groups of two identical pieces (2 rooks, 2 bishops and 2 knights for both sides), and 4 groups of one piece each (king and queen for both sides) which do not affect on the denominator.

This calculation takes into consideration all possible arrangements of the pieces on the board. However, there are positions which are not legal (e.g., two kings in neighborhood squares, both kings are checked, etc.). Therefore, chess problemists and mathematicians (Nievergelt, 1977) estimate the number of different legal chess positions to be 10^{40} .

2.2. Classification of Chess Problems

All chess problems can be classified into the following groups:

Definition 1. *A Chess problem* is a puzzle set by a composer which is to be solved. *A Chess problem*, differently from crosswords puzzles, sudoku, etc., has chess or chess-like rules.

Definition 1.01 *Orthodox chess problems* are problems with rules of regular chess.

Definition 1.02 *Fairy chess problems* are problems in which there are non-orthodox rules, non-orthodox pieces, and non-orthodox boards. (An orthodox chess problem can be viewed as a fairy chess problem without any non-orthodox extensions.) For instance, Trice (2004) describes 80-squares chess proposed in 1920 by then world champion, Kapablanka. Van Haeringen et al. (2003) introduce Superchess with a description of various kinds of chess-like pieces.

Definition 1.03 *Near chess problems* are problems based the use of chess pieces. The most famous examples are the Eight Queens' problem and the movement of a knight throughout the entire board.

Another break down is by stipulations, i.e., what should the solver do:

Definition 2.

2. BACKGROUND

Definition 2.01 *Directmates* are problems of type “mate in N moves”. This stipulation means that White starts and mates Black in N moves, assuming that both White and Black make their best moves. Such problems are called "2-movers", "3-movers", "4-movers", and, generally, " N -movers".

Definition 2.02 *Helpmates* are problems of the form: "Black starts and helps White to gain a mate in N moves".

Definition 2.03 *Selfmates* are problems of the form: "White's strategy is to force Black to mate him in N moves".

Definition 2.04 *Retrograde analysis problems* are problems of the form: "Prove that a given problem can be achieved from the initial chess position".

Definition 2.05 *Studies* are problems of the form: "White starts and reaches a winning or a stalemate position in N moves”

There are some additional types of problems. However, the above five (2.01-2.05) are most common.

The orthodox directmate chess problems in 2-, 3-, 4-, and 5- moves due to definitions 1, 1.01, and 2.01 are the object of this research. This type of problems is one of the most popular both for solving and composing. Problems in 2-, and 3-moves were studied by chess composers (e.g., Harley (1931, 1944), Howard (1943)) very thoroughly.

2.2.1. Rules of Orthodox Directmates

Composition of chess problems has its specific rules. There are several criteria of quality and correctness for chess problems (see Harley 1931, 1944 and Rice 1996 for more details). Orthodox directmates must be legal. According to Howard (1943), *Legality* means:

- 1) The position is legal according to the chess rules described in Section 2.1.

2. BACKGROUND

- 2) There are no more than 8 pawns, 2 knights, 2 bishops, 2 rooks, 1 queen and exactly 1 king for each side.
- 3) Bishops of one side are not allowed to be of the same square color.
- 4) It must be proved that the problem can be developed from the initial position (Figure 9). Retrograde analysis is one of the ways to prove it.
- 5) Castling and en-passant (see Appendix A) are legal unless it has been explicitly disallowed. Several kinds of problems implicitly ask to prove that castling or enpassant are legal.
- 6) The first move of White, which is called the *keymove*, is unique. That is, if there is another move of White that forces another solution, then the problem is not legal. In such a case, the problem is called *cooked*.

In addition, there are few requirements of quality:

- 1) There should be just one unique way for White in every step of the solution. Theoretically, the problem is *ideal* if all lines of play are unique. However, if Black does a weak move, White is allowed to have more than one possible solution. In such a case, a problem is known to have *duals* if there are two lines of play, *triples* if there are three, and *multiples* if there are more. *Duals*, *triples*, and *multiples* are usually referred to as *duals*. Such *duals* decrease the quality of the problem. This can be avoided by rearranging of pieces placement or by adding a piece. There is a trade off and a composer usually allows duals which are forced in other lines of play (so-called minor duals).
- 2) The number of the pieces on the board should be no less and no more than is needed for the theme(s) included in the problem. This kind of quality measure is called *Economy of Piece or Material* (Harley 1931). This approach is different from the old approach which assumed that more pieces on the board would make the solution harder. If there are

2. *BACKGROUND*

different choices for a White piece on a square, the weakest piece should be chosen. The opposite choice is taken for Black, i.e., the strongest piece should be placed.

- 3) *Key or keymove* is the first move of White. A good key decreases White's options and increases Black's options. A good key is neither a check nor a piece capture. However, sometimes it is allowed.
- 4) *A Theme* is a special important structure or a special way of solving a problem.
- 5) The composed position must be new and original. Building global database is a way of checking whether the position is not unique. However, it is not enough, because minor changes in pieces, living similar ideas on the board, are also considered as plagiarism.

2.2.2. Composition Themes

Modern chess problems represent various themes. According to different composition books (Harley (1931, 1944), Howard (1943)), themes may be different and depend only on the fantasy of the composers or the editors of chess composition journals. Composition themes are patterns by which a composer may judge about the quality of the problem. The more such patterns can be recognized by a composer, the higher the quality of the problem is. There are thousands of such patterns.

HaCohen-Kerner et al. (1999) built a composing model. Their model, called ICP, applied on several important themes in chess composition (see Appendix B) with the help of two international masters of chess composition.

We started off with the same themes for our current research. However, a different approach was used. ICP defines themes observing given positions. The real themes are defined on the line of plays, from the given position to the mating positions. Therefore, ICP's definitions give just a direction of its heuristic function. The new definitions of the same

2. BACKGROUND

themes are more exact, because the whole lines of play are used to define themes. All 11 themes used in this thesis for 2-movers are presented in Appendix C.

2.3. Chess Problems and Computer Chess Programs

There are several works on computer chess problems. Below we introduce different tools and methods for taking care of chess problems.

2.3.1. Problem Solving

Most computer programs dealing with chess problems are dedicated to problem solving. The performance of such programs is described in Lindner (1985, 1989, 1991). He presents tools that, in his opinion, a chess problem-solver software should supply. Examples for such programs are Leschemelle's "Problemist" [il-3], Blom's "Alybadix" ("Metabadix") [il-1], "Popeye" by Schnoebelen et al. [il-5], and Nowakowski's "Chess Explorer" [il-6]. The main disadvantage in Lindner's papers is the lack of algorithms.

Another problem solving program is VKSACH by Kotěšovec (1996). Like other programs, VKSACH checks the correctness of problems. In addition, it composes problems by itself. However, its main concern is helpmates (definition 2.02). The author describes some sub-composition algorithms.

When a simple brute force method is applied to problems in N -moves ($N > 5$), it could take sometimes unpractical amount of time. Some authors use different methods to deal with this problem. Conspiracy numbers and proof numbers are two such methods.

2. BACKGROUND

2.3.2. Conspiracy Numbers

The term "Conspiracy numbers" references to a best-first search algorithm firstly proposed by McAllester (1988). Schaeffer (1990) implemented this method to different tactical chess problems. The idea of the method is to expand selectively terminal nodes considering the conspiracy number for each possible root value. A conspiracy number is a minimal number of terminal nodes ("leaves" in a sub-tree) that changing their values changes the root score (the initial position). The gain is that the algorithm searches deeper in those sub-trees that are more likely, probability-wise, to have a better solution and searches less in unlikely sub-trees. The algorithm is not limited by depth as alpha-beta. Its drawbacks are: (1) The possibility of sometimes searching to ridiculous depths in a "wrong" direction, (2) the algorithm does not guarantee to find the best solution, and (3) possible the memory explosion, because the algorithm must store all developed nodes with their numbers.

2.3.3. Proof Numbers

Proof numbers can be viewed as an implementation of conspiracy numbers for AND/OR trees. This idea was presented in various papers (Allis (1994), Allis et al. (1994), and Breuker et al. (1994)). AND/OR trees are special graphs, particularly trees, where every node is an AND-node or an OR-node. In the case of a game tree, White (or MAX) has OR nodes, while Black (or MIN) has AND nodes. The goal is to "prove" (resolve) the tree, i.e., to "solve" (evaluate) the root node, i.e., to find a mate solution. We intend to answer the question whether there is a mate. If there is a mate, then the tree is proved. Otherwise, the tree is disproved. Terminal nodes are "true", "false", and "unknown". A proof number is a number that for each node gives an answer to the questions "how many terminal nodes in the sub-tree must become true in order to prove the node" and "how many terminal nodes in the

2. *BACKGROUND*

sub-tree must become false in order to disprove the node". This selective best-first algorithm chooses the most-proving node (maximizing the "how many"s) to develop at each step. Seo's PN* (Seo et al. 2001) is an improvement of the proof numbers algorithm.

2.4. Endgames

In chess, the term "opening" means the start of the game. "Endgame" means the end of a game where just a few pieces are left on the board. Strong chess playing programs have special big databases for openings and endgames. The volume of the databases changes from programmer to another. Openings usually are written with the help of chess experts. Practical known endgame databases were built for endgames with three, four, five, and sometimes six pieces. These databases are used by many strong playing chess programs.

There are 2 types of databases (sometimes known also as tablebases), characterized by distance to mate (DTM) and distance to conversion (DTC). In the first case (DTM), the shortest possible mate for each position is stored. In the second case (DTC), the shortest number of plies between each position and a "conversion" is stored. By conversion, a piece capturing, a pawn promotion, or a checkmate are considered.

Thompson (1986, 1996) used distance to conversion (capturing a piece). It is hard to use this database in a chess engine when it is compressed (Heinz 1999). The actual result of Thomson's database is a set of four regular CDRoms (each of 640 Mbytes).

Thompson, one of the originators of UNIX, proposed the idea of Retrograde Analysis (RA) for chess endgames. The idea is to select end positions (for example, mate positions) and go as far back as possible, selecting scores of all the positions on the way back (Thompson (1986, 1996)).

Edwards' so-called tablebases use DTM (Heinz (1999)). Nalimov's tablebases are an improvement to Edwards' tablebase. The improvement is achieved by compression.

2. *BACKGROUND*

Nalimov's tablebases are considered better than Edward's because they are relatively small, i.e., about 7.5 GBytes instead of Edward's 30 GBytes.

Heinz (1999) explains why Edwards' so-called tablebases are better than Thompson's databases. Thompson's databases are not so easy to use in chess-playing programs and are relatively slow. Nalimov's tablebases are an improvement of Edwards' tablebases. They are known for their space efficiency and are in use as a standard in current chess playing programs.

2.5. Composing of Chess and Chess-like Problems

There had been limited research on computer chess composing. Some methods are shown below. Some methods in chess-like games are presented as well.

2.5.1. Composition of Two-Move Mate Problems

The concept “high-quality chess mate problem” is hard to define, especially for an automatic program. It is not simple to define concepts, such as beauty, originality, uniqueness of the solution, and difficulty of solving process. Therefore, a major part of the knowledge needed for evaluating the quality of chess problems, in general, and of two-move problems, in particular, was defined in a model called an Improver of Chess Problems (ICP) (HaCohen-Kerner et al. 1999). This knowledge was collected with the help of two international masters in chess problems composition. It includes definitions of themes (Table 9 in Appendix B), bonuses (Table 10 in Appendix B), and penalties (Table 11 in Appendix B) in the domain of chess composition for two-move problems.

ICP tries to improve the quality of a given problem by a series of special transformations. These transformations are divided into four main classes:

2. BACKGROUND

1) Simple transformations – deletions and additions of a piece on the board. The idea behind deletions is to express the same ideas using the smallest number of pieces possible. The idea behind additions is trying to add new themes or trying to prevent duals. Harley (1931) defines these ideas as tradeoff between *Economy of Play* and *Economy of Force*. So-called English composers' school prefers *Economy of Play*. In contrast, the Bohemian school prefers *Economy of Force*.

2) Stereotypical-agent transformations – replacing one type of piece with another. In chess composing, Black should use the strongest pieces; White should use the weakest pieces. Thus, a Black bishop becomes a Black queen and a White queen becomes a White rook if it is still a 2-mover.

3) Stereotypical-area transformations – exchanging between ranks or between files. ICP implements moving a piece as this kind of transformations. The idea is to take sliding pieces (bishops, rooks, and queens) as far as possible.

4) Transparency transformations – moving all pieces toward the center. The idea is to put the Black king in the center, which is regarded as a small improvement, because it is usually harder to mate the Black king there.

ICP uses a hill-climbing search. It means that the best son was chosen for development at each step. For each newly generated position the score was calculated using the quality function described in Appendix B. Thus, at the end of the process we achieve local maxima. This is the reason why only slight improvements were found by ICP.

ICP improved the quality of only 10 out of 36 known two-move mate problems (about 28%). Most of the achieved improvements are considered as slight ones.

2.5.2. A Global Scheme for Problems Composing

In contrast to ICP, Shlosser's (1988, 1991) main goal was to construct problems in as many moves as possible. He was probably the first to propose a general method of using a retrograde analysis for chess problems composition. His method includes three main stages:

- 1) Constructing a complete database of new problems starting from given mating positions, in a similar way to the construction of endgames by Thompson (1986, 1996), using a retrograde analysis.
- 2) Eliminating all incorrect positions according to chess composition rules positions.
- 3) Selecting high-quality chess problems based on evaluation values given by a human chess expert.

The differences between ICP and Shlosser's model are as follows:

- 1) ICP, practically, deals with any number of pieces on the board, while Shlosser's model examines the problems with just a few pieces.
- 2) ICP uses a move-forward hill-climbing technique for the given problem instead of using a move-backward retrograde analysis.
- 3) ICP starts with 2-movers and ends with them. There is no change in the number of the moves, while in Shlosser's model we start in 1-mover and can finish with 32-mover.
- 4) ICP uses a quality function instead of a proposed by Schlosser human expert in order to evaluate many new problems automatically and more quickly.

Retrograde analysis is useful to create problems in many moves. Haworth (2000) achieved a KRNKNN problem (White has king, rook and knight; Black has king and two knights) with mate in 243 moves and Thomson (2000) achieved a KRNKNN problem in 262

2. BACKGROUND

moves. The only quality of problems such as these two is the length of the solution. ICP looks for quality of 2-movers, defined by international masters of chess composition.

2.5.3. Composing in Tzume-Shogi

Shogi is a popular chess-like game in Japan [il-6]. It has different rules. It is played on a 9x9 board, some of the pieces are the same as in chess, but some are different. The first player that moves is called Black. Pieces can be promoted as in chess but in a different way. There is an ability to reuse captured pieces. The average branching factor of Shogi is 80 rather than 35 in the traditional chess. An average game lasts more time than a usual chess game. Other differences can be found in Matsubara and Grimbergen (1997). A detailed description of Shogi's rules can be found in Leggett (1996).

Tsume-Shogi is a composition problem in Shogi, where the opposite king (there is no White or Black) must be mated in a given number of moves. There are a few works that discuss composing of new Tsume-Shogi problems.

Noshita (1991) presents a random-generation approach for Tsume-Shogi. Using his method many positions are generated at random and the positions (positions-to-mate) that lead to mate positions (position-in-mate) are chosen. Finally, the positions are reduced into other positions with fewer pieces but with longer sequences of moves. Using this approach, Noshita has composed problems in 13-19 moves ("moves" means "plies" in Shogi's terminology while in chess a move means 2 plies). This method is called algorithmic-generation by its inventor.

Hirose et al. (1997) use a reverse method which is actually an extension of the retrograde method of Thompson (1986, 1996). The whole process includes 4 steps:

2. BACKGROUND

1) Generating of a mating position. It is done by generating a mating position with a few pieces and adding about 1 to 3 attacking pieces at distance of 1 to 3 from the opposite king.

2) Applying a reverse method up to the N reversed plies from the selected mating position in step 1. The generated positions are tested for being problems in n -moves.

3) Optimization. All useless pieces are removed. It is done by testing whether we still have a problem in N -moves.

4) Evaluation of the remaining positions. The evaluating function uses 13 factors for estimating a value of a problem. Not all these 13 factors are described by Hirose et al. (1997). The main parameters of the evaluation function is described as follows, citing:

1) The number of the attacking-side moves in which the moved or dropped pieces are taken immediately by the defending-side (this is the most important).

2) The number of the attacking-side moves in which the moved pieces are not promoted.

3) The number of the attacking-side moves which take the defending-side pieces.

4) The depth and the breadth of the search space.

5) Position of opponent's king.

6) The number of pieces on the board.

(end of citation).

From among the problems the best valued problem is chosen. Some of the composed problems were published, under pseudonym, in Tsume-Shogi composition journals.

Watanabe (1999) also implements similar ideas for composing problems in Tsume-Shogi. Watanabe et al. (2000) conclude the ideas of the reverse methods and the algorithmic-generation methods combining them into a new computer composition approach. He claims that this combined method is suitable for all chess-like games.

3. DICP, 2-move Chess Composer, and K -move Chess Composer Models

In order to find the best improvement(s) for each mate problem (if exists), we investigate all possible similar positions. Therefore, we use brute-force search to generate these positions. All regular brute-force search methods have a time complexity of $O(b^d)$, where b represents the branching factor and d represents the depth of the generated positions tree (Russell et al. 2002). The depth is the number of applied changes/transformations starting from an original mate problem. In our domain, b is all applied transformations rather than all legal chess moves. The estimated average branching factor in the chess game is about 35 while our branching factor is 368 on average (see Table 1, subsection 3.3).

We developed 2 models. *DICP* was the first model that was capable of applying transformations up to three levels. We ran *DICP* on the same set of 36 mate problems as ICP (see results in subsection 4.1). The *Chess Composer* model is an improvement to *DICP*. It is capable of running some problems up to four levels (see results in subsection 4.2).

3.1. General Description

Chess Composer uses the Depth First Iterative Deepening (DFID) search algorithm described by Korf (1985). Korf proved this scheme to be asymptotically optimal among all brute force methods. The total time for nodes to be expanded is $O(b^d)$, where b is the branching factor, and d is the depth of the developed tree.

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

In practice, observing a small set of problems, we set the depth to 3, since it takes on average about 32 minutes for a problem at this level, while it takes about 50 hours at level 4.

A simple Bounded DFS with order on transformations was applied to the set of 36 problems (see Figure 1). It is slightly preferable to DFID, used in *Chess Composer*, because DFID investigates inner nodes a couple of times. The algorithm is self-explanatory. Its functions are the same as in the *Chess Composer's* algorithm which is presented later.

1) BoundedDepthFirstSearch (OriginalProblem , MaxDepth)

1. PushIntoStack ({OriginalProblem, 0}) // 2nd parameter is for the node's level in the tree
2. While NotEmptyStack ()
 - 2.1. {CurrentPosition, Level} ← PopFromStack()
 - 2.2. if ImprovedMateProblem (CurrentPosition, OriginalProblem) then
 - 2.2.1. StoreImprovedMateProblem (CurrentPosition)
 - 2.3. if (Level < MaxDepth) then
 - 2.3.1. For each Successor of CurrentPosition (from right to left)
 - 2.3.1.1. CurrentPosition = ApplyNextTransform (CurrentPosition)
 - 2.3.1.2. PushIntoStack({CurrentPosition, Level + 1})

2) ImprovedMateProblem (CurrentPosition, OriginalProblem)

if (LegalChessPosition (CurrentPosition) and
LegalTwoMoveMateProblem (CurrentPosition) and // see page 28 for details
ProblemEvaluator (CurrentPosition) > ProblemEvaluator (OriginalProblem))
then return true
else return false

Figure 1: Simple Bounded DFS used in DICP

In *Chess Composer*, the algorithm was changed. Asymptotically, DFID is equal to the bounded DFS. The problem of inner nodes was decreased to the minimum by developing a better algorithm. The general algorithm is described in Figure 2. The improvement is due to the function *Iterate* explained below.

3. *DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS*

We tested an extended database of 100 problems including the 36 problems mentioned before. For problems with no improvement after a sequence of three transformations we seek one level deeper using the iterative deepening scheme. The results in subsection 4.2 show that *Chess Composer* improves problems that *DICP* could not improve.

Composer is the main function. It gets as input *MaxDepth* which is equal to 3. It then *iterates* to depth 1, 2, and 3. If no improvement was found, it iterates to depth 4.

Iterate is a bounded DFS function. It generates all nodes until *Composer's* bound (level 3). In order not to reapply the complex and time-costly function *ImprovedMateProblem* (see explanation below), the algorithm calls this function only for nodes at the bottom level. By this, we gain some improvement in inner nodes and reduce the main disadvantage of Iterative Deepening scheme which is duplicate expansion of inner nodes. The Iterative Deepening scheme, described in Korf (1985), expands inner nodes $MaxDepth - d$ times, $0 < d < MaxDepth$, while in our algorithm these nodes are fully expanded only once. Of course, the number of generated nodes is the same in both algorithms. However, it is less important because the cost of generating a node is much less than the cost of its expansion.

ImprovedMateProblem checks whether a new position is an improved 2-move chess mate problem. It is a very costly function because it calls three functions: *LegalChessPosition*, *LegalTwoMoveMateProblem*, and *ProblemEvaluator*. The last function is the most involved and time-consuming because it checks whether a new position contains any themes, bonuses, and penalties.

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

1) Composer(OriginalProblem, MaxDepth)

1. ImprovedList = NULL
2. For (I = 1; I <= MaxDepth; I \leftarrow I + 1)
 - 2.1. Iterate (OriginalProblem, I)
3. If isEmpty (ImprovedList)
 - 3.1. Iterate (OriginalProblem, MaxDepth + 1)
4. return bestScored problem from ImprovedList

2) Iterate (OriginalProblem, Bound)

3. PushIntoStack ({OriginalProblem, 0}) // 2nd parameter is for the node's level in the tree
4. While NotEmptyStack ()
 - 4.1. {CurrentPosition, Level} \leftarrow PopFromStack()
 - 4.2. if (Level == Bound) // in this case we investigate the CurrentPosition
 - 4.2.1. If ImprovedMateProblem (CurrentPosition, OriginalProblem) then
 - 4.2.1.1. ImprovedList \leftarrow StoreImprovedMateProblem (CurrentPosition)
 - 4.3. else // i.e.: if (Level < Bound) – in this case we do not investigate the CurrentPosition
 - 4.3.1. For each Successor of CurrentPosition // from right to left
 - 4.3.1.1. CurrentPosition = ApplyNextTransformation (CurrentPosition)
 - 4.3.1.2. PushIntoStack ({CurrentPosition, Level + 1})

3) ImprovedMateProblem (CurrentPosition, OriginalProblem)

1. if (LegalChessPosition (CurrentPosition) and
LegalTwoMoveMateProblem (CurrentPosition) and
ProblemEvaluator (CurrentPosition) > ProblemEvaluator (OriginalProblem))
then return true
else return false

Figure 2: Optimized Depth First Iterative Deepening Composer, used in Chess Composer

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

Other components:

- LegalChessPosition: Checks the legality of a position according to the chess rules.
- LegalTwoMoveMateProblem: Tests whether a given position is a two-move mate problem (including the test for non-cooking. i.e., no more than one keymove). This component uses a suitable limited search engine and checks all legal chess moves including the two special moves (in contrast to ICP): *castling* and *en passant* capture.
- ProblemEvaluator: Analyzes a position as a mate problem and computes its quality score. This function is described in the next sub-section.
- ApplyNextTransformation: Applies the next transformation on a given position according to a fixed ordered list of transformations.

Chess Composer uses three kinds of transformations while attempting to improve a problem (the *ApplyNextTransformation* function):

- (1) deletion of a specific piece
- (2) addition of a specific piece
- (3) transparency of all pieces

The last one is done through two possible movements:

- (1) file-transparency: All pieces are transferred I files to the right or to the left,
and
- (2) rank-transparency: All pieces are transferred J ranks to the up or down.

Additional possible transformations (e.g., moving a certain piece, exchanging a piece) are introduced in Kerner (1995) and HaCohen-Kerner et al. (1999). These transformations can be seen as complex transformations based on the basic transformations of deletions and additions. That is, every transformation can be presented by suitable deletion(s) and/or addition(s).

3.2. Heuristic Quality Function

The heuristic function that computes the quality score of a position is defined as in ICP (HaCohen-Kerner et al. 1999) as follows:

$$q_m = \begin{cases} 0 & \text{illegal} \\ \sum_i V(T_i) + \sum_j V(B_j) - \sum_k V(P_k) & \text{legal 2-mover} \end{cases}$$

where q_m is the quality score. A position is illegal when:

- (1) It is not legal by the chess rules (see Appendix A)
- (2) It is not a two-move mate problem (see subsection 2.2.1)
- (3) There is more than one keymove (cooked)

Let V be the value function, T_i is the improved set of all themes (Appendix C) included in the position, B_j is the set of all bonuses (Table 10 in Appendix B) granted to the position and P_k is the set of all penalties (Table 11 in Appendix B) associated with the position. Various themes, bonuses, and penalties were collected consulting two international masters in chess composition and taken from various literature including the classic books of chess composition (Harley 1931, Howard 1943). The evaluation function, as it was presented at HaCohen-Kerner et al. (1999), is given in Appendix B. We represent it with some improvements (Appendix C) because: (1) some definitions have been revised and (2) for the convenience of the reader.

We define and apply eleven composition themes: Self-blocking, self-pinning, unpinning, half-pinning, direct battery, indirect battery, Grimshaw, pickaniny, king-flights, lonely Black king and tempo. Each theme has its own unique score given by international masters of chess composition depending on its complexity and relative importance. Definitions of these themes are given in Appendix A and with revision in Appendix C.

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

Bonuses and penalties are additional tools for evaluating a problem. An example of a bonus is a placement of the Black king in the center of the board. A Black king in the center is supposed to be better because it is harder to mate him. On the other side, if the Black king is at an edge or at a corner, the position gets a penalty.

3.3. Complexity of DICP and Chess Composer

Chess composition rules do not allow a chess problem to contain more than one king, one queen, two rooks, two bishops, two knights, and eight pawns for each color (Harley 1931, Howard 1943). Another rule is that bishops of the same side have to be on differently colored squares. In this model, we keep to these rules. However, in order to estimate the complexity, we assume, in contrast to chess composition rules, that each one of the p pieces can be added at each step on an empty square.

Let p be the number of pieces we can add and let s be the number of empty squares. Assuming d to be the number of additions on this additions-tree, the number of nodes on level d by naïve additions is:

$$ps * p(s-1) * p(s-2) * \dots * p(s-d+1) = p^d * \frac{s!}{(s-d)!} \quad (1)$$

For instance, the 1-level enables $p*s$ additions, i.e., each kind of piece on each empty square. The 2-level enables $p*(s-1)$ additions, because the number of the empty squares was reduced by one. Obviously, many positions are repeated by permutations of additions. For example, a White bishop can be added before a Black knight and vice-versa.

To prevent these repetitions, we apply an order on the added pieces as follows: Q (White queen) > R (White rook) > B (White bishop) > N (White knight) > P (White pawn) > q (Black queen) > r (Black rook) > b (Black bishop) > n (Black knight) > p (Black pawn),

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

which means that if we have added a piece from the list (say N), then we cannot add a piece which precedes it in the list (Q, R, B).

Applying this order we get the following formula for the number of nodes at level d of the additions-tree without repetition, assuming d to be the number of the additions:

$$\frac{s!}{(s-d)!} \sum_{i_{d-1}=1}^{i_d=p} \sum_{i_{d-2}=1}^{i_{d-1}} \sum_{i_{d-3}=1}^{i_{d-2}} \dots \sum_{i_2=1}^{i_3} \sum_{i_1=1}^{i_2} \sum_{i_0=1}^{i_1} 1 \quad (2)$$

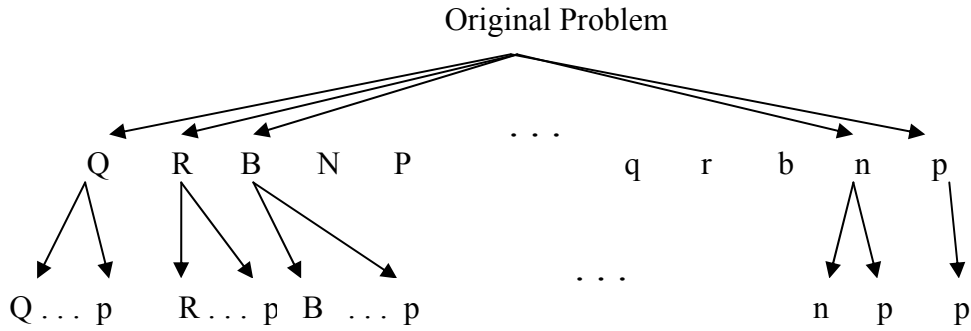


Figure 3: The additions-tree up to depth $d = 2$

Figure 3 presents the tree until depth $d = 2$. The number of possible additions is: 10 (for [Q ...p]) + 9 (for [R ...p]) + 8 (for [B ...p]) + ... 1(p) = 55, exactly as expressed in formula (2). The implementation of the order on additions is quite simple (Appendix F).

Expression (2) leads to the following equation.

Claim:

$$\frac{s!}{(s-d)!} \sum_{i_{d-1}=1}^{i_d=p} \sum_{i_{d-2}=1}^{i_{d-1}} \sum_{i_{d-3}=1}^{i_{d-2}} \dots \sum_{i_2=1}^{i_3} \sum_{i_1=1}^{i_2} \sum_{i_0=1}^{i_1} 1 = \frac{s! \prod_{i=1}^d (p+i-1)}{(s-d)! d!} \quad (3)$$

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

Proof (by induction on d):

$$d=1, \quad \frac{s!}{(s-1)!} \sum_{i_0=1}^{i_1=p} 1 = sp = \frac{s! \prod_{i=1}^1 (p+i-1)}{(s-1)!1!}$$

d=2,

$$\frac{s!}{(s-2)!} \sum_{i_1=1}^{i_2=p} \sum_{i_0=1}^{i_1} 1 = s(s-1)(1+\dots+p) = s(s-1) \frac{p(p+1)}{2} = \frac{s! \prod_{i=1}^2 (p+i-1)}{(s-2)!2!}$$

d=3,

$$\begin{aligned} \frac{s!}{(s-3)!} \sum_{i_2=1}^{i_3=p} \sum_{i_1=1}^{i_2} \sum_{i_0=1}^{i_1} 1 &= s(s-1)(s-2) * \sum_{i_2=1}^p \frac{i_2(i_2+1)}{2} = \\ \frac{s(s-1)(s-2)}{2} \left(\sum_{i_2=1}^p i_2^2 + \sum_{i_2=1}^p i_2 \right) &= \\ \frac{s(s-1)(s-2)}{2} * \left(\frac{p(p+1)(2p+1)}{6} + \frac{p(p+1)}{2} \right) &= \\ s(s-1)(s-2) * \frac{p(p+1)(p+2)}{6} &= \frac{s! \prod_{i=1}^3 (p+i-1)}{(s-3)!3!} \end{aligned}$$

Base of induction, d=k:

$$\frac{s!}{(s-k)!} \sum_{i_{k-1}=1}^{i_k=p} \sum_{i_{k-2}=1}^{i_{k-1}} \sum_{i_{k-3}=1}^{i_{k-2}} \dots \sum_{i_2=1}^{i_3} \sum_{i_1=1}^{i_2} \sum_{i_0=1}^{i_1} 1 = \frac{s! \prod_{i=1}^k (p+i-1)}{(s-k)!k!}$$

Or

$$\sum_{i_{k-1}=1}^{i_k=p} \sum_{i_{k-2}=1}^{i_{k-1}} \sum_{i_{k-3}=1}^{i_{k-2}} \dots \sum_{i_2=1}^{i_3} \sum_{i_1=1}^{i_2} \sum_{i_0=1}^{i_1} 1 = \frac{\prod_{i=1}^k (p+i-1)}{k!} \quad (4)$$

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

We need to prove:

$$\sum_{i_k=1}^{i_{k+1}=p} \sum_{i_{k-1}=1}^{i_k} \sum_{i_{k-2}=1}^{i_{k-1}} \sum_{i_{k-3}=1}^{i_{k-2}} \dots \sum_{i_2=1}^{i_3} \sum_{i_1=1}^{i_2} \sum_{i_0=1}^{i_1} 1 = \frac{\prod_{i=1}^{k+1} (p+i-1)}{(k+1)!}$$

Let us define "t" for "trivial".

$$\begin{aligned} & \sum_{i_k=1}^{i_{k+1}=p} \sum_{i_{k-1}=1}^{i_k} \sum_{i_{k-2}=1}^{i_{k-1}} \sum_{i_{k-3}=1}^{i_{k-2}} \dots \sum_{i_2=1}^{i_3} \sum_{i_1=1}^{i_2} \sum_{i_0=1}^{i_1} 1 = \\ & \sum_{i_k=1}^{i_{k+1}=p} \frac{\prod_{i=1}^k (i_k + i - 1)}{k!} \stackrel{\text{taking the 2 first terms}}{=} \frac{\sum_{i_k=1}^{i_{k+1}=p} \prod_{i=1}^k (i_k + i - 1)}{k!} = \frac{\sum_{j=0}^{p-1} \prod_{i=1}^k (i + j)}{k!} = \\ & \frac{\prod_{i=1}^k i + \prod_{i=2}^{k+1} i + \sum_{j=2}^{p-1} \prod_{i=1}^k (i + j)}{k!} \stackrel{\text{taking the 2 first terms}}{=} \frac{(\frac{k+2}{k+1}) \prod_{i=2}^{k+1} i + \sum_{j=2}^{p-1} \prod_{i=1}^k (i + j)}{k!} = \\ & \frac{(\frac{1}{k+1}) \prod_{i=2}^{k+2} i + \sum_{j=2}^{p-1} \prod_{i=1}^k (i + j)}{k!} \stackrel{\text{taking the third term}}{=} \frac{(\frac{1}{k+1}) \prod_{i=2}^{k+2} i + \prod_{i=3}^{k+2} i + \sum_{j=3}^{p-1} \prod_{i=1}^k (i + j)}{k!} \stackrel{\frac{1}{k+1} * 2 + 1 = \frac{k+3}{k+1}}{=} \\ & \frac{(\frac{k+3}{k+1}) \prod_{i=3}^{k+2} i + \sum_{j=3}^{p-1} \prod_{i=1}^k (i + j)}{k!} \stackrel{\text{taking the third term}}{=} \frac{(\frac{1}{k+1}) \prod_{i=3}^{k+3} i + \prod_{i=4}^{k+3} i + \sum_{j=4}^{p-1} \prod_{i=1}^k (i + j)}{k!} \stackrel{\frac{1}{k+1} * 3 + 1 = \frac{k+4}{k+1}}{=} \\ & \frac{(\frac{k+4}{k+1}) \prod_{i=4}^{k+3} i + \sum_{j=4}^{p-1} \prod_{i=1}^k (i + j)}{k!} \stackrel{\text{continuing}}{=} \dots \\ & \frac{(\frac{k+p-1}{k+1}) \prod_{i=p-1}^{k+p-2} i + \sum_{j=p-1}^{p-1} \prod_{i=1}^k (i + j)}{k!} \stackrel{\text{reindexing}}{=} \frac{(\frac{1}{k+1}) \prod_{i=p-1}^{k+p-1} i + \prod_{i=p}^{k+p-1} i}{k!} \stackrel{\frac{1}{k+1} * (p-1) + 1 = \frac{k+4}{k+1}}{=} \\ & \frac{(\frac{p-1+k+1}{k+1}) \prod_{i=p}^{k+p-1} i}{k!} \stackrel{\text{trivial}}{=} \frac{\prod_{i=p}^{k+p} i}{(k+1)!} \stackrel{\text{trivial}}{=} \frac{\prod_{i=1}^{k+1} (p+i-1)}{(k+1)!} \end{aligned}$$

□

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

The ratio between Eq. (3) and Eq. (1) suggests significant asymptotical improvement compared with formula (1). That is, our special order of additions improves the order of our generation algorithm.

$$\frac{(3)}{(1)} = \frac{\prod_{i=1}^d (p+i-1)}{p^d d!} = \frac{\prod_{i=1}^d (p+i-1)}{\prod_{i=1}^d p^i} = \prod_{i=1}^d \frac{p+i-1}{p^i} = \prod_{i=1}^d \left(\frac{1}{i} + \frac{1}{p} - \frac{1}{p^i} \right) \xrightarrow{d \rightarrow \infty} \frac{1}{p^\infty} \rightarrow 0 \quad (5)$$

Taking into consideration also the two other kinds of transformations, deletions and transparencies, we can achieve the complete order, assuming that:

- (1) Deletions are always done before additions, and transparency is before deletions
- (2) In case of the addition of the same pieces, the order is due to the proximity of a piece to 0-square (h8 = 0; a8 = 7; h1 = 56; a1 = 63).
- (3) Ordering deleted pieces is also due to their proximity to 0 (i.e. to square h8).

To clarify, on the first transformation we use all transformations, e.g. transparency, deletions, additions. The second transformation depends on the first: supposing the first transformation was a deletion, we can choose only deletion of the rest pieces and additions on the second transformation. Thus, we can have a sequence deletion->deletion->addition, but never deletion->addition->deletion.

Having applied all orders, we get a complete order on the transformations, and the algorithm becomes $d!$ times faster compared with the naïve algorithm (Formula 1) at the expenses of applying the order on transformations which is simple (see Appendix F).

Deletions become relatively important when we treat positions with many pieces. This is because of the already mentioned chess composition rule about number of pieces: there

3. *DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS*

cannot be more than 8 pawns, 2 bishops, 2 rooks, 2 knights, 1 queen, and 1 king for each color and two same colored bishops cannot be placed on squares with the same color.

Our theory is supported by experimental results. Each chain containing the same d transformations leads to the same position. That is, the same position can be reached in $d!$ different permutations. Thus, at level d we need to explore only $O(b^d/d!)$ nodes. Expression (3) shows us that the order on additions only is efficient enough. It is reasonable because additions contribute to branching factor much more than other transformations.

The results presented in Table 1 support this theory. The theoretical average is always in practical average range. If we consider each problem, the theoretical and the practical values are not the same. There are few reasons:

- 1) Theoretical calculations do not take into account the rule described at the previous paragraph about maximal number of pieces. Therefore, we can add fewer pieces, than we expect to add theoretically, and the actual average becomes lower.
- 2) Theoretical calculations do not take into account the fact that we can add more if we removed a piece and delete more if we added a piece. Thus, practically generated nodes can be more than expected.
- 3) Theoretical calculations do not take into account the fact that we can add less if we added a piece and delete less if we removed a piece. Thus, practically generated nodes can be less than expected.

Because there are many more additions than deletions on the average, the theoretical value is usually higher. It happens in 99 positions of 100 after 2 transformations and in all positions after 3 transformations. As expected, the only positions in which the theoretical value is higher than the actual are those with many pieces. What happened is, we cannot add a piece anymore, but we can delete many pieces on the first transformation. Then, we can add more than theoretically.

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

| | No. of Generated Positions for All 100 Problems | | | |
|----------------------------|--|------------------------|---------------------------|---|
| <i>d</i> | 1-level | 2-level | 3-level | 4-level (for only 6 problems) |
| Average # of nodes | 364 | 67,087 | 7,822,004 | 842,936,433 |
| Deviation (nodes) | 85 | 28,727 | 4,584,337 | 289,034,621 |
| Theoretical average | $364^1/1! =$ 364 | $364^2/2! =$ 66,248 | $364^3/3! =$ 8,038,091 | $364^4/4! = 731,466,251$ |
| Average time (sec.) | 0.16 | 20.73 | 1,881.74 | 168,272 |
| Deviation (sec.) | 0.07 | 7.21 | 782.64 | 57,785 |

Table 1: Average number of generated positions and running time by *Chess Composer* for all 100 original problems

In practice, there are hundreds (364 ± 85) of nodes developed after one transformation for an average problem, tens thousands of nodes ($67,087 \pm 28,727$) are developed after two transformations, millions of nodes are developed after three transformations ($7,822,004 \pm 4,584,337$), and almost trillion ($842,936,433 \pm 289,034,621$) nodes are developed after four transformations (Table 1). These values are highly comparable with theoretically calculated values, taking into account the above explanations about the limitations of the model. The running-time needed for investigating all generated nodes on the first 3 levels for an average problem is approximately 32 minutes and about 47 hours on the first 4 levels.

3.4. 64-bit Representation

Although our model implements a soft real-time system (i.e., we should give solution within either minutes or an hour) rather than a hard real-time (i.e., we must give an answer within seconds; there is a deadline after which very costly things can happen), the run-time is still important. Therefore, we speed up the model by using a 64-bit representation (see details in Appendix D).

Adelson-Velskiy et al. (1970) were the first to propose an approach of 64 bits for a board presentation applied in their chess program Kaissa. Other programmers used the same approach with success in their chess playing programs (Slate and Atkin, 1977; Heinz, 1997; Hyatt, 1999). The main ideas behind this representation are economy of loops, absence of checking whether a piece moves out of the board, and the ability to compute complex calculations just in a few bitwise operations (Adelson-Velskii et al., 1970; Slate and Atkin, 1977).

In this representation, each kind of piece has a unique 64-bit machine word (see Appendix D). Adelson-Velsky et al. (1970) used the M20, a Russian computer. According to them, the real length of the word in this machine was less than 64 bits, and a 64-bit word was only virtual. State and Atkin (1977) used CDC 6400 which also did not have 64-bit words. The use of 64 bits in nowadays computers becomes wider and wider. Heinz's Darkthought (1997) and Hyatt's Crafty (1999) use 64-bit machines.

In our experiments, we use two 64-bit machines: AMD64 and Itanium-2. The output of our move generator is an array of new positions. All positions in the array are legal, presupposing that the previous position is legal too (full details of our implementation are given in Appendix D).

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

Our move generator is fast enough compared to those of known programs. It gives a speed of more than 22 million positions per second if we develop all variants starting from the initial position (see Diagram 19 of Appendix A) using simple DFID. The two special chess moves, castling and en-passant, are also allowed.

Table 2 presents different runs of our move generator on different platforms with different configurations. To estimate speed, the initial chess position was developed up to depth 8. The speed is defined as the number of generated positions developed per run-time of the program. We use AMD64-3400+ running on linux 2.6.11 and compile with gcc 3.4.3. The known professional program *ChessExplorer* [il-4] (we have no results regarding other professional programs) uses a 32-bit approach and make the same with speed of about 15 million positions per second. Using the representation of 64 bits, the Chess Composer's (on AMD64 3400+) move generator is much faster (see Table 2). It must be also pointed that our move generator is not fully optimized. Therefore, its results are more impressive in comparison with 32-bits oriented programs.

| Pos./sec. | AMD64+ilogb | AMD64+bsf/bsr | Itanium2+ilogb |
|------------|-------------|-------------------|--------------------------|
| 'gcc -m64' | 18,401,251 | 22,163,976 | 4,873,584 |
| 'gcc -m32' | 7,109,440 | 7,445,146 | --- (no 32-bits support) |

Table 2: Generation of all possible positions from the initial chess position up to 8 plies with different configurations

Roughly speaking, 64-bit representation for a board uses a word for each kind of pieces. There are 12 such a words. Using bitwise operations, we can solve different questions in position analysis fast. One of the questions is a fast move generator. Usual 32-bit

3. *DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS*

representation does not use bitwise operations (see details of 64-bit representation in Appendix D).

The 'ilob' and 'bsf/bsr' in Table 2 mean two possible implementations for λ and μ (see full details in Appendix D). λ and μ are, respectively, the least and the most important bit that is on, following Adelson-Velskii et al. (1970). 'bsf/bsr' are two commands of the AMD's instructions set. They are, unfortunately, absent in Itanium2. '-m64' and '-m32' are two different compilation parameters for gcc, compiling into 64-bit platform and into 32-bit platform (we use the exactly same 64-bit structure but compile into 32-bit architecture; this is different from a usual representation used in Chess Explorer, for example).

Table 2 presents the following result. The best architecture for our program is AMD64. We can implement λ and μ using two special commands, 'bsf' and 'bsr'. If we compile with gcc -m64, we get 3 times faster results than if it would with gcc -m32. Itanium2 is not good for our generator. Finally, we can use existing 'ilob' command if we do not want any assembler support.

The main idea of rotated bitboards is to precalculate moves for the sliding pieces (Heinz 1997 and Hyatt 1999). The main difference between the generators based on rotated bitboards and our generator is the absence of use of rotated bits in our generator. The reason for this is the need to manage 3 additional bitboards with occupied pieces. "Looping" through different directions of sliding pieces is efficient enough, especially on a 64-bit architecture. However, we have a use in pre-calculated ranks for horizontal moves of rooks and queens (see "rm" in Table 17 of Appendix D).

3.5. Comparison to Other Models

Chess Composer, as opposed to ICP, uses only three kinds of transformations. Therefore: (1) the branching factor of *Chess Composer* is smaller and (2) the depth of the tree (the number of the applied transformations) developed by *Chess Composer* should be higher. The idea is that by combination of primitive transformations, i.e., deletions and additions, we can realize all other complex transformations. For instance, moving a piece from x to y is equal to its deletion from x and its following addition on y . Exchanging of a piece x with piece y is equal to deletion of piece x and the following addition of piece y on the same square. Nevertheless, a complex transformation such as transparency was added because it gives immediate improvement for problems in which it is possible to move all pieces in such a way that the Black king will be in the center. Transparency transformation does not affect the total complexity.

ICP uses a hill-climbing search based on a heuristic function (see Appendix B) to find a local maximum in the near-problem space. Hill-climbing search is an informed search. That is, it uses information of the newly developed nodes (Russell et al. 2002) in order to prune some of the transformation sub-trees.

In contrast, *Chess Composer* uses a brute force search. This is an uninformed search. That is, it does not depend on any information contained in the newly developed nodes. This approach overcomes the main weakness of ICP which is not developing the sub-trees whose root-positions are:

- 1) not legal according to the rules of chess composition.
- 2) not two-move mate problems with only one keymove.
- 3) of a lower quality score than the original problem.

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

Chess Composer develops all possible transformations on all levels (Table 1). That is, it traverses all nodes to a fixed depth in order to find the best improvement (global maximum). Of course, the complexity grows up, but this is exactly the non-trivial solution: using the slower method we are able to find much better positions, which ICP fails to find because of limitations of its heuristic function. As results show, most of the finest improvements were found after sequences of two and three transformations, while, if we use each independent transformation from these sequences, the best first method fails, because we get illegal positions.

The main differences between *Chess Composer* and Schlosser's model related to computer composition of chess and chess-like mate problems are:

1. *Chess Composer* uses a move-forward technique from the given problem instead of using either move-backward or random techniques.
2. *Chess Composer* tries all possible transformations for every examined position. Therefore, its branching factor is much higher.
3. *Chess Composer* starts with 2-movers and ends with them. There is no change in the number of the moves, in contrast to the model presented by Schlosser (1988, 1991).
4. *Chess Composer* deals with a relatively high number of composition themes which are treated automatically by a computer rather than by a human.
5. The evaluation is done automatically and not by a human expert as in Schlosser's model.

The differences between *Chess Composer* and ICP are:

1. *Chess Composer* tries all possible transformations for every examined position. Therefore, its branching factor is much higher.
2. *Chess Composer* uses brute force which allows finding such a problems that would not been found by the best-first of ICP.

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

3. The quality function in *Chess Composer* is different. The definitions of themes are more detailed (Appendixes B and C).
4. *Chess Composer* uses 64-bit approach which gives additional speed-up.

3.6. K-movers Model ($k > 2$)

K-movers are direct mate problems in which White mates Black in k moves. An example of a solution tree for a 3-mover is shown in Figure 4. Dashed lines represent White's moves. Solid lines represent Black's moves. There are five plies. The first ply is the keymove (1-2). It is always unique. When the keymove is not unique the problem is considered as cooked and it is canceled.

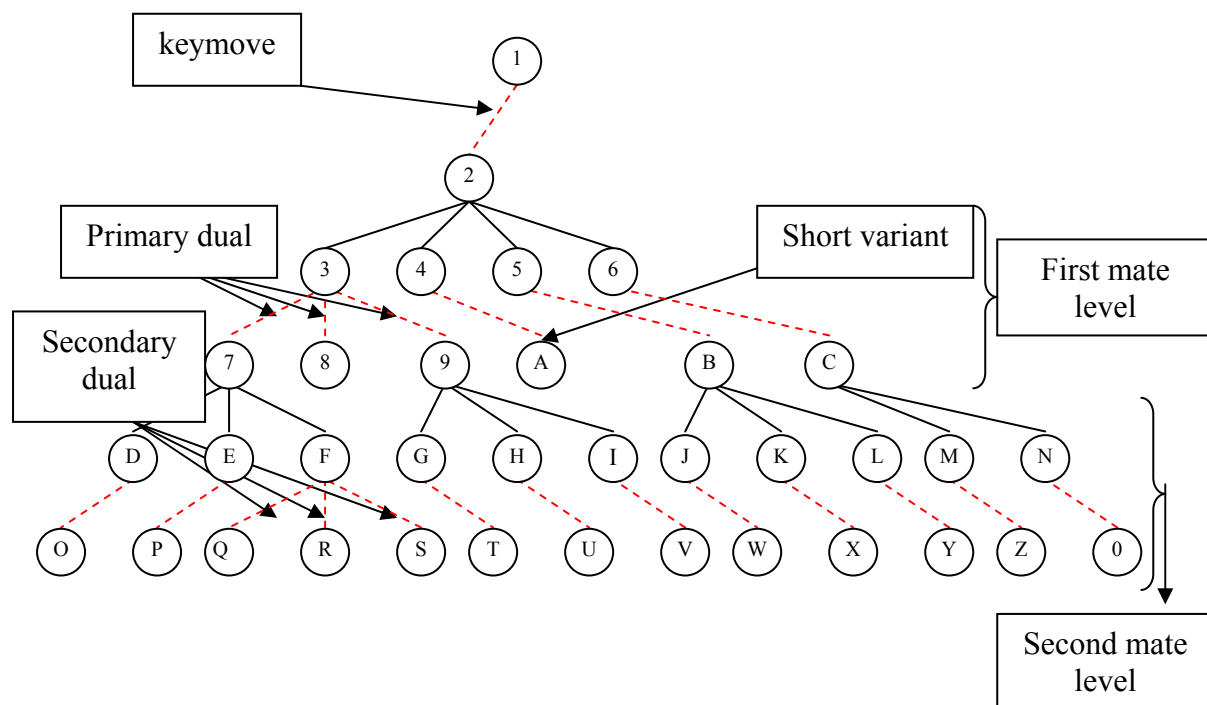


Figure 4: An example of a solution tree for 3-movers

Duals occur when there is more than one possible continuation for White. Duals are a significant flaw. When a dual occurs on the first ply, the problem is known as cooked. When

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

a dual occurs on the third ply, it is regarded as a "primary dual". When a dual occurs on the five ply, it is regarded as a "secondary dual". In a k -mover, we call duals by their ply, i.e. 3-ply dual, 5-ply dual etc. Examples of primary or 3-ply duals are represented by 3-7, 3-8, and 3-9. Examples of secondary or 5-ply duals are represented by F-R, F-S, and F-T. The deeper a dual occurs, the less serious is it ("cook" >> primary dual >> secondary dual >> 7-ply dual etc.)

A minor dual is a dual each variant of which is forced in other lines of play. For instance, if A=7, B=8, and C=9, than the primary dual is minor dual. Otherwise, it is a major dual. Major duals are considered as a more serious flaw than minor ones.

Short variants are another possible flaw of a problem. Examples for such variants are variants which end in "A" or in "8". The shorter a variant the more serious is the flaw.

So-called Black duals are another insignificant flaw. Black duals occur when two moves of Black, as a reply to White's move, have the same White's reply. For instance, if "B" and "C" or "O" and "P" are equal then it is a Black dual.

Definition 3.1. A "mate level" # n is defined as all pairs of moves, black-white, on plies $n+2$ and $n+3$ (see Figure 4 and 5).

Definition 3.2. A cluster is a set of nodes which includes a white-to-move node, all its sons, and all sons of the sons (nodes inside the dashed circle in Figure 5).

For example, there is only one sub-tree on mate level #1 which starts at "2" and ends at "C" (Figure 4). So, A, B, and C should be uniquely different and 7, 8, and 9 are not part of the calculations because they are a dual. There are 4 clusters on mate level #2. Their roots are 7, 9, B, and C. Therefore, there are 4 groups of different White moves in these clusters (duals are not counted): (1) O and P, (2) T, U, and V, (3) W, X, and Y, (4) Z and 0. We would like to get uniqueness of White moves in clusters of all mate levels.

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

Definition 3.3. A solution tree of an ideal k-move problem (see Figure 5) is defined as (based on discussions in Harley 1931, 1944):

- 1) It is a full tree. This is because "short" variants are considered as a flaw.
- 2) Nodes that represent positions with white-to-move have only one son. The reason is an absence of duals.
- 3) Nodes that represent positions with black-to-move have as many sons as possible. Sons of these sons are uniquely different between them. There are two reasons for that: (1) we would like to have as many variants as possible and (2) we do not want so-called Black duals.

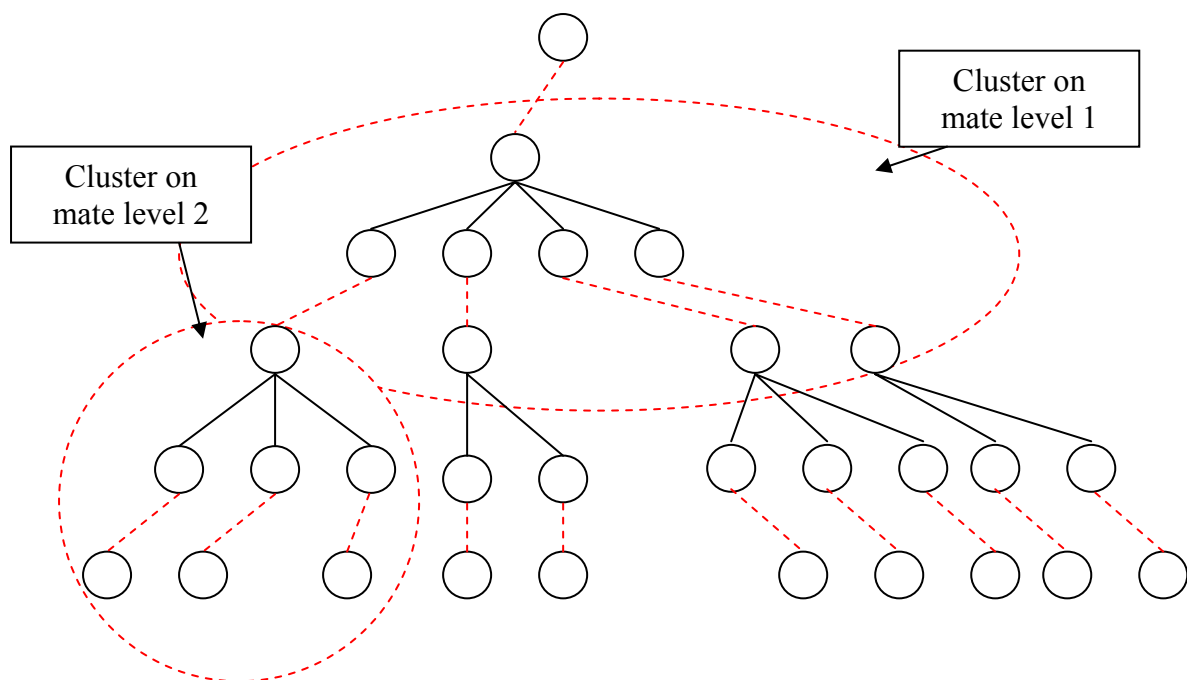


Figure 5: An example of an "ideal" tree for 3-movers

To improve a problem, we can try to change the solution tree towards an ideal tree. The following situations can happen while comparing an original problem to a new problem:

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

- 1) A variant (a sequence of black-white moves) was added to a cluster. If it adds an additional unique variant, it is good for us. If it adds a Black dual, it is bad for us. The opposite is right when a variant was removed.
- 2) A dual was added to a cluster. It is bad for us. The opposite is right when a dual variant was removed.
- 3) A short variant became longer. It is good for us. The opposite is bad.

The proposed metric for measuring k-move trees of problems in k moves is shown in Figure 6. The measuring is done by evaluating the whole solution tree of a certain problem.

$k - move_metric(k - movetree) =$

$$\left(\sum_{i \in \text{all clusters}} (cluster_score_i - duals_penalty_i) \right) + change_in_pieces,$$

where:

$$cluster_score_i = \frac{relative_cluster_coef * cluster's_level * \#unique_variants^2}{\#all_non_dual_variants}$$

$$duals_penalty_i = (relative_major_duals_coef * \#major_duals + relative_minor_duals_coef * \#minor_duals) * duals_level_score$$

$$change_in_pieces = \left(\sum_{\substack{i \in \text{White pieces} \\ \text{in the original position}}} value(i) - \sum_{\substack{i \in \text{Black pieces} \\ \text{in the original position}}} value(i) \right) -$$

$$\left(\sum_{\substack{i \in \text{White pieces} \\ \text{in the new position}}} value(i) - \sum_{\substack{i \in \text{Black pieces} \\ \text{in the new position}}} value(i) \right) * relative_pieces_value_coef$$

and a sub_tree spawn by duals (major or minor) costs 0

Figure 6: A metric for measuring k-move trees of problems in k-move

In this metric:

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

Change_in_pieces is the change in pieces value. The value of a particular piece is defined following Shannon's (1950) definitions (Table 12).

Relative_pieces_value_coef is a controllable coefficient which describes the importance of the change in pieces value.

cluster_score_i is the score that cluster #*i* achieve.

duals_penalty_i is the penalty for the duals occurred in the discussed cluster.

#unique_variants is the number of unique variants in the discussed cluster.

#all_non_dual_variants is the number of all variants the discussed cluster.

Cluster's_level is a cluster's score which depends on the mate level this cluster belongs to.

Relative_cluster_coef is the coefficient which transforms the variants-based value for the discussed cluster from range [0,1] to range [0, *Relative_cluster_coef*].

#major_duals is the number of major duals in the discussed cluster.

#minor_duals is the number of minor duals the discussed cluster.

Relative_major_duals_coef is the penalty for a major dual.

Relative_minor_duals_coef is the penalty for a minor dual.

Dual's_level_score is the dual's score which depends on the mate level this dual belongs to.

The meaning of this metric is as follows. If we add a variant to a cluster and it is different from other variants, the total score becomes higher due to the *cluster_score_i* parameter. The same parameter recognizes the reduction in variants too. If the added variant is not unique, the score becomes less. This way the problem of Black duals seems to be solved.

The formula for calculating a cluster score is defined in such a way that we get first a number between zero to one, by dividing the number of all unique variants by the number of all variants. This number is transformed into a range that was chosen due to

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

relative_cluster_coef from [0,1] to [0, *relative_cluster_coef*]. An additional bonus is given to the unique factor by taking the square of *#unique_variants*.

This metric prefers more clusters because of the additive nature of the formula, i.e. the more added clusters there are, the higher the total score. It contributes to the problem, because the whole problem becomes harder to solve due to many different variants.

Moreover, this metric prefers clusters on deeper levels to clusters on lower levels due to the *cluster's_level* parameter, meaning that if two clusters have the same score, but in the first situation the cluster is deeper, this situation is preferred and the deeper cluster achieves additional score according to *cluster's_level*. It contributes to the problem because of the two following reasons: (1) in a k-mover we prefer more variants in *k* moves; (2) this encourages short variants to lengthen or to be exchanged with longer variants.

In this metric, duals are not counted as well as the whole dual's subtree. The main reason is that we can get very high-scored clusters in the subtree and therefore the total score becomes higher. However, this situation is not good because duals are flaws and duals on the earlier levels are considered as worse.

The reason for the *duals_penalty_i* parameter is as follows. If a dual is cancelled in a new tree then the score of the cluster becomes higher. On the other hand, an added dual receives a penalty. This parameter depends on the type of the dual: major (score due to *Relative_major_duals_coef*) or minor (score due to *Relative_minor_duals_coef*) dual. *Dual's_level* is a parameter which distinguishes between duals that are nearest to root levels and those outlying from the root duals.

3.6.1. The K-movers' Improver Algorithm ($k > 2$)

The general algorithm for improving k-movers is shown in Figure 7. The general scheme of applying transformations is exactly the same as in the case of 2-movers in *Chess Composer* (see chapter 3.2). The difference is found in two functions: *K-Move_Solver* and *KMoveMetric*. *KmoveMetric* is implemented applying DFS scheme to the metric defined in Figure 6. *KMoveMetric* is shown in Figure 11 and is discussed later.

1) **K-Move_Composer**(OriginalProblem, MaxDepth)

// applies improved depth first iterative deepening to the original problem

1. ImprovedList = NULL
2. For (I = 1; I <= MaxDepth; I \leftarrow I + 1)
 - 2.1. Iterate (OriginalProblem, I)
3. If isEmpty (ImprovedList)
 - 3.1. Iterate (OriginalProblem, MaxDepth + 1)
4. return bestScored problem from ImprovedList

2) **Iterate (OriginalProblem, Bound)**

// uses Bounded DFS to develop and analyze all nodes to depth Bound

1. PushIntoStack ({OriginalProblem, 0})
 - // 2nd parameter is for the node's level in the tree
2. While NotEmptyStack ()
 - 2.1. {CurrentPosition, Level} \leftarrow PopFromStack()
 - 2.2. if (Level = Bound) then // in this case we investigate the CurrentPosition
 - 2.2.1. If ImprovedMateProblem (CurrentPosition, OriginalProblem) then
 - 2.2.1.1.ImprovedList \leftarrow StoreImprovedMateProblem (CurrentPosition)
 - 2.3. else // if (Level < Bound) – in this case we do not investigate the CurrentPosition
 - 2.3.1. For each Successor of CurrentPosition // from right to left
 - 2.3.1.1. CurrentPosition = ApplyNextTransformation (CurrentPosition)
 - 2.3.1.2. PushIntoStack ({CurrentPosition, Level + 1})

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

3) ImprovedMateProblem (CurrentPosition, OriginalProblem)

// finds if the current position is better than the original

1. if legal (CurrentProblem) then
 - 1.1. {has_solution, type } <- K-Move_Solver(CurrentProblem, k)
 - 1.2. if ([has_solution = true] and [type = (in_k_move and not cooked)] and
KMoveMetric (CurrentPosition) > KMoveMetric (OriginalProblem)) then
 - 1.2.1. return true
 - 1.3. else
 - 1.3.1. return false

Figure 7: The general algorithm for improving k-movers

The general algorithm for the *K-Move_Solver* function is shown in Figure 8. The purpose of this algorithm is to the answer whether or not a problem has a solution in k moves, is it or is it not cooked, and does it have a solution in less than k moves.

```
{has_solution, type } K-Move_Solver(OriginalProblem, k)
// solves a k-mover
1. NumOfPlies <- k*2-1
2. For ( I = 1; I < NumOfPlies; I ← I + 2)
  2.1. {has, cooked} <- has_solution_in_k (OriginalProblem, I)
  2.2. if has = true then
    2.2.1. if cooked = true then
      2.2.1.1. return {true, in_less_than_k & cooked}
    2.2.2. else
      2.2.2.1. return {true, in_less_than_k & not_cooked}
3. return has_solution_in_k (OriginalProblem, plies)
```

Figure 8: The main algorithm for solving a k -mover

K-Move_Solver uses an iterative deepening scheme to determine whether a problem has a solution in less than k moves. In line 1, the number of moves, k , is translated to the number

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

of plies $k*2-1$. In line 2, by calling to the function *has_solution_in_k* with parameters 1, 3, 5, ... k , it is discovered if there is a solution and if it is, is it cooked or not. The information is stored in 2 variables, 'has_solution' and 'type'. In line 2.2.1, there is a check for the uniqueness of the solution. At this point, the algorithm returns whether the problem has a solution in less than k moves and whether it is cooked.

If the cycle started at line 2 has survived all iterations, there is a possibility for a solution in k moves. In line 3, the algorithm checks this fact by another call to *has_solution_in_k* with the maximum available number of plies as a parameter. At his point, the return values depend completely on the return values by *has_solution_in_k*.

The function *has_solution_in_k* is shown in Figure 9. It uses a stack for implementing Bounded DFS scheme and an algorithm on AND/OR trees (see 2.3.3). Each node on the graph can be solved, unsolved, or unknown. Solved nodes are those nodes for which we know that their subtree contains a solution. Unsolved nodes are those nodes for which we know that their subtrees does not contain a solution. Unknown nodes are those nodes for which we cannot say if they are solved or unsolved yet.

At the beginning, all nodes are unknown. At the end of the algorithm, there are three possible situations: (1) the root node is marked as solved only once – there is a solution, (2) the root node is solved once then unsolved – there is a solution, and (3) the root node was marked twice as solved – there are two solutions, i.e. we have a dual (it is enough twice to detect a dual).

The status of each expanded node is updated by *UpdateNodesStatus*. A node is declared as solved, if: (1) it is an AND node and Black is checkmated, (2) it is an AND node and all its sons (which are OR nodes) are selected as solved, and (3) it is an OR node and one of its sons (which is AND node) is solved.

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

```
{has_solution, is_cooked } has_solution_in_k (OriginalPosition, plies)
// finds if a problem has or has not a solution and if it is cooked
0. sol_found ← false
1. PushIntoStack ({OriginalProblem, 0})
    // 2nd parameter is for the node's level in the tree
2. While NotEmptyStack ( )
    2.1. {CurrentPosition, Level} ← PopFromStack()
    2.2. SolvedUnsolved (CurrentPosition, OriginalPosition)
        // update the nodes' un/solved status
    2.3. If Unsolved (OriginalPosition) and sol_found = false return {false, false}
        // no solution
    2.4. If Unsolved (OriginalPosition) and sol_found = true return {true, false}
        // only one solution
    2.5. If Solved (OriginalPosition) and sol_found = true return {true, true}
        // cooked
    2.6. If Solved (OriginalPosition) and sol_found = false then
        2.6.1. Mark OriginalPosition as Unknown // canceling the solution
        2.6.2. sol_num ← true
    2.7. If Level <= plies then
        2.7.1. For each Successor of CurrentPosition // from right to left
            2.7.1.1. CurrentPosition = GenerateSons (CurrentPosition)
            2.7.1.2. PushIntoStack ({CurrentPosition, Level + 1})
```

Figure 9: An algorithm for checking if a problem has a unique solution in k moves

A node is declared as unsolved, if: (1) it is a node on the last level (which is AND node) and Black can move, (2) it is a stalemate situation, (3) White is checkmated (it can be only at OR node), (4) it is an OR node and all its sons are selected as unsolved, and (5) it is an AND node and one of its sons is selected as unsolved..

Each expanded node is checked whether it is solved or unsolved and the appropriate changes are made in the stack in the function *SolvedUnsolved* according to the description

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

above. On each application of the function the root node can become solved or unsolved. In this case, we know whether there is a solution.

In lines 2.4-2.7 there is a check if we solved the whole problem or, contrariwise, if the problem has no solution. The variable 'sol_found' is a flag, showing if during the search a solution was found. If it is found that the root node is unsolved and 'sol_found' has not been changed, the problem has no solution (line 2.4). If the variable was changed, it means that the problem has just one solution, i.e. it is not cooked (line 2.5).

If the root node is solved and it is the second found solution ('sol_found' = true), then the problem is cooked (line 2.6). Finally, 'sol_found' is set to be true if we find a first solution (line 2.7).

Figure 10 presents the recursive algorithm *SolvedUnsolved* which implements the ideas described above. Line number 1 takes care for the nodes that were checked as solved. The function receives 'CurrentPosition' node as a parameter. If it is solved, we may conclude that it is always an AND node because, for the first time, *has_solution_in_k* passes such a node as AND node, during the recursive calls the passed node is a grandfather of 'CurrentPosition'. *has_solution_in_k* declares a node as solved when Black got a checkmate at this node.

If we have an AND node solved, we can immediately say that its father is solved too, because the father is an OR node. Therefore, all still uninvestigated brothers of the AND node can be pruned. It is done at line 1.1. The OR node is marked as solved at lines 1.2-1.3. At this point, it is possible that we solved the whole tree. Therefore, the appropriate test is done at line 1.4.

The father of the OR node (white-to-move) is always an AND node (black-to-move). Therefore, all the OR node's brothers must be solved before we can mark the AND node as solved. The algorithm checks whether this condition is satisfied at line 1.4.1. At this point, we returned to the initial situation: the recursive call can be done.

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

SolvedUnsolved (CurrentPosition, OriginalPosition)

// updates unknown nodes to be solved or unsolved

1. If Solved (CurrentPosition) then //CurrentPosition is always an AND node at this point
 - 1.1. Remove from stack all CurrentPosition's brothers // pruning
 - 1.2. CurrentPosition \leftarrow CurrentPosition->father
 - 1.3. mark CurrentPosition as solved
 - 1.4. if UnSolved (OriginalPosition) then
 - 1.4.1. if CurrentPosition is the last solved brother
 - 1.4.1.1. mark CurrentPosition->father as solved
 - 1.4.1.2. SolvedUnsolved (CurrentPosition->father, OriginalPosition)
2. elseif UnSolved (CurrentPosition) then
 - 2.1. if UnSolved (OriginalPositio) then
 - 2.1.1. if CurrentPosition is an OR node then
 - 2.1.1.1. Remove from stack all CurrentPosition's brothers // pruning
 - 2.1.1.2. CurrentPosition \leftarrow CurrentPosition->Father
 - 2.1.1.3. mark CurrentPosition as unsolved
 - 2.1.2. if CurrentPosition is the last unsolved brother
 - 2.1.2.1. mark CurrentPosition->father as unsolved
 - 2.1.2.2. SolvedUnsolved (CurrentPosition->father, OriginalPosition)

Figure 10: Solved-unsolved recursive pruning procedure

The situation with unsolved nodes is slightly different because *SolvedUnsolved* can receive the parameter 'CurrentPosition' as an OR node or an AND node. For this reason, there is an additional check at line 2.1.1. If it is an AND node, i.e. the situation is shifted, we just shift it back and we can continue with the cycles of the recursion.

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

Generally, while checking the effect of an unsolved node, we note that the events that take place are exactly the opposite of the solved node. For instance, if it is an unsolved OR node we can prune all its brothers. Otherwise, if it is an unsolved AND node we should wait while all its brothers become unsolved to declare its father as unsolved too.

The algorithm KMoveMetric that applies the metric in Figure 6 is represented in Figure 11. Starting this algorithm we already have the keymove and know that the problem is not cooked. If it is cooked, the $-\infty$ is returned at line 2.1. In the second case, the function calls the recursive function RecKMoveMetric.

First of all, RecKMoveMetric checks the stopping condition which is the number of plies to develop ('plies' is always odd). In the body of the function we calculate all parameters of the cluster. Particularly, we check for each response of White if it is a dual (line 4.1). If it is not, each position (except mate positions) reproduces another cluster. This is done by a recursive call (line 4.3.3).

Let us denote the branching factor as b . Analyzing the complexity of the function shows that for the original position, in the worst case, there are $O(b^{2k-1})$ nodes to find if the problem is cooked or not by *has_solution_in_k*. (The worse case can happen when all solution moves of White happen after we have observed all brother nodes in all clusters.) This is done at line 1 of KMoveMetric. Then, for each son of a Black node, we check again if it is dual in line 4.1 of RecKMoveMetric. This is the same check for being cooked as for original position but 2 plies deeper. There are b nodes which are the replies of Black to the keymove. For each such a node $O(b^{2k-3})$ nodes must be developed by *has_solution_in_k* to determine if there is a dual. Therefore, we have $b * O(b^{2k-3}) = O(b^{2k-2})$ in total.

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

1) KMoveMetric (OriginalPosition, k)

// calculates the metric k -move_metric for a problem in k moves

1. PositionAfterKeymove \leftarrow GetPositionAfterKeymove (OriginalPosition)
2. If PositionAfterKeymove = NULL then // in case of a cook or short solutions
 - 2.1. return $-\infty$
3. Return (RecKMoveMetric (PositionAfterKeymove, $2*k-1$) +
 $change_in_pieces$ (OriginalPosition, CurrentPosition))

2) RecKMoveMetric (CurrentPosition, plies, Cluster's_level)// always black-to-move

// recursively implements the proposed measuring metric for k -move problems. All variables

// are the same as in the metric in Figure 6.

1. If plies < 0 return 0
2. Score \leftarrow 0
3. BlackMoves \leftarrow GetAllBlackMoves (CurrentPosition)
4. For all BlackPosition in BlackMoves
 - 4.1. {has_solution, is_dual } \leftarrow has_solution_in_k (BlackPosition, plies)
 - 4.2. If is_dual then
 - 4.2.1. duals_penalty \leftarrow relative_major_duals_coef*is_major +
 relative_minor_duals_coef*is_minor
 - 4.2.2. Score \leftarrow Score – relative_duals_coef * plies // dual's level
 - 4.3. Else
 - 4.3.1. update #unique_variants and #all_non_dual_variants
 - 4.3.2. WhiteMoves \leftarrow GetAllWhiteMoves (BlackPosition)
 - 4.3.3. Score \leftarrow Score + RecKMoveMetric (Position, plies – 2, Cluster's_level + 1)
 - 4.3.4. Score \leftarrow Score +

$$\frac{relative_cluster_coef * cluster's_level * \#unique_variants^2}{\#all_non_dual_variants}$$

5. Return Score

Figure 11: An algorithm for k -move metric calculation

3. DICP, 2-MOVE CHESS COMPOSER, AND K-MOVE CHESS COMPOSER MODELS

For the second mate level, there are b^2 Black nodes, because for each Black move after the keymove there are unique responses of White and b Black moves in response to each White move. There are $O(b^{2k-5})$ nodes in subtrees developed by *has_solution_in_k* of the second mate level in the worse case because subtrees start at level 5. Therefore, we have $b^2 * O(b^{2k-5})$ nodes in total. In the same way, there are $b^{k-1} O(b^{2k-(2k-1)=1})$ nodes to develop on the mate level k in the worse case.

To determine duals, the algorithm checks in total in the worst case as follows:

$$1 * O(b^{2k-1}) + bO(b^{2k-3}) + b^2(b^{2k-5}) + \dots + b^{k-1}O(b^{2k-(2k-1)}) =$$

$$O(b^{2k-1}) + O(b^{2k-2}) + O(b^{2k-3}) + \dots + O(b^k) = O(b^{2k-1})$$

The main advantage of this algorithm is its complete memory independence. In order to find all duals and use them in the metric calculations, we could store the whole solution tree and then to calculate. The worse case for storing the solution tree happens when on each step there are no duals for White because we can ignore subtrees of duals. Therefore, we have b moves of Black, 1 unique response of White, b responses of Black, and so on:

$$all_nodes = 1 + b + b + b^2 + b^2 + b^3 + \dots + b^{k-1} + b^{k-1} =$$

$$1 + 2b(1 + b + b^2 + \dots + b^{k-2}) = 1 + \frac{2b(b)^{k-1} - 1}{b - 1} = O(b^{k-1})$$

There are $O(b^{k-1})$ nodes to store in the worse case. In practice, it is impossible even for relatively small k -s.

The model for k -movers is only a skeleton for a future research. No themes were inserted into the metric yet.

4. Experimental Results

The experiments were carried out in two steps. The first step in our experiments was to work on the same set of 36 problems that was checked by ICP. For this purpose, the light version of *Chess Composer* was used. This version checks always to level 3. We call it DICP (Deep Improver of Chess Problems). Its algorithm is shown in Figure 1. Results are presented in section 4.1. The second step was tried using *Chess Composer* (the improved version) on the extended database of 100 problems (including the mentioned 36). Its algorithm is shown in Figure 2. Its results are shown in section 4.2. Four illustrative examples for problems improved by our models are presented in section 4.3.

4.1. DICP

DICP has been tested on the same 36 problems as ICP. Most of the problems were collected from relevant composition books and correspondences (Haymann, 1988-1991; Howard, 1943; Howard, 1962; Howard, 1970; Harley, 1931). Each problem included at least one theme from the themes that have been defined in (HaCohen-Kerner et al., 1999).

Table 3 presents the number of generated improvements by DICP for all 36 original problems. In average, for each problem 6, 429 and 29422 improvements were found on the first, second and third levels, respectively. Most of the improvements were found on the third level. Therefore, it is natural that the best improvement for each problem (if exists) will be found on the third level, as shown in Table 3.

4. EXPERIMENTAL RESULTS

| # of problems | 1-level | 2-level | 3-level | total |
|-------------------------------|---------|---------|----------|----------|
| # of improvements | 6 | 429 | 29382 | 29817 |
| Deviation (nodes) | 9 | 701 | 50306 | 51006 |
| Generated | 448 | 96066 | 12499298 | 12595812 |
| Deviation (nodes) | 38 | 15651 | 2961155 | 2976844 |
| # of improvements / generated | 1.339% | 0.447% | 0.235% | 0.237% |

Table 3: Number of all generated improvements by DICP for the 36 original problems

It is interesting to notice that the ratio of improved problems to generated problems decreases as more transformations are applied. Thus, 1.34% of the problems were selected as improved after one transformation. However, only 0.24% of the problems were selected as improved after three transformations. The general conclusion is that on lower levels we find more improvements in absolute numbers but with the fewer rates.

| | | # of improved problems | Best improvement after exactly | | |
|-------------|---|------------------------|--------------------------------|-------------------|-------------------|
| | | | 1 transformation | 2 transformations | 3 transformations |
| ICP | # | 10 | 8 | 2 | 0 |
| | % | 100.0% | 80.0% | 20.0% | 0.0% |
| DICP | # | 32 | 2 | 2 | 28 |
| | % | 100.0% | 6.2% | 6.3% | 87.5% |

Table 4: Rate of improved problems in ICP and DICP (best improvement for each problem)

4. EXPERIMENTAL RESULTS

Table 4 presents the rate of improved problems in ICP and DICP (for each problem only the best improvement if found). While ICP improves only 10 out of 36 problems (about 28%), DICP improves 32 out of 36 problems (about 89%). Most of the best improvements achieved by ICP occurred after only one transformation, and no problem has been improved by three transformations. In contrast, 87.5% of DICP's best improvements have been achieved after three transformations (see also Table 6 for statistics about the transformations). It supports the assumption that the addition of pieces improve the quality of the problems, e.g., by expressing more themes and bonuses.

Table 5 presents other interesting statistics concerning the performance of DICP. The most clear finding is an average addition of 1.66 pieces (increases from 7.31 to 8.97, 22.7%) to each problem that have been improved. These additions have contributed to the following improvements:

| | # of pieces | # of themes | # of unique thematic variants | proportional rate of thematic variants |
|--|-------------|-------------|-------------------------------|--|
| (A) average for original problems | 7.31 | 2.25 | 1.69 | 0.43 |
| Deviation for A | 0.96 | 0.69 | 1.01 | 0.31 |
| (B) average for improved problems | 8.97 | 3.09 | 2.22 | 0.52 |
| Deviation for B | 1.41 | 0.87 | 1.13 | 0.34 |
| Improvement rate | 22.7% | 37.4% | 31.5% | 21.6% |

Table 5: Statistics concerning the performance of DICP for all 32 improved problems

4. EXPERIMENTAL RESULTS

- (1) the average number of themes increases from 2.25 to 3.09 (37.4%)
- (2) the average number of unique thematic variants increases from 1.69 to 2.22 (31.5%)
- (3) the average proportional rate of the thematic variants increases from 0.43 to 0.52 (21.6%)

Table 6 presents the distribution of the improving transformations occurred in the best improvement which was found for each one of the 32 problems successfully improved by DICP. One clear finding is that the addition transformation is the most contributing transformation. The next contributing transformation is deletion and, eventually, transparency. 25 out of the 32 best improvements occurred with applying either one deletion with two additions (16 out of 32) or three additions (9 out of 32).

| | add | del | add- add | del-add- add | add-add- add | trans-add- add | trans-del-add |
|----------------------|------------|------------|---------------------|-------------------------|-------------------------|---------------------------|----------------------|
| # of problems | 1 | 1 | 2 | 16 | 9 | 2 | 1 |

Table 6: Distribution of improvement transformations

Figure 12 shows that most improved problems contain more pieces than the original problems. 30 problems out of 32 got additional pieces at their best improvements. This finding supports the hypothesis that adding more material improves the quality of the problems.

Figure 13 shows that most new problems contain more themes. 66% of the problems got one or more new themes. 33% of the improvements are without adding themes. This is because our evaluation function (Appendixes B and C) gives bonuses for "good" pieces' placement, e.g. a keymoved piece moves longer than in an original problem. This finding

4. EXPERIMENTAL RESULTS

supports the hypothesis that adding more material to a certain extent increases the number of the themes included in the problems.

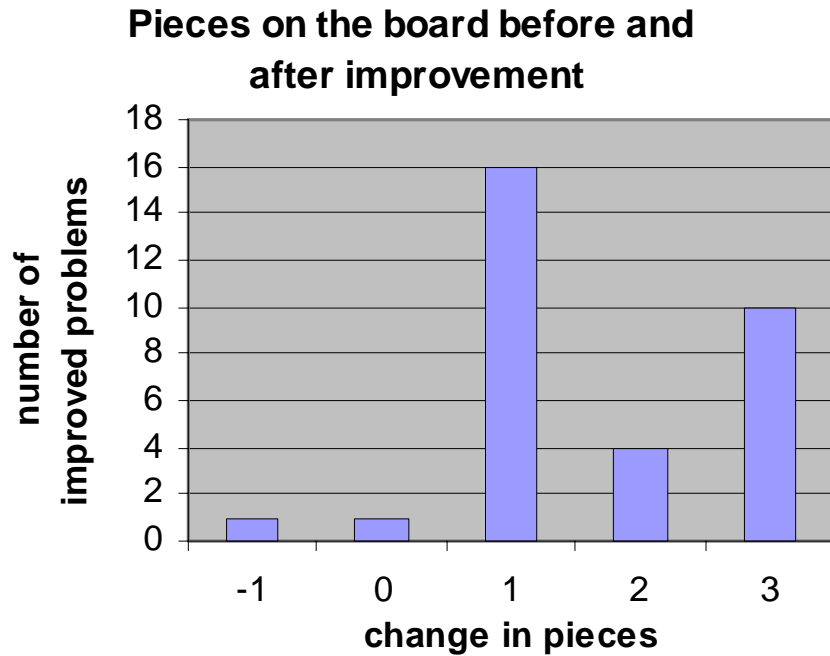


Figure 12: Improved problems as a function of the change in number of pieces

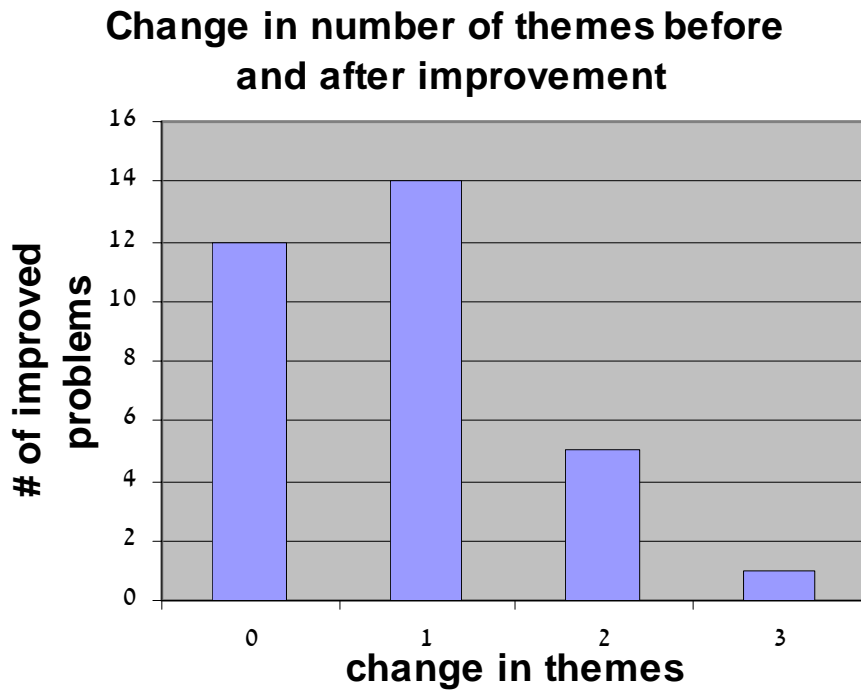


Figure 13: Improved problems as a function of the change in number of themes

4. EXPERIMENTAL RESULTS

Upon change in themes, intuitively, it is harder to add three themes to a problem than one. Figure 13 supports this intuition: 14 problems got 1 additional theme, 5 problems got 2, and just 1 problem got 3 additional themes.

4.2. Chess Composer

Chess Composer has been tested on 100 problems. Most of the problems were collected from relevant composition books and correspondences (Haymann, 1988-1991; Howard, 1943; Howard, 1962; Howard, 1970; Harley, 1931; Thulin 2000, 2003, 2004; Török, 2001, 2002; Godbout and Alain, 2001). The database includes the 36 problems used in ICP and in DICP.

Figure 14 shows that the new transformed positions are usually not legal by chess rules (about 30%) or not legal by chess composition rules ("not 2-movers" – about 63%). In about 7% of cases, the problems are legal, but they are of worse quality than the original problem. Finally, only 0.32% of the problems are considered as real improvements.

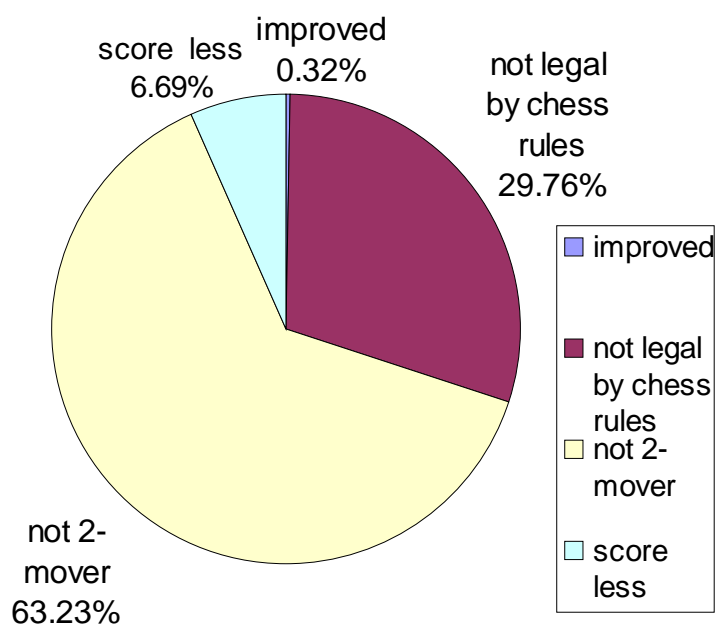


Figure 14: Distribution of transformed new positions for all levels

4. EXPERIMENTAL RESULTS

Figure 15 presents the total change in the scores (quality values) from the original problems to the best scored problems. While 37 original problems have a score less than 50, only 11 improved problems have such a score. While only 57 original problems have a score higher than 50, there are 87 improved problems that have such a score.

The average score of the original problem was 49.62 ± 50.68 . The average score of the best-scored improved problem is 94.93 ± 27.52 . This interesting fact tells us that our evaluation function stabilizes problems in terms of scores. That is, taking a random problem with different range of scores, we bring them by the series of transformations to an "ideal" problem within the new range.

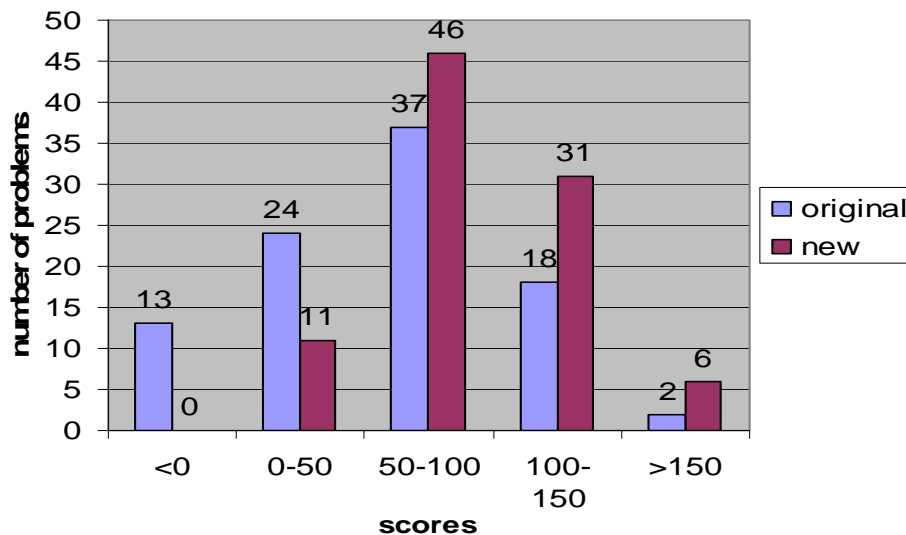


Figure 15: Change in scores

Tables 1 and 7 show that the more nodes are developed, the more improvements were found. Our goal is to find the best improvements (if exist). The chance to find a better improvement grows with the number of generated nodes. However, the growth is not linear (see percentages in Table 7) - the deeper we look, the fewer improvements we find. The last result is similar to DICP's results (see Table 3).

4. EXPERIMENTAL RESULTS

| # of problems | # of improvements | | | Total |
|--------------------|------------------------------|---------|---------|-------|
| | 1-level | 2-level | 3-level | |
| Average # of nodes | 5 | 371 | 24356 | 24732 |
| Deviation (nodes) | 7 | 554 | 39803 | 40364 |
| | improvements/nodes per level | | | |
| improved/generated | 1.52% | 0.56% | 0.32% | |

Table 7: Number of all improvements generated by *Chess Composer* for all 100 original problems

It must be pointed that the quality of a new problem does not depend linearly on a number of applied transformations. A problem and its improvement presented in positions in Diagrams 1 and 2 (section 5.1) are the disproof. We got the position in Diagram 2 from position in Diagram 1 by three transformations. If we use only one or two transformations from this set of three, the positions are even not legal!

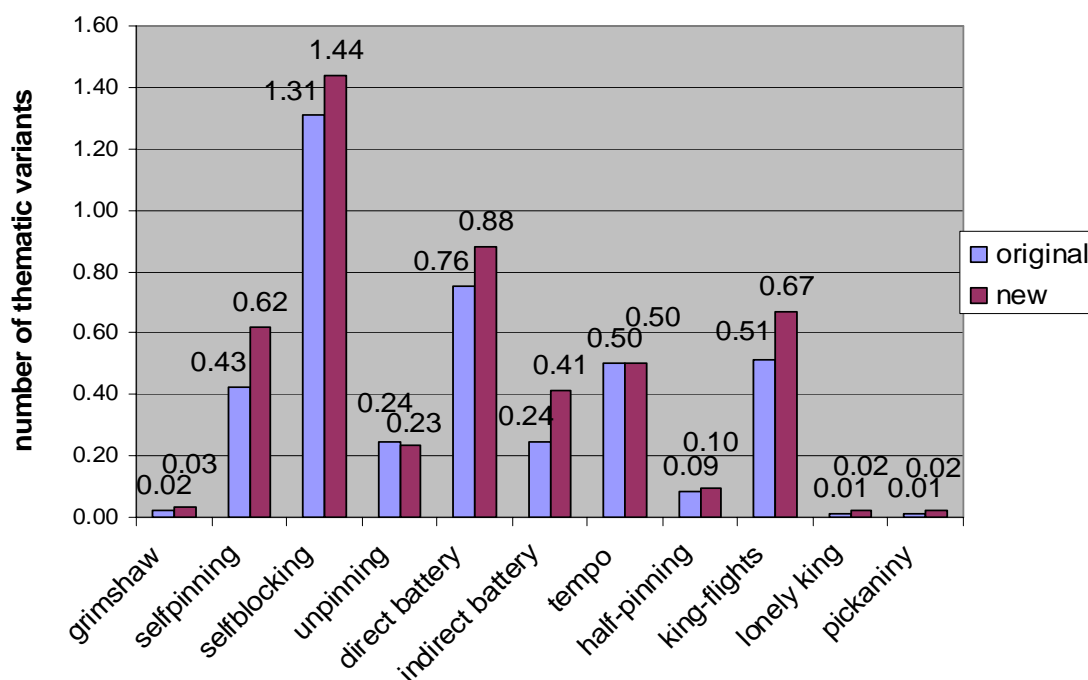


Figure 16: Average growing in themes

4. EXPERIMENTAL RESULTS

On average, each problem achieves additional thematic variants (Figure 16) or additional themes (Figure 17). Almost half of the problems (48) got significant improvement because additional themes were added (Figure 17). However, two problems got reduction in one theme. The explanation is that these problems contain thematic duals and triples which are regarded as penalties. In these cases, *Chess Composes* found new problems with no duals at the cost of decreasing in themes.

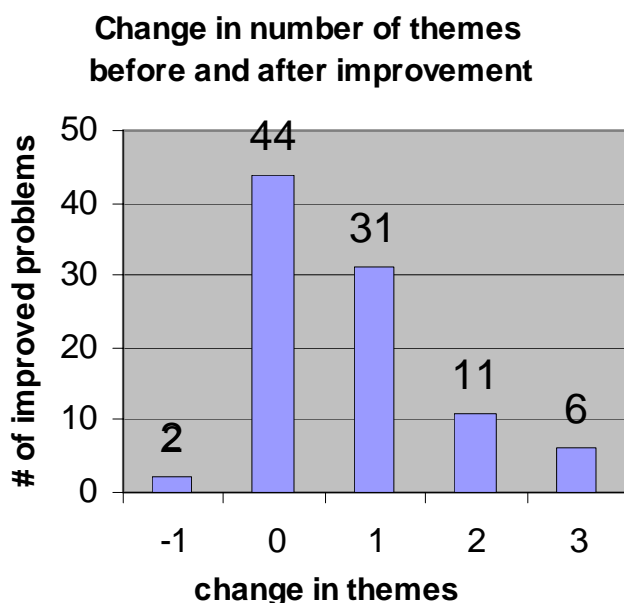


Figure 17: Almost half of the problems achieve additional themes

Figure 18 shows that there is an increase in the number of pieces in 87 improved problems. Generally, due to the minimalism principle, the "dressing of the board" (an old technique of adding pieces for complication of a problem) reduces the quality of the problem in modern chess composition (Harley, 1931). However, despite this many meaningful improvements have been found.

As shown in Table 3, a sequence of three transformations is the most frequent case. The same is right in the case of the best scored improvements (78 cases in Figure 19). However, in 41 problems just one piece was added (Figure 18). The explanation is as follows: Figure

4. EXPERIMENTAL RESULTS

16 shows that 45 improvements contain at least the following two transformations: "deletion of a piece" and "addition of a piece". This sequence can represent either a "moving of a piece" or a "change of a piece to another piece", or a combination of the change and the moving, transformations applied in HaCohen-Kerner et al. (1999). Another finding, shown in Figure 19, is that an addition of a piece is the most contributing transformation.

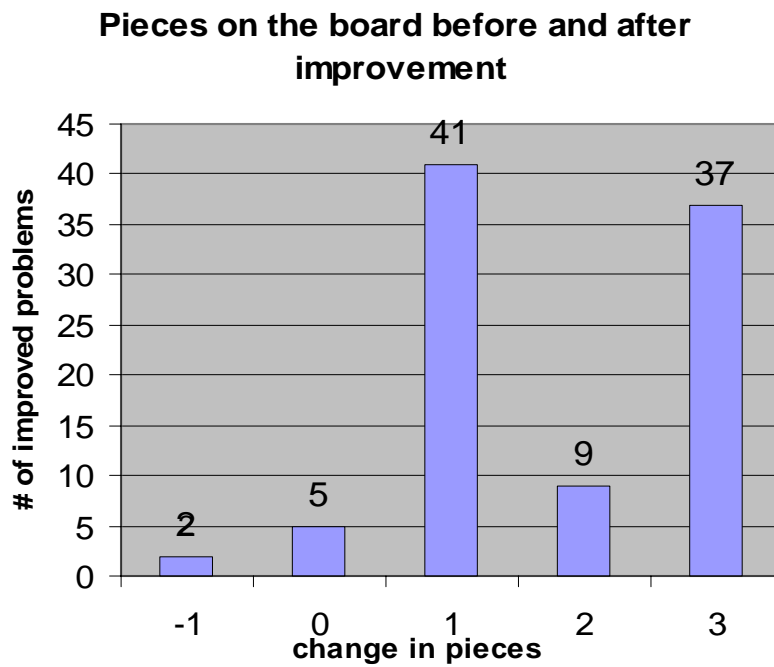


Figure 18: Number of improved problems as function of the change in pieces at the best-scored problems

4. EXPERIMENTAL RESULTS

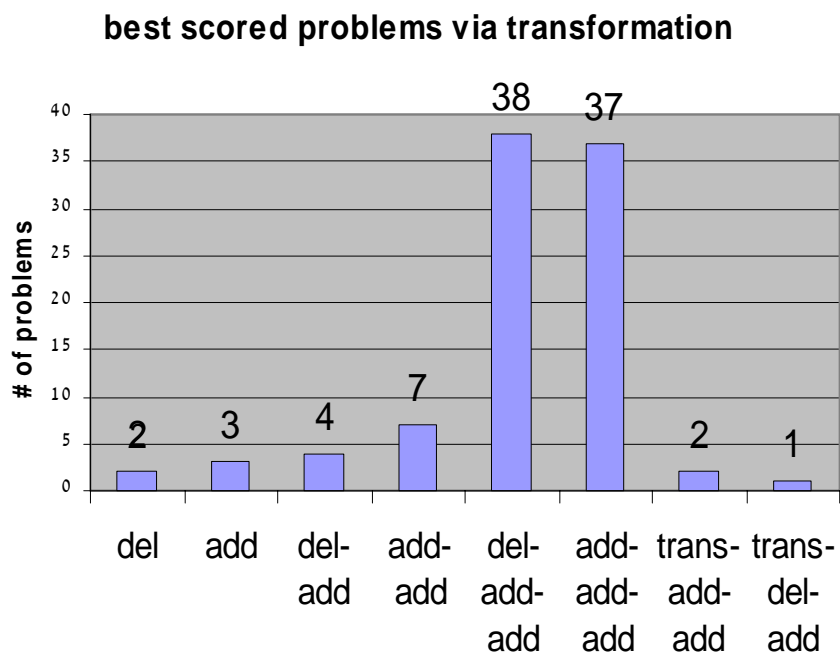


Figure 19: Distribution of transformations

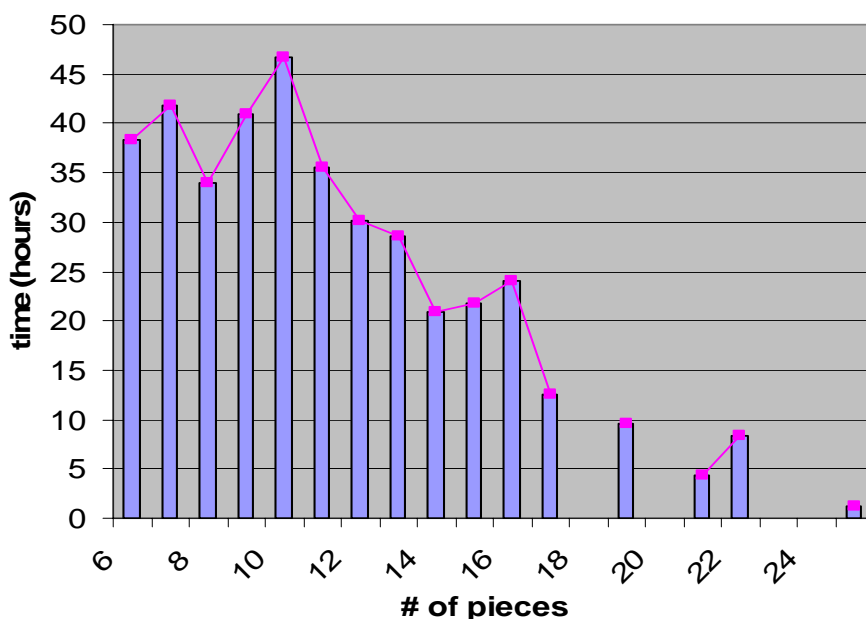


Figure 20: Run-time as a function of the number of pieces

In Figure 20, we can see, as expected, that the time is decreasing as a function of the number of pieces at the initial position. This is because the more pieces are on the board, the less free squares left. Therefore, because on each free square about 10 types of pieces can be

4. EXPERIMENTAL RESULTS

added, the "addition" transformation gives less possibilities. The decreasing line is not straight because of the other component - the already mentioned rule of chess composition: there can be no more than 1 queen, 2 rooks, 2 bishops, 2 knights, 8 pawns, and 1 king for each color. Thus, as well as there are more pieces on the board, time needed to develop all positions becomes more problem depended.

There are three problems that *Chess Composer* failed to improve even after four transformations. Three other problems have been improved slightly only after four transformations. These 4-transformation improvements did not add themes. All these six problems have several common features:

- (1) they include more than one theme for each problem,
- (2) there are neither duals, nor triples nor multiples,
- (3) every (or almost every) variant is thematic.

In other words, these six original problems are either perfect or almost perfect.

4.3. Illustrative Examples

Below, four examples are presented. These four problems are taken from the database of 100 problems. Part of the problems has not been improved by ICP but only by the new models (DICP and the *Chess Composer*). One of the problems has been improved by ICP, but much better improvement was found by *Chess Composer*.

4.3.1. Example 1

The miniature (a problem that contains at most 7 pieces) presented in Diagram 1 has not been improved by ICP. However, it has been improved significantly by *Chess Composer* to the problem in Diagram 2 (see example 1 in Appendix E for the detailed analysis). The composition themes expressed in position 1 are: (1) "Grimshaw", which means that two

4. EXPERIMENTAL RESULTS

Black pieces block each other's line by interfering on a same square and by that cause different mate variations and (2) "Self-blocking", which means that a Black piece blocks one of Black King's flights and by that creates different mate variations (Appendix C).

The solution to the problem in Diagram 1 is the keymove Queen b5 – c4. There are 9 possible variants; only 3 of them are thematic (full details in example 1 in Appendix E).

The best improvement that has been found by *Chess Composer* contains the 3 following transformations: "addition of a White pawn on e3", "addition of a Black pawn on e4" and "addition of a Black rook on a7". As a result, we reach the position in Diagram 2.

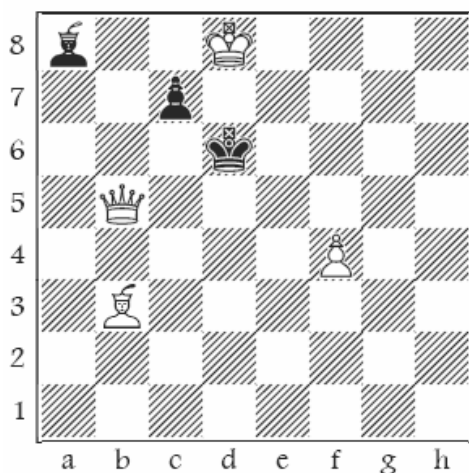


Diagram 1: H. Weenink, *Good Companion*, 12/1917. Original Problem

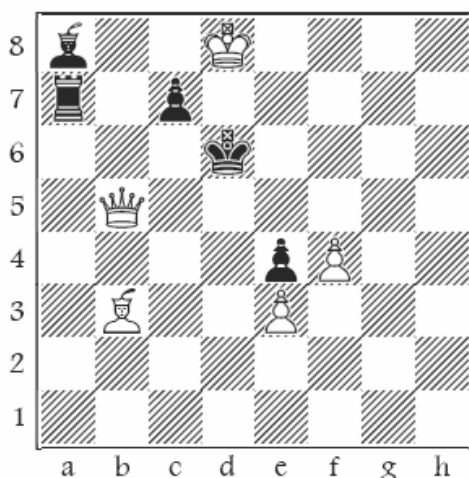


Diagram 2: The best improvement by *Chess Composer*

4. EXPERIMENTAL RESULTS

The solution to problem in Diagram 2 is the same - keymove Queen b5 – c4. Then, there are 12 possible variants; 5 of them are thematic (all these variants are presented at example 1 in Appendix E).

The problem in Diagram 2 is considered better than the problem in Diagram 1 for the following three main reasons:

1. The problem in Diagram 2 includes two pairs of the complex composition theme “Grimshaw” (i.e., 4 variants) instead of one pair (i.e., 2 variants) in the problem in Diagram 1. The two Grimshaw squares are c6 (the mutually interfering pieces are pawn and bishop) and b7 (rook and bishop). Problems that contain two pairs of the Grimshaw theme are relatively rare and hard to accomplish. Therefore, this addition is considered as a meaningful improvement of the original problem.
2. The problem in Diagram 2 contains 5 thematic variants out of 12 (42%), while the problem in Diagram 1 contains only 3 thematic variants out of 9 (33%).
3. Moreover, after making the keymove for the problem in Diagram 1, White has a *mate-threat 2*. Queen c4:c7 (for notation see Appendix A). In the problem in Diagram 2, White has no mate-threat, yet succeeds in mating Black in two moves. This feature is called “*tempo*”, and it is also considered as an additional composition theme. Such problems are harder for human solution (Harley 1931).

The added pawns in Diagram 2 are necessary for the solution. The Black pawn on e4 prevents Black to make moves by its bishop to e4, f3, g2, and h1, moves that are not needed. The White pawn on e3 just keeps the Black pawn on e4 not moving.

There is only one slight disadvantage in the problem in Diagram 2 comparing to the problem in Diagram 1. The original problem includes only 7 pieces (i.e., it is a miniature),

4. EXPERIMENTAL RESULTS

while the problem in Diagram 2 contains 10 pieces. The full evaluation of the improvement process is presented in example 1 in Appendix E.

4.3.2. Example 2

The second example is the original miniature presented in Diagram 3. The composition theme expressed in this problem is “self-blocking”, which means that Black closes one of his king-flights by at least one of his moves, enabling White to mate him.

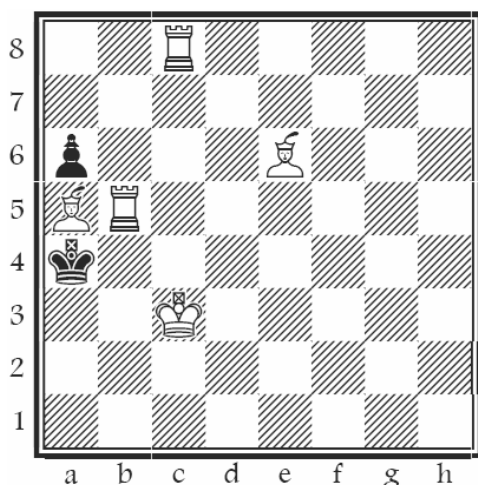


Diagram 3: Unknown source 1. Original problem

The solution to the problem in Diagram 3 is as follows (example 2 in Appendix E). The keymove is Bishop a5 - d8. Then there are four variants, which can be derived from the four possible answers of the Black. Variant d) expresses the self-blocking theme:

| | Black's move | White's mate-move | Themes included in the variant |
|----|---------------------|--------------------------|---------------------------------------|
| a) | 1... King a4 - b5. | 2. Bishop e6 - d7. | |
| b) | 1... King a4 - a3. | 2. Rook b5 - a5. | |
| c) | 1... Pawn a6 - a5. | 2. Rook b5 - a5. | |

4. EXPERIMENTAL RESULTS

d) 1... Pawn a6 : b5. 2. Rook c8 - a8. self-blocking

ICP, while trying to improve the position in Diagram 3, applies the following transformation “taking off the White bishop on a5”. As a result, we reach the position in Diagram 4. The solution is as follows. The keymove is 1.Bishop e6 - d7. There are three possible variants then:

| Black's move | White's mate-move | Themes included in the variant |
|--------------|-------------------|--------------------------------|
|--------------|-------------------|--------------------------------|

| | | |
|-----------------------|------------------|--|
| a) 1... King a4 - a3. | 2. Rook b5 - a5. | |
|-----------------------|------------------|--|

| | | |
|-----------------------|------------------|-------------------------------------|
| b) 1... Pawn a6 - a5. | 2. Rook b5 - b3. | self-blocking, direct-battery fired |
|-----------------------|------------------|-------------------------------------|

| | | |
|-----------------------|------------------|--------------|
| c) 1... Pawn a6 : b5. | 2. Rook c8 - a8. | self-pinning |
|-----------------------|------------------|--------------|

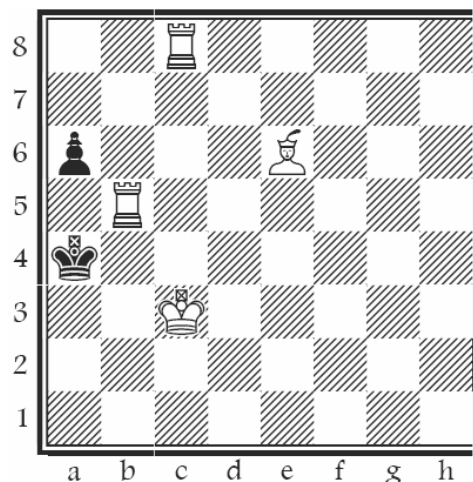


Diagram 4: The best improvement by ICP for unknown source 1.

The deletion of the White bishop on a5 leads to omission of the first variant for the problem Diagram 3. Therefore, the position in Diagram 4 has two main disadvantages in comparison to the one in Diagram 3:

1. Loss of the nice mate achieved in the mentioned variant.
2. Loss of the Black's king-flight 1... King a4 - b5 in the same variant.

However, the problem in Diagram 4 is considered to be better than the problem in Diagram 3 for four main reasons:

4. EXPERIMENTAL RESULTS

1. The same composition theme (self-blocking) is expressed by a smaller problem (with one less bishop to White, the strongest side!).
2. In contrast to the solution variants in the position in Diagram 3, all solution variants in the problem in Diagram 4 contain different mate-moves. That is, all variants are rather different and each of them contributes a novelty by its mate-move.
3. In addition, we achieved two new composition themes in the problem in Diagram 4: (1) "self-pinning" (see Appendix C) in the third variant when the Black makes a move 1... Pawn a6 : b5 and pins his king; and (2) "direct battery" (see Appendix C) in the second variant when the Black makes a move 1... Pawn a6 - a5. and White mates with Bishop on d7 because of the move 2. Rook b5 - b3.
4. Moreover, after making the keymove in the position in Diagram 3, White has a mate-threat 2. Rook b5 - a5. White has no mate-threat in the position in Diagram 4. White still succeeds in mating Black in two moves. This feature is called "tempo", which is also considered as an additional composition theme.

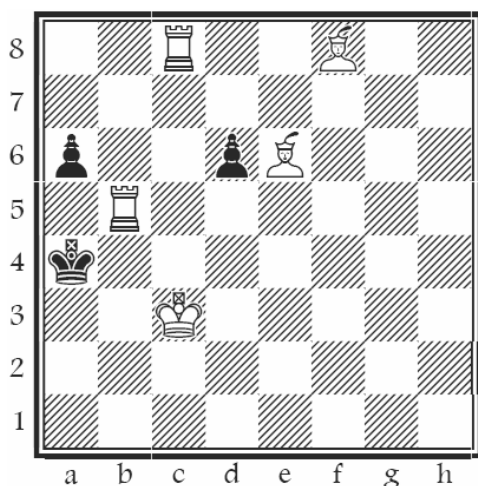


Diagram 5: The best improvement by Chess Composer for unknown source 1

4. EXPERIMENTAL RESULTS

DICP and *Chess Composer* found a better improvement for the problem in Diagram 3 (see Appendix E example 2 for full details). It applies the 3 following transformations: “taking off the White bishop on a5”, “addition of a White bishop on f8”, and “addition of a Black pawn on d6”. As a result, we reach the position in Diagram 5. The new solution is the same as to the problem in Diagram 4: the keymove is Bishop e6 - d7. Then, there are four possible variants then:

| Black's move | White's mate-move | Themes included in the variant |
|-----------------------|--------------------------|---------------------------------------|
| a) 1... Pawn d6 - d5. | 2. Rook b5 - d5. | direct battery fired |
| b) 1... Pawn a6 - a5. | 2. Rook b5 - b3. | self-blocking, direct battery fired |
| c) 1... Pawn a6 : b5. | 2. Rook c8 - a8. | self-pinning |
| d) 1... King a4 - a3. | 2. Rook b5 - a5. | self-pinning |

The new problem is considered as much better than the problems in Diagrams 3 and 4. Above, there was a comparison between the problems in Diagrams 3 and 4. Now we shall compare between the problems in Diagrams 4 and 5. The problem in Diagram 5 has many advantages in comparison to the problem in Diagram 4, as follows:

1. The problem in Diagram 5 has 4 different variants while the problem in Diagram 4 has just 3 variants.
2. All variants in the problem in Diagram 5 are thematic (expressing themes) while the problem in Diagram 4 has one non-thematic variant.
3. There are 2 variants that include the theme “self-pinning” and 2 variants that include the theme “direct battery fired” in the problem in Diagram 5. The problem in Diagram 4 has only one variant for each theme.
4. The position in Diagram 4 has been achieved after only one transformation. The position in Diagram 5 has been achieved after 3 transformations (they can be regard as 2 because

4. EXPERIMENTAL RESULTS

the deletion of the White bishop and its addition on another square can be regarded as one moving transformation).

There is only one disadvantage in the problem in Diagram 5 comparing with the problem in Diagram 4. The problem in Diagram 5 includes one White bishop more (which returns us to the original position in Diagram 3) and one additional Black pawn than the problem in Diagram 4. Therefore, the problem in Diagram 5 is not considered as a miniature (a problem which contains at most 7 pieces) any more.

4.3.3. Example 3

The next example is the miniature presented in Diagram 6. The composition themes expressed in this problem are: “Tempo/Waiting move”, “self-blocking” and “king flights” (Appendix C). The solution is the keymove 1.N e4 (example 3 in Appendix E). Then, there are 4 possible variants, only 3 of them express “self-blocking”, as follows.

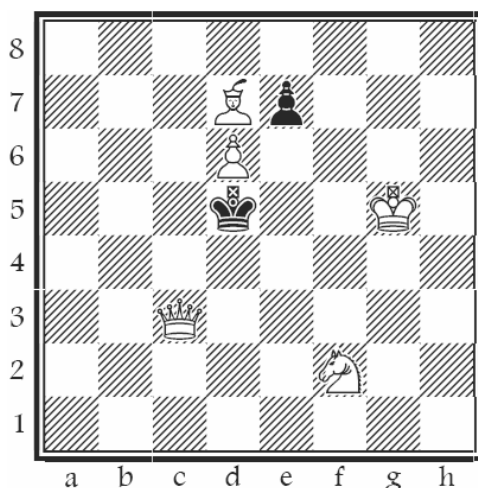


Diagram 6: Geoffrey Mott-Smith. The Chess Review, December, 1937.

Appeared also in Howard (1962) - #122. Original problem

| Black's move | White's mate-move | Themes included in the variant |
|---------------------|--------------------------|---------------------------------------|
| a) 1... Pawn e7-e6. | 2. Bishop d7 – c6. | self-blocking |
| b) 1... Pawn e7-e5. | 2. Queen c3 – d3. | self-blocking |

4. EXPERIMENTAL RESULTS

c) 1... Pawn e7-d6. 2. nKnight e4 – f6. self-blocking

d) 1... King d5- e4. 2. Bishop d7 – c6.

The other themes (tempo and king-flights) have no thematic variants due to their definition (see Appendix C).

The problem presented in Diagram 6 has not been improved by ICP. However, it has been improved significantly by DICP and Chess Composer. The result is shown in Diagram 7 (for full details see example 3 in Appendix E). The solution to the problem in Diagram 7 is the same. The keymove is 1. kNight f2 – e4. Then, there are 5 possible variants, 4 of them express an additional theme “pickaniny” which is considered as a rather complex theme. The “pickaniny” theme means that there are 4 different variants based on different moves of the same Black pawn and for each one of them there is another mate-move by White. The 5 possible variants are:

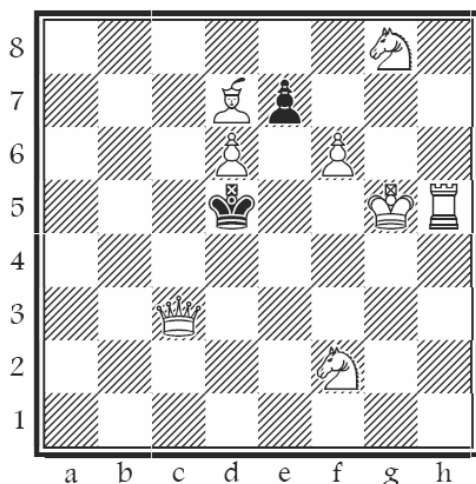


Diagram 7: The best improvement by Chess Composer

| Black's move | White's mate-move | Themes included in the variant |
|---------------------|--------------------------|---------------------------------------|
| a) 1... Pawn e7-e6. | 2. Bishop d7 – c6. | pickaniny, self-blocking |
| b) 1... Pawn e7-e5. | 2. Queen c3 – d3. | pickaniny, self-blocking |

4. *EXPERIMENTAL RESULTS*

- c) 1... Pawn e7-d6. 2. King g5 – f4. pickaniny, self-blocking
- d) 1... Pawn e7-f6. 2. nKight g8 – f6. pickaniny
- e) 1... King d5- e4. 2. Bishop d7 – c6.

The problem in Diagram 7 is considered better than the problem in Diagram 6 for the following main reasons:

1. The problem in Diagram 7 includes two new composition themes: “pickaniny” and “direct battery” (see Appendix C).
2. In addition, the problem in Diagram 7 contains one variant where the mate move is done by White king, which is considered as a bonus.
3. The problem in Diagram 7 contains one additional thematic variant. That is, this problem contains 4 variants out of 5 that are thematic (80%), while the problem in Diagram 6 contains only 3 variants out of 4 that are thematic (75%).

There are two slight disadvantages in the problem in Diagram 7 comparing to the problem in Diagram 6:

1. The problem in Diagram 6 includes only 7 pieces (i.e. it is regarded as a miniature), while the problem in Diagram 7 is not a miniature.
2. The problem in Diagram 6 includes 3 variants that express the theme “self-blocking” while the problem in Diagram 7 includes only 2 variants that express this theme.

4.3.4. Example 4

The last example is the miniature presented in Diagram 8. The composition theme expressed in this problem is “self-blocking”. The solution to is the keymove 1.King c7 - d8. There are 13 possible variants, only 4 of them express “self-blocking”. All 9 other variants

4. EXPERIMENTAL RESULTS

include the same mate-move and do not contribute any novelty (full details are given in example 4 in Appendix E).

ICP, trying to improve the problem, applies the following rank-transparency transformation "moving all pieces 2 ranks below". As a result, we reach the problem in Diagram 9. The solution is the keymove 1. King c5 - d6.

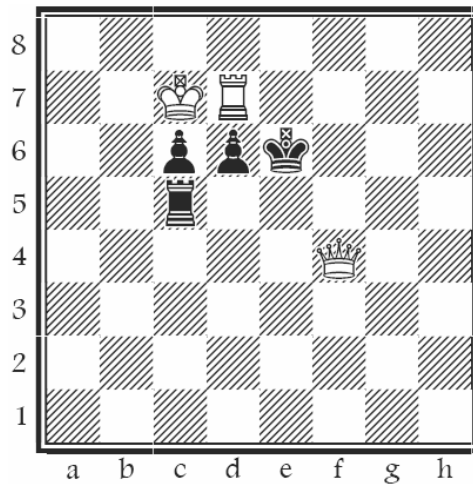


Diagram 8: Werner Speckmann, Neueste Kieler Nachrichten, 1939. Original problem

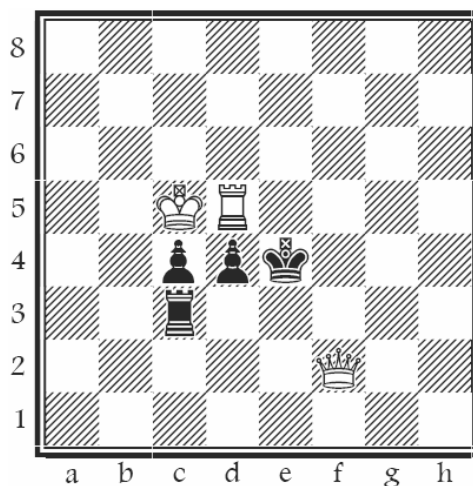


Diagram 9: The best improvement by ICP

4. EXPERIMENTAL RESULTS

The ICP's problem is considered better than the original problem for the two reasons:

- a) Black king in the problem in Diagram 9 is placed on a central square which is considered harder for White to mate.
- b) The new problem includes 11 possible variants (versus 13 in the original), 4 of them (as in the original problem) express "self-blocking". Only 7 other variants (versus 9 in the original problem) do not contribute any novelty. The four thematic variants are:

| Black's move | White's mate-move | Themes included in the variant |
|----------------------|--------------------------|---------------------------------------|
| a) 1... Pawn d4-d3. | 2. Rook d5 – d4. | |
| | 2. Rook d5 – e5. | self-blocking; (thematic minor dual) |
| b) 1... Rook c3- f3. | 2. Queen f2 – d4. | self-blocking |
| c) 1... Rook c3- e3. | 2. Queen f2 – f5. | self-blocking |
| d) 1... Rook c3- d3. | 2. Rook d5 – e5. | self-blocking |

DICP and Chess Composer apply the 3 following transformations: "rank-transparency transformation by moving all pieces one rank below", "addition of a White bishop on b2", and "addition of a Black knight on c3" (see Appendix E example 4 for details). As a result, we reach the problem in Diagram 10. The solution for this problem is the same as for the problem in Diagram 8: the keymove 1.King c6 – d7. There are nine possible variants. Only four of them are thematic (i.e., express "self-blocking"), as follows:

4. EXPERIMENTAL RESULTS

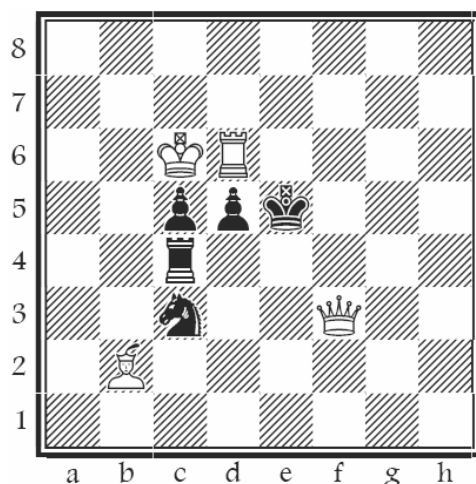


Diagram 10: The best improvement by Chess Composer

| Black's move | White's mate-move | Themes included in the variant |
|----------------------|--------------------------|---------------------------------------|
| a) 1... Pawn d5-d4. | 2. Rook d6 – e6. | self-blocking |
| b) 1... Rook c4- f4. | 2. Queen f3 – d5. | self-blocking |
| c) 1... Rook c4- e4. | 2. Queen f3 – f6. | self-blocking |
| d) 1... Rook c4- d4. | 2. Rook d6 – e6. | self-blocking |

The problem in Diagram 10 is considered much better than the original in Diagram 8 and than the ICP's improvement in Diagram 9. The comparison between the original problem and the ICP's improvement has already been presented. The comparison between the problems in Diagrams 9 and 10 is given below. The position in Diagram 10 has several advantages comparing to the problem in Diagram 9, as follows:

- a. While in the problem in Diagram 10 there are only 5 non-thematic variants, in the problem in Diagram 9 there are 7 non-thematic variants.
- b. One of the thematic variants in the problem in Diagram 9 has a serious disadvantage. It is a thematic minor dual (i.e., there are two possible mate-moves for White instead of one). In contrast, the problem in Diagram 10 contains only pure thematic variants.

4. *EXPERIMENTAL RESULTS*

The only disadvantage of the problem in Diagram 10 comparing to the others is that the others include only 7 pieces (i.e. they are regarded as miniatures), while the problem in Diagram 10 is not a miniature.

5. Summary and Future Research

DICP and *Chess Composer* are two versions of our model for 2-movers. They present an ability to improve the quality to 87% and 97% to two sets of known chess 2-movers, respectively. Most of the improvements were achieved after various sequences of three transformations (mainly three additions or a deletion with two additions), while the most contributing transformation was "an addition of a piece". The successful sequences of transformations improved the quality of the problems by increasing the average number of themes, the average number of unique thematic variants, and the average proportional rate of the thematic variants.

In the majority of improvements (87%) additional piece (pieces) was (were) added. As a result, problems got additional themes at 66% of the cases. For 6 problems there were no improvements after all possible sequences of three transformations, but for three of them there were slight improvements after a sequence of four transformations.

Some of the improvements are rather impressive, considering that most of the tested problems were composed by experienced human composers. These new improved problems can be regarded as creative from the viewpoint of experts in chess composition because these problems are better and they are not too similar to the original problems.

A general theoretical model, the k -move Chess Composer, was proposed. It uses the same algorithm as in Chess Composer for 2-movers. To detect whether or not a problem has a solution, the Depth First Search (DFS) with pruning on AND/OR trees is used. A special metric has been built in order to improve solution trees of k -movers. (A solution tree is a tree developed for all possible moves of Black and the right responses of White). This metric can

5. SUMMARY AND FUTURE RESEARCH

be used in a future evaluation function. Also, advanced programming such as Lisp and Prolog may be involved into implementation of the functions.

The main disadvantage of our applications is their relative slow speed. We apply only sequences of three and sometimes four transformations. This is because we attempt to investigate all possible positions using a brute-force search method while working with a very large branching factor. Smart pruning of estimated as non contributing sub-trees may enable searching to deeper levels in order to find better and more complex improvements.

The heuristic evaluation function is improvable too. For now, only 11 themes were applied. There are hundreds of those. Every theme can be seen as a pattern. The more such patterns are implemented in the model, the better evaluation function we get. The problem of complexity that arises can be decreased by recognizing similar patterns in the design phase. However, this problem is more domain-based.

The ideas presented at the current model may be useful not only for orthodox chess. For example, a similar evaluation function can be built for: (1) non-orthodox chess, (2) chess-like games, like Tsume-Shogi, and (3) Checkers.

A few ideas can be proposed for further research. One idea is to take an advantage of the probabilities that each kind of transformation has contributed until now. Then, we can develop only nodes with reasonable probabilities. As a result, we will search deeper in those sub-trees in which we have higher probabilities to find more and better improvements.

Another idea is to use pruning methods. We can try to detect non-useful additions of pieces (e.g., adding of non-contributing pieces far away from the black king) and not add them. As a result, we will get a lower branching factor. Again, we will be able to search deeper and to find more and better improvements.

In the area of composing k -movers, a quality function should take into consideration k -move themes and suitable bonuses and penalties. Such a function is much harder defining

5. *SUMMARY AND FUTURE RESEARCH*

than for 2-movers. Even themes with the same name as in 2-movers must be defined for the whole solution tree. The fact that there are more plies in k -movers makes it more problematic. These themes, as in the case of 2-movers, should be collected from various chess composition books, such as Harley (1944) and Howard (1943). To improve our proposed K -move model, there is a need to apply it to real k -movers, while consulting chess compositions experts.

References

Adelson-Velskiy, G.M., Arlazarov, V.L., Bitman, A.R., Zhitvotovskiy, A.A., and Uskov, A.V. (1970). Programming a computer to play chess. *Russian Mathematical Survey*, Vol. 25, pp. 221-262.

Allis, L.V., Van der Meulen, M., Van den Herik, H.J. (1994). Proof-Number Search. *Artificial Intelligence*, Vol. 66, No. 1, pp. 91-124.

Allis, L.V. (1994). Games and Artificial Intelligence. Ph.D. Thesis, University of Limburg, Maastricht, the Netherlands.

Beal, D.F. and Smith, M.C. (1997). Learning Piece Values Using Temporal Differences. *ICGA Journal*, Vol. 20, No. 3, pp. 147-151.

Breuker, D.M., Allis, L.V., Van den Herik, H.J. (1994). How to Mate: Applying Proof-Number Search to Chess. *Advances in Computer Chess 7*, pp. 23-33.

Campbell, M. S., Hoane Jr. A. J., and Hsu F.-h. (2002). Deep Blue. *Artificial Intelligence*, Vol. 134, pp. 57-83.

Frey, P.W. (1977). An Introduction to Computer Chess. *Chess skill in man and machine*, Springer-Verlag, New York, pp. 54-81.

Godbout, A. J. (2001). Echos du Nord, 256 Problemes Choisis de Johan Scheel. *Éditions de l'Apprenti Sorcier*.

HaCohen-Kerner, Y., Cohen, N., Shasha, E. (1999). An Improver of Chess Problems. *Cybernetics and Systems: an International Journal*, Taylor & Francis, Vol. 30, No. 5, pp. 441-465.

Harley, B. (1931). Mate in Two Moves. Reprinted by Bell & Sons, LTD, London, (1944).

REFERENCES

Harley, B. (1944). *Mate in Three Moves*, David McKAY CO, Philadelphia.

Haworth, G.M^cC. (2000). Strategies for Constrained Optimization. *ICCA Journal*, Vol. 23, No. 1, pp. 9-20.

Haymann, J. (1988-1991). First Steps in Composition. A Series of Correspondences. In *Variantim: Bulletin of the Israeli Chess Composition Association*.

Heinz, E.A. (1997). How Darkthought Plays Chess. *ICCA Journal*, Vol. 20, No 3, pp. 166-176.

Heinz, E.A. (1999). Endgame Databases and Efficient Index Schemes. *ICCA Journal*, Vol. 22, No. 1, pp. 22-32.

Hirose, M., Matsubara, H., Itoh, T. (1997). The Composition of Tsume-Shogi Problems. *Advances in Computer Chess 8*, eds. van den Herik, H.J. and Uitewijk, J.W.H.M., Univerciteit Maastricht, The Netherlands.

Howard, K. S. (1943). *The Enjoyment of Chess Problems*. David McKAY CO Philadelphia.

Howard, K. S. (1962). *One Hundred Years of the American Two-Move Chess Problem*. New York: Dover Publication, Inc.

Howard, K. S. (1970). *Classic Chess Problems by Pioneer Composers*. New York: Dover Publication, Inc.

Hyatt, R.M. (1999). Rotated Bitmaps, a New Twist on an Old Idea. *ICCA Journal*, Vol. 22, No. 4, pp. 213-222.

Kerner (HaCohen-Kerner), Y. (1995). Learning Strategies for Explanation Patterns: Basic Game Patterns with Application to Chess. In Veloso M.; and Aamodt, A. (Eds.), *Case-Based Reasoning: Research and Development*, Proceedings of the First International Conference,

REFERENCES

- ICCBR-95. *Lecture Notes in Artificial Intelligence*, Vol. 1010, pp. 491-500, Berlin: Springer-Verlag.
- Korf, R. E. (1985). Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, Vol. 27, No. 1, pp. 97-109.
- Kotěšovec, V. (1996). Mezi Šachovnicí a Počítačem (Between Chessboard and Computer – in Czech). Kotesovec, Praha.
- Kotěšovec, V. (1984). Řešení a Komponování Šachových Úloh Počítačem (Solving Composed Chess Problems with Aid of a Computer – (in Czech)). *Šachové umění*, Vol. 7, p.129.
- Leggett, T. (1996). *Shogi: Japan's Game of Strategy*. Charles E. Tuttle Company.
- Lindner, L. (1985). A Critique of Problem-Solving Ability. *ICCA Journal*, Vol. 8, No. 3, pp. 182-185.
- Lindner, L. (1989). Performance Improvements in Problem-Solving Programs since 1984. *Advances in Computer Chess 5*, ed. D.F.Beal, Ellis Horwood Ltd, Chichester, UK, pp. 231-264.
- Lindner, L. (1991). New Ideas in Problem Solving and Composing Programs, *Advances in Computer Chess 6*, ed. D.F.Beal, Ellis Horwood Ltd, Chichester, UK, pp. 97-116.
- Matsubara, H., Grimbergen, R. (1997). Differences between Shogi and Western Chess from a Computational Point of View. *In Proceedings: Board Games in Academia, An Interdisciplinary Approach*, Leiden, The Netherlands.
- McAllester, D. A. (1988). Conspiracy Number for Min-Max Search. *Artificial Intelligence*, Vol. 35, No. 3, pp. 287-310.
- McCarthy, J. (1997). AI as Sport. *Science*, Vol. 276, pp. 1518–1519.

REFERENCES

Nalimov, E. V., Haworth G. McC., and Heinz E.A. (2000). Space-Efficient Indexing of Chess Endgame Tables. *ICGA Journal*, Vol. 23, No. 3, pp. 148-162.

Nievergelt, J. (1977). Information Content of Chess Positions. *ACM SIGART Newsletter* 62, pp. 13-14. Reprinted as: Nievergelt, J. (1991). Information Content of Chess Positions: Implications for Game-Specific Knowledge of Chess Players. In Hayes J. E., Michie D. & Tyugu E. (Eds.), *Machine Intelligence* 12. Clarendon Press, Oxford, pp. 283-289.

Noshita, K. (1991). A Note on Algorithmic Generation of Tsume-Shogi Problems. In *proceedings of the Game Programming Workshop '96*, pp. 27-33.

Rice, J. (1996). *Chess Wizardry: The New ABC of Chess Problems*. Batsford, London.

Russell, S. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Second Edition.

Schaeffer, J. (1990). Conspiracy Numbers. *Artificial Intelligence*, Vol. 43, No. 1, pp. 67-84.

Schaeffer, J. and Plaat, A. (1997). Kasparov versus Deep Blue: the Rematch. *ICCA Journal*, Vol. 20, No. 2, pp. 95-102.

Schlosser, M. (1988). Computers and Chess Problem Composition. *ICCA Journal*, Vol. 11, No. 4, pp. 151-155.

Schlosser, M. (1991). Can a Computer Compose Chess Problems? *Advances in Computer Chess* 6, ed. D.F.Beal, pp. 117-131.

Shannon, C. E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Vol. 41, No. 4, pp. 256-275.

Seirawan, Y. (1997). The Kasparov - Deep Blue Games. *ICCA Journal*, Vol. 20, No. 2, pp. 102-125.

REFERENCES

Seo, M., Iida, H., and Uiterwijk, J.W.H.M. (2001). The PN*-Search Algorithm: Application to Tsume-Shogi. *Artificial Intelligence*, Vol. 129, No. 1-2, pp. 253-277.

Slate, D.J. and Atkin, L.R. (1977). Chess 4.5-the Northwestern University University chess program. In Frey, P.W. (ed.), *Chess Skill in Man and Machine*, Springer-Verlag, New York, New York, pp. 82-118.

Thompson, K. (1996). 6-Piece Endgames. *ICCA Journal*, Vol. 19, No. 4, pp. 215-226.

Thompson, K. (1986). Retrograde Analysis of Certain Endgames. *ICCA Journal*, Vol. 9, No. 3, pp. 131-139.

Thompson, K. (2000). The Longest: KRNKNN in 262. *ICGA Journal*, Vol. 23, No. 1, pp. 35-36.

Thulin, A., (2000). John Thursby - Seventy-Five Chess Problems (1883), *electronic edition**.

Thulin, A., (2003). Frank Healey- 200 Chess Problems (1866), *electronic edition**.

Thulin, A., (2004). T.Taverner - Chess Problems Made Easy (1924), *electronic edition**.

Török, G. (2001). Karol Mlynka - 409 Selected Chess Problems. *Éditions de l'Apprenti Sorcier*. ISBN 0-9688828-7-0.

Török, G. (2002). Zoltan Labai - Selected Compositions. Volume 1 (1968-1982). *Éditions de l'Apprenti Sorcier*. ISBN 0-9688828-9-7.

Trice, E. (2004). 80-square Chess. *ICGA Journal*, Vol. 27, No. 2, pp. 81-95.

Turing, A. M. (1950). Computing Machinery and Intelligence. *Mind*, Vol. 59, pp. 433-460.

* downloaded from <http://www.algonet.se/~ath/>

REFERENCES

Turing, A.M. (1953). Digital Computers Applied to Games. *Faster Than Thought*, B.V. Bowden, Pitman, London, pp. 286-295.

Van Haeringen, H. and Van den Herik, H. J. (2003). Superchess. *ICGA Journal*, Vol. 6, No. 4, pp.239-250.

Watanabe, H. (1999). Master's Thesis: Artificial Creativity and Evolutionary History in Games, Graduate School of Industrial Science and Engineering, Shizuoka University.

Watanabe, H., Iida, H, and Uiterwijk J.W.H.M. (2000). Automatic Composition of Shogi Mating Problems. In Jaap van den Herik and Iida (eds.), *Games in AI Research*, Institute for Knowledge and Agent Technology IKAT, Universiteit Maastricht, pp. 109-124.

Internet Links

(Jan, 2006)

- [il-1] Blom, I., "Alybadix". <http://www.saunalahti.fi/~iblom/alybadix>
- [il-2] 2004. 12th World Championship in Computer Chess, held at Bar-Ilan University.
<http://www.cs.biu.ac.il/games/>
- [il-3] Leschemelle, M., "Problemist". <http://perso.wanadoo.fr/problemiste/>
- [il-4] Nowakowski J., "Chess Explorer". <http://www.geocities.com/explorer127pl>
- [il-5] Schnoebelen, P., Bartel. E., Geissler, N., Maeder, T., Linss, T., Hoening, S., Brunzen, S., Denker, H., Bark, T., and Emmerson. S. "Popeye".
<http://archiv.leo.org/pub/rec/games/popeye> or <http://www.uciengines.de/UCI-Engines/PopeyeUCI/popeyeuci.html>
- [il-6] Shogi Association. <http://www.computer-shogi.org>

Appendix A - The Game of Chess

The purpose of this appendix is to describe chess and its rules. The game of chess has simple and well defined rules. The game takes place on a 64-square board. There are two players, White and Black. Each player has six types of pieces at the beginning: king, queen, rook, bishop, knight, and pawn (Table 8). At the beginning, each side has eight pawns, two bishops, two rooks, two knights, one king and one queen. The initial position is in Diagram 11.













| | King | queen | rook | bishop | knight | pawn |
|-------|---|---|---|---|---|---|
| White |  |  |  |  |  |  |
| Black |  |  |  |  |  |  |

Table 8: The pieces in chess

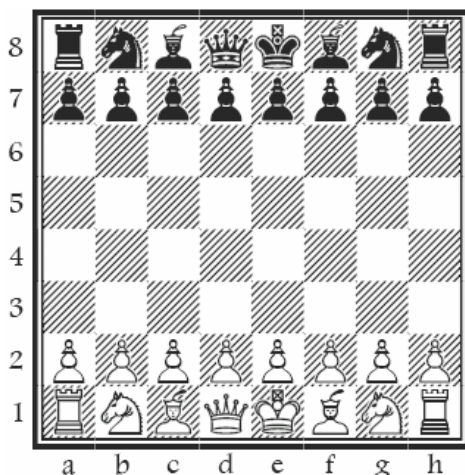


Diagram 11: The initial position

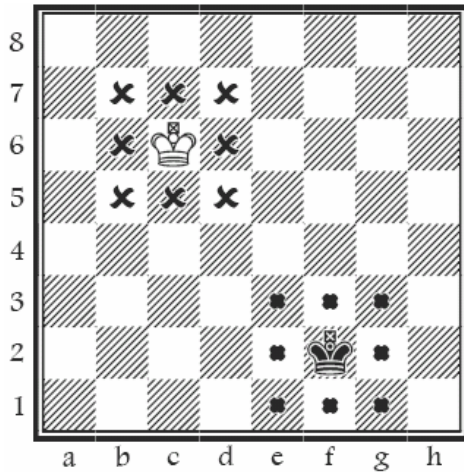


Diagram 12: King's moves

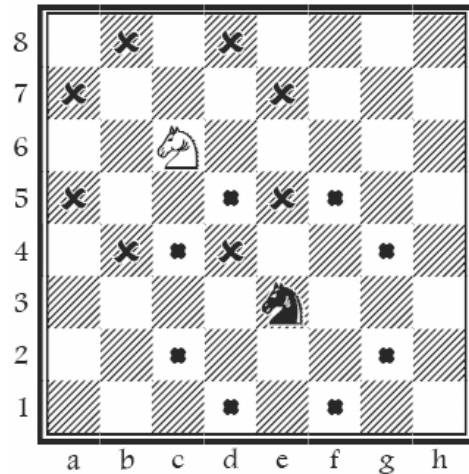


Diagram 13: Knight's moves

Diagram 12 shows all possible moves of the king. It can move only to neighbored squares. Knight's moves are observed in Diagram 13. Knight can jump over other pieces.

Pawn's moves are shown in Diagram 14. On the vertical "c", the regular pawn's move is shown. On the vertical "f", there is a special pawn's move named "jump". It can happen on the second horizontal for White and on the seventh horizontal for Black. According to the rules, a pawn may jump if there is no piece between its current place and the square the pawn jumps to. In the example, the pawn jumps from f2 to f4.

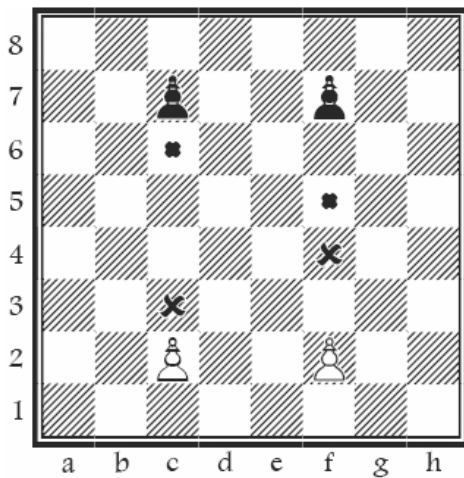


Diagram 14: Pawn's moves

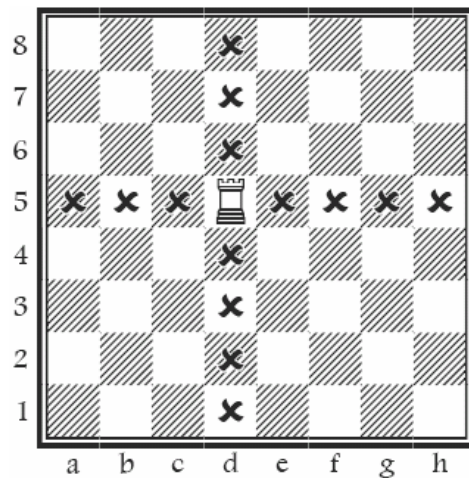


Diagram 15: Rook's moves

APPENDIX A – THE GAME OF THE CHESS

Diagrams 15, 16, and 17 present moves of sliding pieces. A rook may move in vertical and horizontal directions, as Diagram 15 shows. A bishop may move in diagonal directions (Diagram 16). Finally, a queen may move in all 8 directions, as Diagram 17 shows.

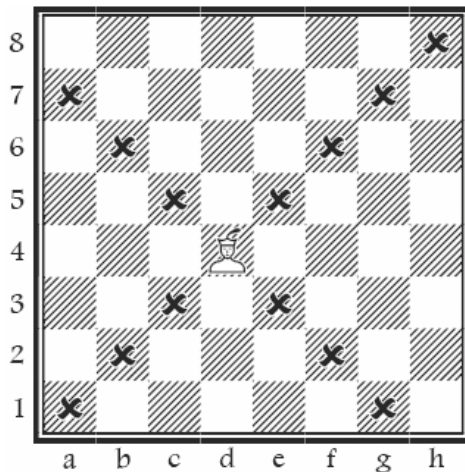


Diagram 16: Bishop's moves

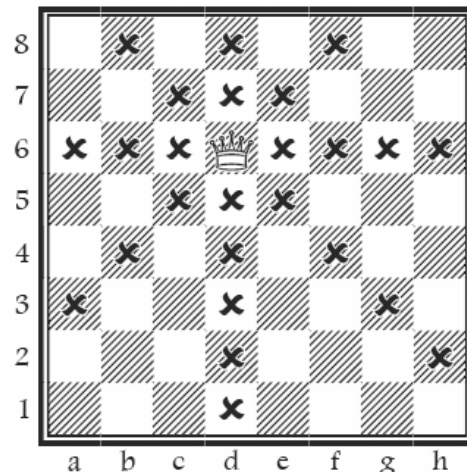


Diagram 17: Queen's moves

White and Black sliding pieces move the same. They cannot jump over other pieces as knights.

White pawns move always towards the eighth horizontal. Black's pawns move always towards the first horizontal. At the last horizontal (White at the eighth and Black at the first), a pawn becomes a knight, a bishop, a rook, or a queen by a decision of the player. The name of this rule is "promotion".

There is a restriction on all moves: all pieces may move only if the king of the color of the moved piece stay un-attacked. That is, no piece of the opposite site may move to the king's place. If a piece, other than a king, cannot move because of this rule, it is called "pinned". If a king cannot move, because it will be under attack, the attacked square is called "checked". Generally, a king under attack is called "checked". When it happens, the checked king's side must move and make the position without his king being checked. It is done either

APPENDIX A – THE GAME OF THE CHESS

by moving a king to a safe place, or by interfering with the checking piece, or by capturing the checking piece.

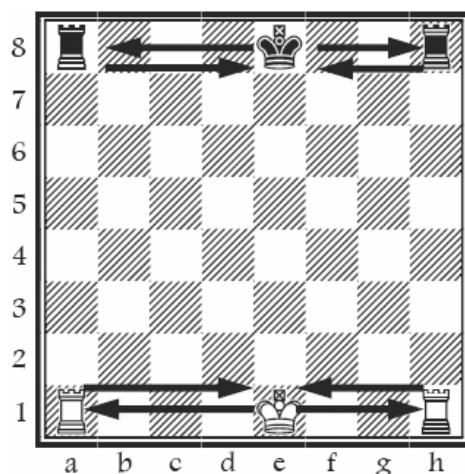


Diagram 18: Castling

A special move is castling (Diagram 18). Castling is a special interaction between a rook and a king of the same color. If neither the king nor the rook has moved from the beginning of the game, they can move simultaneously each one towards the other. There are 2 kinds of castling, the king-side and the queen-side. In the first case, the king moves from vertical "e" to vertical "g" and the rook moves from vertical "h" to "f". In the second case, the verticals are "e" and "c" for king, and verticals "a" and "d" for rook. The horizontals are the first for White and the eight for Black. Squares "e1", "f1" and "g1" must stay un-attacked if White makes a king-side castling. Squares "e1", "d1" and "c1" must stay un-attacked if White makes a queen-side castling. For Black, the un-attacked squares are the same and the horizontal is eight.

All pieces may capture an opposite piece. The capturing piece moves on the captured piece's place. The captured piece is removed from the board. Obviously, the captured piece is of the opposite color. The capturing side must stay his king un-attacked as in the case of

APPENDIX A – THE GAME OF THE CHESS

regular move. There is no jumping captures as in checkers. Capturing is not obligatory. A pawn captures on two nearest squares in the diagonal direction. For instance, the Black pawn on g3 may capture either the White bishop or the White knight (Diagram 19). A pawn may capture with promotion. For example, the White pawn on b7 (Diagram 19) may capture the Black rook on c8. There is no capturing while castling.

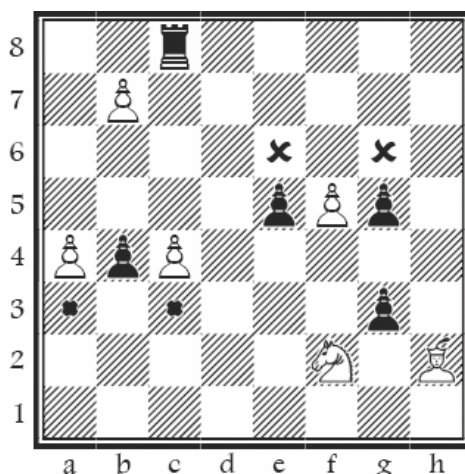


Diagram 19: Pawn's captures

A special capturing move is en-passant capture (Diagram 19). Here, both pieces, the rook and the king, move each one towards another. In case of White, if a Black pawn jumped from the seventh to the fifth horizontal, the White pawn may capture it on the next move. In case of Black, the en-passant horizontals are 2 and 4. For example, if the previous move was c2-c4 (Diagram 19), the next move of Black can be b4:c3, removing the White pawn on c4. Black, in this example, cannot capture the pawn on a4 because it did not jump on the previous move. If the last move of White were c3-c4, Black might not capture the White pawn on c4 too.

In the regular chess game, White moves first, starting at initial position (Diagram 11). Black moves afterwards, and so on until one of three happens: 1) White mates Black; 2) Black mates White; 3) a draw happens. The first situation happens when Black king is under

APPENDIX A – THE GAME OF THE CHESS

attack, it is Black's turn, and there is no legal answer of Black. The second situation happens when White king is under attack, it is White's turn, and there is no legal answer of White.

The third situation happens when it is one of the sides' turn, it cannot move, and there is no check. This situation's name is "stalemate". Draw occurs also when it is impossible to give mate to one of the sides. It happens when: (1) only two kings left, (2) there are three pieces on the board, one of the pieces is a knight or a bishop. When a position happened three times during a game, it is also considered as a draw.

We use capital letter for White's pieces and small letter for Black's pieces. Thus, "K" stands for "White king", "p" stands for "Black pawn", etc. "K", "Q", "R", "B", "N", and "P" are in use for king, queen, rook, bishop, knight, and pawn, respectively.

In this thesis, we use the following algebraic notation. We use letters a – h for verticals and number 1 – 8 for horizontals. To show that a Black knight moves from d8 to c6 on the first move, we use minus "-", and write 1. ... nd8 – c6. To show that a piece captures we use ":". So, "a White bishop moves from c4 and captures a Black knight on f1 on the second move" is written as 2. Bc4:f1. We also use "+" to represent a check, "++" to represent a double check, and "×" to represent a checkmate. However, we sometimes do not put "×" when we discuss mate variants.

Appendix B - The ICP Heuristic Function

Table 9 presents the 10 themes defined in ICP (HaCohen-Kerner et al., 1999). Models described in this thesis use new themes defined in Appendix C. The overall function is:

$$q_m = \begin{cases} 0 & \text{Severe deficiency} \\ \sum_i V(T_i) + \sum_j V(B_j) - \sum_k V(P_k) & \text{otherwise} \end{cases}$$

where the meaning is as follows. V is "value", T is "themes", B is "bonuses", and P is "penalties".

| Theme # | Composition Theme | Definition | Value in Points |
|---------|-----------------------|--|-----------------|
| 1 | Tempo or waiting move | The keymove does not threat any mate move | 10 |
| 2 | Direct battery | A battery with a White piece in the middle of the battery where a battery is defined as follows: a piece is standing between a long-range piece (queen, rook or bishop) and a king | 15 |
| 3 | Indirect battery | A battery to a square adjacent to the king's square | 25 |
| 4 | King-flights | The keymove creates free square/s that the Black king can escape to | 15 |
| 5 | Lonely king | The Black has only a king | 2 |
| 6 | Half-pinning | Two Black pieces standing between a White long-range piece (queen, rook or bishop) and a Black king | 25 |
| 7 | Self-pinning | The Black makes a move and pins the Black king | 15 |
| 8 | Unpin opponent piece | The Black makes a move and unpins a White piece | 20 |
| 9 | Self-blocking | A Black piece blocks another Black piece creating different mate variations | 25 |
| 10 | Grimshaw | Two Black pieces, each of which blocks the other's line, and causes different mate variations | 45 |

Table 9: ICP's themes

APPENDIX B – THE ICP HEURISTIC FUNCTION

Table 10 and Table 11 present bonuses and penalties used in ICP, respectively. The same bonuses and penalties were used in the models described at this thesis.

| Bonus # | Feature | Bonus in Points | Chess Composer's update |
|---------|---|---|--|
| 1 | Miniature | 10 | |
| 2 | Meredith | 5 | |
| 3 | Black king is in the center | 10 | |
| 4 | X pieces on board | $3*(18-X)$ | |
| 5 | Key by king | 15 | |
| 6 | Mate move by king | 20 | 20 for the first unique mate, 4 for the rest |
| 7 | Key gives X more king flights to Black | $15*X$ | |
| 8 | Key enables Black to check White X more times | $30*X$ | 30 for the first unique check, 6 for the rest (X-1) checks |
| 9 | Key pins a White piece | $3*\text{piece's value}$ | |
| 10 | Key unpins a Black piece | $5*\text{piece's value}$ | |
| 11 | Key sacrifices a White piece | $5*\text{piece's value}$ | |
| 12 | Mates' value | $\frac{10*(\# \text{ of diff mate moves})^2}{\# \text{ of variants}}$ | $\frac{10*(\# \text{ of unique thematic mate moves})^2}{\# \text{ of variants}}$ + $\frac{3*(\text{rest unique mates})}{\# \text{ of variants}}$ |
| | Distance of the keymove | | The distance is defined as a distance between start and end placement of the keymoved piece, as like it has been done by a queen (possibly in two moves). The formula is $3*(\text{diagonal part})+2*(\text{horizontal or vertical part})$ |
| | Distance to/from the Black king | | The difference in distances, in terms of 14, between the Black king's square and start and end squares of the keymove (if the result is positive, it's a bonus; otherwise, a penalty) |

Table 10: ICP's bonuses

| # of penalty | Feature | Penalty in points |
|--------------|--|--|
| 1 | X minor duals | X |
| 2 | X minor triples | 2*X |
| 3 | X minor multiples | 3*X |
| 4 | Black king is in the corner | 20 |
| 5 | Black king is in the edge | 10 |
| 6 | Key is a check | 50 |
| 7 | Key is a double check | 70 |
| 8 | Key is a capture of a Black piece | 10*piece's value |
| 9 | Key is a promotion of a pawn (unless it's the theme) | 5*piece's value |
| 10 | Key takes X king flights from Black | 15*X |
| 11 | Key pins a Black piece | 5* piece's value |
| 12 | Key unpins a White piece | 3* piece's value |
| 13 | If Value of White pieces > Value of Black pieces | 2* (Value of White pieces – Value of Black pieces) |

Table 11: ICP's penalties

Values of White and Black pieces are calculated due to Shannon's relative values for pieces (Shannon 1950) represented in Table 12. This values are suitable in chess composition too because relative strength or weakness of a particular piece takes a role too due to Harley (1931). However, there are other opinions about the values of pieces. For instance, Beal at al. (1997) built a system that was able to learn pieces value from a number of played games. The learned values varied as follows: pawn – 1 (other pieces are normalized to the pawn value), knight – 2.2 – 2.6, bishop – 2.8 – 3.2, rook – 4.0 – 4.4, and queen – 7.5 – 8.8.






| | queen | rook | bishop | knight | pawn |
|-------|---|---|---|---|---|
| White |  |  |  |  |  |
| Black | 9 | 5 | 3 | 3 | 1 |

Table 12: Shannon's relative values for pieces

Appendix C - Definitions of New Themes

Theme 1: "*Tempo*". Harley (1931) and Howard (1943) split all problems into two main groups, the *tempo* problems, and the *threat* problems. The *tempo* problems are those problems in which the keymove (see 2.2.1) does not *threat* any mate. The theme is called also the *waiting* move. The word "*threat*" means that White would mate Black if it would be White's move after the keymove. *Threats* are defined as an opposite of *tempo* problems. *Double threat* is two threats after the keymove. Finally, *multi threat* happens when there are more than two threats after the keymove.

Theme 2: "*King-flights*". King-flights are squares where the Black king may escape. According to Howard (1943), problems in which the number of king-flights does not change after the keymove are harder to solve. Therefore, their quality is higher.

Theme 3: "*Direct battery*". A *battery* is a placement of three pieces. One is the Black king. The second is a White piece which stands on the way between a White sliding piece and the Black king. Let us call the sliding piece to be the checking or firing piece, the second piece to be the middle piece. When the middle piece is moving, the checking piece "*fires*" and gives a check to the Black king. When it happens, the battery *fires*. Of course, a battery can be defined for the White king and Black pieces. This is not the case in the *direct battery* theme. In this thesis, direct battery happens when it is prepared before the mating move and it fires with the mating move.

Theme 4: "*Indirect battery*". This theme is similar to *direct battery*. The difference is the fired square. It is never the Black king's square but one of the squares near the Black king. It is also important that the fired square is not attacked by the other piece of White. In the last case there is no importance for the firing piece.

Theme 5: "*Self-blocking*". It happens when a Black piece moves and blocks possible flights of the Black king. White mates Black afterwards. The Black king could escape to the

APPENDIX C – NEW THEMES DEFINITIONS

blocked square if the Black piece was not there. If this is the case, than it is *self-blocking* theme.

Theme 6: "*Self-pinning*". As opposed to *battery* (see "direct battery" theme), in a pin, the middle piece is of the same color as the pinned king. If Black moved and pinned one of his pieces, this is potential *self-pining*. If there is a variant in which White mates and Black could prevent this mate by the pinned piece, than it is *self-pining* variant.

Theme 7: "*Half-pinning*". This theme is built upon the *self-pining* theme. Potential half-pin is a pin with 2 middle pieces. If both middle pieces take part in different *self-pinning* variants with different mates, than it is *half-pinning* theme.

Theme 8: "*Unpinning*". This theme happens when there is a Black piece which unpines a White piece. If the White piece mates, than it is an unpinning theme. The act of unpinning means that a piece moves and the opposite sliding piece can capture the unpinned piece.

Theme 9: "*Lonely king*". It is an example of the almost no-theme theme. This happens when Black has only a single king.

Theme 10: "*Grimshaw*". It is an example to a complex theme. In this theme, two Black pieces interfere with each other on the same square. The pieces are either sliding pieces (see Diagrams 4, 5, 6) or a jumping pawn. Therefore, by moving on the square they mutually prevent moves of each other. If this causes different mates, then it is the *Grimshaw* theme. Sometimes, if one of the Black pieces is a jumping pawn, than the theme is called *Pickabish*.

Theme 11: "*Pickaniny*". It is a theme that is created if there are 4 variants for a Black pawn after the keymove, i.e. two captures, moving, and jumping, and each variant finishes with a different mate. This happens only on the seventh horizontal.

APPENDIX C – NEW THEMES DEFINITIONS

All lines of play that help to recognize themes (except 1 and 9) are called "*thematic variants*". Duals in thematic variants decrease the quality of the problem. Without any connection of being thematic or not, duals can be *major* or *minor*. In a *minor* dual, each dual variant is forced in other lines of play. A *major* dual is an opposite of a *minor* dual.

All "pinning" themes have mirrored themes when pieces are of the mirrored color. There are hundreds and thousands of other themes as a fantasy of authors provides. After consulting the two international masters in chess composition that helped us, the following most common eleven themes was chosen. Table 13 shows the new scores (usually the same as in ICP).

| Themes # | Composition Theme | Value in Points |
|----------|-----------------------|-----------------|
| 1 | Tempo or Waiting move | 10 |
| 2 | Direct battery | 15 |
| 3 | Indirect battery | 25 |
| 4 | King-flights | 15 |
| 5 | Lonely king | 2 |
| 6 | Half-pinning | 25 |
| 7 | Self-pinning | 15 |
| 8 | Unpin opponent piece | 20 |
| 9 | Self-blocking | 25 |
| 10 | Grimshaw | 45 |
| 11 | Pickaniny | 25 |

Table 13: Scores for the new definitions

Notes:

- 1) Only unique mates are counted.
- 2) For the first unique thematic variant the full number of scores is given; for the rest, just

APPENDIX C – NEW THEMES DEFINITIONS

the fifth of the scores for the first variant is given.

- 3) Pins-related themes may be defined for opposite color pieces (Harley, 1931; Howard, 1943). However, the most frequent themes were defined in Table 13.
- 4) Thematic variants are not threats after the keymove.
- 5) *Tempo, king-flights, and lonely king* are special themes without thematic variants by definition (Harley, 1931; Howard, 1943).

Appendix D - 64-bit Representation

Let us use the following common denotations for the variables that represent the chess board:

| | |
|--------------------|--------------------|
| WP – White Pawns | BP – Black Pawns |
| WN – White kNights | BK – Black kNights |
| WR – White Rooks | BR – Black Rooks |
| WB – White Bishops | BB – Black Bishops |
| WQ – White Queens | BQ – Black Queens |
| WK – White King | BK – Black King |

Table 14: Primitives for the board representation

Table 14 presents the primitives that are enough to store the whole position. Each square in the table presents a certain type of pieces: pawns, knights, rooks, bishops, queens and king of both colors. In our program, the relationship between the board squares and its bits is as follows: a1,...,h1,a2,...,a7,...,a8,...,h8, where a1 is msb (bit number 63) and h8 is lsb (bit number 0). There are 12 64-bits words. Surely, we need to care about special cases of castling and en-passant capturing. That is because we need to remember the last opposite pawn's move in case of en-passant and whether a king or a rook moved in case of castling.

Using bitwise operations "|" (or) and "~" (not), we can immediately get 4 helpful 64-bit variables. These are "all White pieces", "all Black pieces", "all pieces", and "empty squares" (see Table 15). The rest helpful variables on Table 15 are pre-calculated 64-bits. "1" and "8" mean a 64-bit word in which bits are on the first and the eights horizontal respectively; bits are off on other horizontals. "a" and "h" are the same for verticals. By "~" before a number or

APPENDIX D – 64-BIT REPRESENTATION

a letter "n", we mean another 64-bit word which is bitwise operation "not" on the "n" (also pre-calculated).

| | |
|---|-------------------------|
| AW = WP WN WR WB WQ WK = all White pieces | |
| AB = BP BN BR BB BQ BK = all Black pieces | |
| AP = AW AB = all pieces | |
| NAP = ~AP = empty squares | |
| 1 – first horizontal | a – vertical a |
| 8 – eight horizontal | h – vertical h |
| ~1 – all but first horizontal | ~a – all but vertical a |
| ~8 – all but eight horizontal | ~h – all but vertical h |

Table 15: Helpful variables

Using primitives from Table 14 and helpful variables from Table 15 in addition to other bitwise operations "<<" (shift left), ">>" (shift right), "&" (and), we can easily get all pawns moves and captures (see Table 16). For example, all pawns' captures to the left, *clPW*, we get by ANDing all White pawns, *WP*, with the negation of the vertical "a", *~a*, shifting the result one vertical left and ANDing again with all pieces of Black (see line 5 in the Table 16).

We distinguish between "captures" and "regular" moves. This is because when a piece "captures", it influences the opposite side too. The promotion to the eight horizontal (or the first for Black) is also a special case. This is because 4 new positions (a pawn can be promoted to 4 other kinds of pieces) are created instead of one in regular move.

| Pawns | White | Black |
|---|---------------------------------------|---------------------------------------|
| Pawns' moves | $mPW = (WP \gg 8) \& NAP$ | $mPB = (BP \ll 8) \& NAP$ |
| Pawns moves, no promotion | $mPW \sim 8 = mPW \& \sim 8$ | $mPB \sim 1 = mPB \& \sim 1$ |
| Pawns jumps | $jPW = (mPW \gg 8) \& NAP$ | $jPB = (mPB \ll 8) \& NAP$ |
| Pawns captures to the left | $clPW = ((WP \& \sim a) \gg 7) \& AB$ | $clPB = ((BP \& \sim a) \ll 9) \& AW$ |
| Pawns captures to the right | $crPW = ((WP \& \sim h) \gg 9) \& AB$ | $crPB = ((BP \& \sim h) \ll 7) \& AW$ |
| Pawns captures to the left, no promotion | $clPW \sim 8 = clPW \& \sim 8$ | $clPB \sim 8 = clPB \& \sim 1$ |
| Pawns captures to the right, no promotion | $crPW \sim 8 = crPW \& \sim 8$ | $crPB \sim 8 = crPB \& \sim 1$ |
| Pawns moves, with promotion | $mPW+8 = mPW \& 8$ | $mPB+1 = mPB \& 1$ |
| Pawns captures to the left with promotion | $clPW+8 = clPW \& 8$ | $clPB+1 = clPB \& 1$ |
| Pawns captures to the right with promotion | $crPW+8 = crPW \& 8$ | $crPB+1 = crPB \& 1$ |
| enpassant | Special case | Special case |

Table 16: Pawns' moves and captures

A different approach is used for knights and kings. To get all legal moves and captures we use masks, *maskN* for knights and *maskK* for kings (Table 17). The color in this case makes no difference. Masks are arrays of size 64 of 64-bits words. Each mask represents moves of a piece. A 64-bits word on index n of an array means all moves of the piece if it would be on the square n.

| | | |
|---------------------------|--|--|
| All knight's moves | $\text{amWN} = \text{maskN}[\lambda(\text{WN})]$ | $\text{amBN} = \text{maskN}[\lambda(\text{BN})]$ |
| Knight's moves | $\text{mWN} = \text{amWN} \& \text{NAP}$ | $\text{mBN} = \text{amBN} \& \text{NAP}$ |
| Knight's captures | $\text{cWN} = \text{amWN} \& \text{AB}$ | $\text{cBN} = \text{amBN} \& \text{AB}$ |
| All king's moves | $\text{amWK} = \text{maskK}[\lambda(\text{WK})]$ | $\text{amBK} = \text{maskK}[\lambda(\text{BK})]$ |
| King's moves | $\text{mWK} = \text{amWK} \& \text{NAP}$ | $\text{mBK} = \text{amBK} \& \text{NAP}$ |
| King's moves | $\text{cWK} = \text{amWK} \& \text{AB}$ | $\text{cBK} = \text{amBK} \& \text{AB}$ |

Table 17: Use of masks for knights and kings

To be able to use masks, we use a λ (bitword) function ('bitword' is parameter) that returns the number of the least significant bit that is on. One of the possible implementations of λ is using of look-up tables. However, on some platforms there is a faster way to calculate λ . For example, on Intel Pentiums and Amd64 processors there is a special instruction *bsf* which is faster than the use of look-up tables. The disadvantage of the look up tables is their memory space consuming. So, the whole table will not fit in the cache and thus the use of it can be potentially slow. On architectures without this special instruction the following code can be used instead of look up tables:

```
typedef unsigned long long uINT64;
typedef long long INT64;
inline int  $\lambda$  (const uINT64 a){
    return (0==a)?64:ilogb(lsb_bit(a)); }
```

Where *lsb_bit* is defined this way:

```
inline uINT64 lsb_bit(const uINT64 a){
    return a & -(INT64)a;
}
```

The idea is simple: *lsb_bit* gives the nearest to the lsb bit which is on. The number which has just 1 bit on is a power of some number. So, we just find the binary logarithm. 'ilogb' is a relatively fast function.

Sliding pieces are carried out differently. The reason is the ability of existing of a piece on the way of sliding. So, in addition to pre-calculated masks in the sliding direction ("rays" in Table 18), we need to cut all squares behind the blocking piece. The solution for down and up directions in case of White rooks is shown on Table 18. μ is the number of the most significant bit that is on. Similar to λ , there are architectures with the special instruction *bsr* for μ . For the architectures without it, the following code can be used:

```
inline int  $\mu$  (const uINT64 a){
    return (0==a)?64:ilogb(a);
}
```

| | |
|-------------------------------|--|
| Rooks | For each White rook |
| Masks | dr = down_ray(λ (WR); ur = up_ray(λ (WR); rm = rankMoves[λ (WR)][(char)(AP >> number of rank)] |
| Moves down | mR = dr & ((lsb_bit(dr & AP) << 1) - 1) |
| Moves up | mR = ur ^ up_ray[μ (ur & AP)] |
| Moves left & right | mR = rm |
| just moves | jmR = mR & NAP |
| just captures | jcR = mR & AB |

Table 18: Moves and captures by sliding pieces via example of White rooks

APPENDIX D – 64-BIT REPRESENTATION

For the left and right directions, the placement of a rook and all other pieces on the rook's horizontal determine all moves of the rook. So, we can pre-calculate all possible situations and to store them into 2-dimensions array (rankMoves[64][256]) similar to Heinz (1997) and Hyatt (1999). In addition to left-right directions, they also pre-calculate all other three pairs of directions for all sliding pieces. In purpose to use these pre-calculations, Heinz and Hyatt use so-called rotated and flipped bitboards. The disadvantage of the method is managing of additional bitboards. We have no use of flipped and rotated bitboards.

Black rooks are carried out the same as White rooks. We achieve bishops' moves and captures in the same way as for rook, but using 4 diagonal rays instead of vertical and horizontal. Queens are carried out the same as bishops and rooks together.

Using the same technique as for queens in addition to knights we determine whether a king is still under attack and thus, the position is illegal. In this way, we cancel illegal positions fast.

Using the same method, we check the "can't castle" issue for 5 squares for each color: c1, d1, e1, f1, g1 for White and c8, d8, e8, f8, g8 for Black. In addition, we must remember if a castling king or a rook moved at the past. As in the case of en-passant capture, the cost is an addition of a word to the position structure (Table 14).

Appendix E - Results of the Improvement Processes

Example 1

Problem number is 4

White: Kd8 Qb5 Bb3 Pf4 **Black:** kd6 ba8 pc7

The **keymove** is: Qb5-c4

ORIGINAL POSITION EVALUATION:

THEMES:

threat: Qc4:c7

selfblocks: 3 score: 35 (price is 25 for the first variant, 5 for the rest)

grimshaw pairs: 1 score: 45 (price is 45 for each occurrence)

Total score for themes :---> 80

BONUSES:

position size: miniature score: 10

pieces on the board: 7 score: 33 (price is due to $3 * (18 - \#pieces)$)

*** different thematic mates: 3, different nonthematic mates: 2, variants:9

score: 10 (price is due to $(10 * \#diff\ thematic\ mates^2 + 3 * other\ diff\ mates) / \#variants$)

keymove distance: 3 (price is due to horizontals & verticals 2, diagonals 3)

Total score for bonuses :---> 56

PENALTIES:

pieces value, white:13, black: 4 score: 18 (price is due to $2 * (whites - blacks)$)

Total score for penalties:---> 18

POSITION SCORE:--->118

VARIANTS:

1) pc7-c6 Qc4-d4 X self-blocking; grimshaw

2) pc7-c5 Qc4-e6 X self-blocking

3) ba8-c6 Qc4-b4 X self-blocking; grimshaw

4) ba8-d5 Qc4:d5 X

APPENDIX E – RESULTS OF THE IMPROVEMENT PROCESSES

THEMES:

tempo (waiting) score: 10 (price is 10)

direct battery uniquely fired: 1 times score: 15 (price is 15 for the first thematic variant,
3 for the rest)

king-flights score: 15 (price is 15 for non-decreasing)

selfblocks: 2 score: 30 (price is 25 for the first variant, 5 for the rest)

pickaniny pawns: 1 score: 25 (price is 25 for each occurrence)

Total score for themes :---> 95

BONUSES:

position size: Meredith score: 5

black king is in the center score: 10

pieces on the board: 10 score: 24 (price is due to $3 * (18 - \#pieces)$)

mate move is by king: 20 score: 20 (price is 20 for the first variant, 4 for the rest)

*** after keymove sacrificed pieces on squares: d6 f6 e4

sacrificed pieces: 3 score: 5 (price is due to $1 * \text{sacrificed piece value}$)

*** different thematic mates: 4, different nonthematic mates: 0, variants:5

score: 32 (price is due to $(10 * \#diff \text{ thematic mates}^2 + 3 * \text{other diff mates}) / \#variants$)

keymove distance: 5 (price is due to horizontals & verticals 2, diagonals 3)

Total score for bonuses :--->101

PENALTIES:

pieces value, white:25, black: 1 score: 48 (price is due to $2 * (\text{whites} - \text{blacks})$)

keymove to king distance: 5 (price is due to horizontals & verticals 2, diagonals 3)

Total score for penalties:---> 53

POSITION SCORE:--->143

VARIANTS:

- | | | | |
|-----------|----------|----------------|-----------|
| 1) pe7-e6 | Bd7-c6 X | self-blocking | pickaniny |
| 2) pe7-e5 | Qc3-d3 X | self-blocking | pickaniny |
| 3) pe7:d6 | Kg5-f4 X | direct battery | pickaniny |
| 4) pe7:f6 | Ng8:f6 X | pickaniny | |

Appendix F - Example of applying order on additions of White pieces

Below, it is an example of code implementing an order on pieces. The explained in chapter 3.2 order is Q (White queen) > R (White rook) > B (White bishop) > N (White knight) > P (White pawn) > q (Black queen) > r (Black rook) > b (Black bishop) > n (Black knight) > p (Black pawn). The purpose of the code below is to insert clearance into the question how the order on the pieces can be implemented. The order on the Black pieces is the same.

```
int make_addition_of_white_transformations(const struct TRANS *prev_trans,
    const bool chess_comp_rules, struct TRANS * goal_trans_positions)
{
    int created_pos = 0;
    char order_addition_piece = prev_trans->order_addition_piece;

    if(order_addition_piece <= en_white_queen_addition)
    {
        // white queen, rook, bishop, knight, pawn are added
        created_pos += make_addition_transformations(prev_trans, 4, true,
            chess_comp_rules, goal_trans_positions + created_pos);

        if(order_addition_piece <= en_white_rook_addition)
        {
            // white rook, bishop, knight, pawn are added
            created_pos += make_addition_transformations(prev_trans, 3, true,
                chess_comp_rules, goal_trans_positions + created_pos);

            if(order_addition_piece <= en_white_bishop_addition)
            {
                // white bishop, knight, pawn are added
                created_pos += make_addition_transformations(prev_trans, 2,
                    true, chess_comp_rules, goal_trans_positions);

                if(order_addition_piece <= en_white_knight_addition)
                {
                    // white knight, pawn are added
                    created_pos += make_addition_transformations(prev_trans, 1,
                        true, chess_comp_rules, goal_trans_positions +
                            created_pos);

                    if(order_addition_piece <= en_white_pawn_addition)
                    {
                        // white pawn is added
                        created_pos += make_addition_transformations(prev_trans,
                            0, true, chess_comp_rules, goal_trans_positions +
                                created_pos);
                    }
                }
            }
        }
    }
    return created_pos;
}
```