Yugoslav Journal of Operations Research
14 (2004), Number 2, 273-288

REALIZATION OF THE CHESS MATE SOLVER APPLICATION

Vladan V. VUČKOVIĆ
Faculty of Electronic Engineering,
University of Niš, Niš, Serbia & Montenegro
vladan@bankerinter.net

Received: July 2003 / Accepted: June 2004

Abstract: This paper presents details of the chess mate solver application, which is a part of the author's *Geniss* general chess application. The problem chess is an important domain connected with solving of the chess problems. The *Geniss Mate Solver (G.M.S.)* application solves *Mate-in-N-move* problems. Main techniques used for the implementation of the application are full-width searching with Alpha-Beta pruning technique and zero evaluation function. The application is written in *Delphi for Windows* programming environment and the searching engine is completely coded in assembly language (about 10000 lines). This hybrid software structure enables efficient program development by using high-level programming environment and the realization of a very fast searching engine at the same time. The machine code is manually coded and could achieve above 7 million generated positions per second on the 1Ghz Celeron PC.

Keywords: Computer chess, game tree searching, alpha-beta optimization algorithm, decision theory.

1. INTRODUCTION

The problem chess is one of the important fields of a chess game regardless of its relative minor popularity compared to classic chess. It comprises some interesting disciplines like self-mates, helpmates or mate-in-N-move problems. The common characteristics of these disciplines and the main goal of the problem chess are to find the shortest sequence of moves providing a mate against the opponent king, commonly black one. In the great amount of problems the white side is on the move trying to mate the black king with or without opponents help. Composition and solving of the chess problems require great imagination and creativity (Grand, 1986; Schlosser, 1988; Wiereyn, 1985.) Problem chess tournaments (including the World Championship) are constantly organized and solvers are ranked. The most important difference between classic and problem chess is that in problem chess solver tries to solve the position on the board in specified number of moves; he does not play against an opponent. That fact is very important to simplify procedures and requirements when we try to formalize problems for the computer solver (Beal and Botvinnik, 1986).

Introducing computers into the problem chess is useful in different ways. Firstly, computer could be a perfect assistant in solving and composition of chess problems; checking all possible variants and finding the alternate combinations, if they exist. Secondly, the problem is much simpler then the situation is classic chess, and so the implementation is easier and the search engine could achieve higher efficiency (PPS factor). On the other side, general approaches suggest using of a brute-force searching techniques with the purpose of finding all possible variants - solutions in concrete chess problem. Of course, in such a situation the exponential explosion of the chess tree requires fast searching algorithm and efficient implementation in low-level programming language.

This paper treats suitable solution of the above mentioned problems realized in G.M.S. application. The composition of the paper is as follows: The definition of problem chess is given in Section 2. In Section 3. different searching techniques are described with an accent on the Alpha-Beta pruning as one of the most important parts of the search engine. In Section 4. details of the basic procedures are presented and some original approaches and solutions in application implementation are described. the implementation and the options of the G.M.S. are presented in Section 5. In Section 6. some experiments are performed to illustrate some abilities of the application.

2. THE PROBLEM CHESS

As it was mentioned, problem chess is a special category of general chess concerned mostly with searching for mates in specified number of moves. Positions in chess problems are composed and design connected with some main idea and could be rarely created by accident, in some tournament games. For instance the diagram in Figure 1. presents the 3-move problem (mate in 3). The former world chess champion Alexander Alekhin composed this problem (it is the only chess problem he had ever created). The right solution is a beautiful queen sacrifice Qf5!! with the principal variation: 1. Qf5 B:f5 2. Ra7 K:e6 3. Nf4++. The key move (Qf5) applied in this position is not direct threat - check, capture, king attack or mate treat. Also it is “clear” queen sacrifice “obviously” creating the lost position for White after Black bishop captures White Queen. The radius of piece movement is short, just one square. But all of these conclusions are based on some static characteristics. In fact, in the view of future dynamic occasions, the position is lost for Black.

- It is obvious that intelligent and sophisticated move generator would hardly select move like Qf5 in its searching list, especially deeper in search tree. Also, similar moves could later appear in combination. These facts implicate that the safest way to design chess problem solving algorithm is to *include all legal moves* for both sides into the calculation - to implement Shannon type-A brute-force searcher. This method does guarantee that all variants (including all sorts of sacrifices, tactical and positional moves) will be examined to the fixed depth and all possible mates, if they exist, be found in that range. It is very simple to prove that the solution of chess problem (Figure 1.) can be found automatically by searching all possible moves and replies by White and Black to the specified depth (3 moves apropos 6+1 ***PLY*** where ***PLY*** is search depth in moves (depth=2*ply-1)).

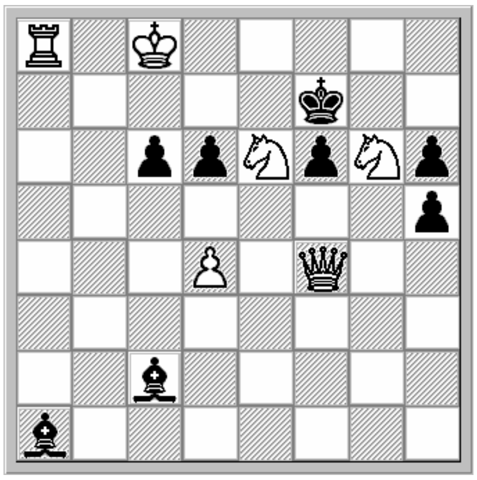


Figure 1: Alekhine problem. Queen sacrifice Qf5!! leads to the mate in 3 moves.

There are many possible ways to implement data structures and procedures to perform this brute-force type of algorithm. We shall demonstrate how these implementation problems were solved in G.M.S. application.

3. THE SEARCHING TECHNIQUES

This Section presents some basic procedures and algorithms needed for construction of the problem chess solver. The main procedures used for the implementation of G.M.S. searcher are MiniMax and fixed-depth (full-width) algorithm with Alpha-Beta pruning technique. Some of the theoretical concepts of these procedures are also presented in sequel.

3.1 MiniMax and NegaMax

The MiniMax procedure determines which move is the best at some level of chess game tree. After evaluation of all legal continuations from some tree node, the best move is chosen as a move with maximum value of the available evaluations if the White side is on the move and with minimum value if the Black side is on the move. Because of the nature of chess game, tree search is an alternation between maximizing and minimizing the evaluations, so the operation is often called *MiniMaxing*. To integrate two procedures, the value of a position is often evaluated from the standpoint of a player to move, so the opponent evaluation is negated. This procedure is called *NegaMaxing*. The standard procedure is illustrated by the following pseudo-Pascal code:

```
procedure NegaMax (Position, Depth): integer;
var …
  { Position – Current position }
  { Depth – Depth in chess tree }

begin
   if (depth=0) then
     begin
          Evaluate(Position);    { Base node evaluation, exit from procedure }
          Exit;
    end;
   best: = -INFINITY;              { Setup for maximizing }
   succ: = Successors(Position);    { Find successor }
   while not Empty(succ) do        { Until all legal succesors are processed }
   begin
         Position: = RemoveOne(succ);              {Remove one succesor from the list }
         Value: = -NegaMax(Position, depth-1);   {Recursive call with negated value
                                                    (depth-1 !)}
         if  (value > best) then best: = value;      { New best move }
   end;
NegaMax:=best;    { Return Maximum (Minimum) }
end;
```

Figure 2: NegaMax standard recursive algorithm.

To avoid the recursive call which always generates some technical problems in programming, especially with assembly language, the classic MiniMax procedure is used in G.M.S application.

3.2. Full-width search

Machine representation of a chess game is based on decision trees theory (Bruin, A. de, Pijls, W. and Plaat, A., 1994). Tree search is one of the central algorithms in a chess game-playing program. The term is based on looking at all possible game positions as a tree. The legal chess game moves create the branches of a game tree. Each possible position represents tree node. The leaves of the tree are all final positions, called terminal positions, where the consequence (evaluation) of the game is known. If all legal move sequences in game tree are searched to the fixed depth the strategy is called *full-width* searching. The synonym for that kind of search is *brute-force* searching.

The main problem in machine chess tree representation is that size of the chess tree is extremely large, approximately like W^D, where W is the average number of moves per position (width) and D is depth of the tree. For instance, in the opening phase of a chess game with average 25 moves per position, after depth 8 (combinations with only 4 moves ahead) around 153 billion terminal positions are to be evaluated. If we presume that processing of a single position lasts one microsecond, the complete calculation will consume about 42 hours!

Obviously, searching of the entire game tree is not rational even on the fastest computers. All practical search algorithms are approximations of doing such a complete tree search.

3.3. Selective search

Different methods for reducing and pruning are invented with an aim to avoid the calculation of complete chess tree. Some of these methods are applied in chess searching algorithm and the strategy is called *selective-search*. Some of the most prominent techniques are forward-pruning, different kinds of knowledge based heuristic pruning, best-n-moves pruning techniques based on high-quality plausible move generators (Fray, 1977,1978) and null-move pruning (Donninger, 1993). If we compare full-width search and selective-search (Kaindl, Horacek and Wagner, 1986) the most important conclusions could be accomplished:

- Selective-search could significantly reduce the magnitude of the game tree suppressing the exponential explosion. For instance, if plausible-move generator selects the 10 best moves at each node, the depth 8 search will produce about 100 million positions. This is 0.00065% of the tree size generated by 8 ply full-width searching example in previous section.

- The selective algorithms must contain large portions of expert chess knowledge to produce efficient selection of plausible moves. Unfortunately, the selecting and programming the expert knowledge into the concrete application is very difficult task considering the complicity of the chess game itself. Regardless the quality of the move selector procedure there is always some probability that some excellent move (piece sacrifice etc.) could be pruned generating the gross errors in move selection. Many researchers have abandoned the heuristic and knowledge-based selective search mostly due to the potential hazard of those techniques.

- The most promising selective searching techniques nowadays are based on null-move heuristic. This technique is not based on specific chess knowledge implementation and could be applied with other logic games. The main characteristic of null-move searchers is possibility to prune efficiently unreal lines of the game tree. In combination with the chess knowledge implemented in evaluation function (Althöfer, 1991,1993), those kinds of searchers could achieve very high quality of play.

- Full-width searchers avoid the problems of move selecting because all legal moves are plausible and all combinations are examined to the fixed depth. Two main principles have to be achieved: *good move ordering* and *fast node processing*. The programming of the full-width searcher is much simpler compared with selective searcher. The main shortcoming of the technique is the request for extremely efficient programming (in machine language) or specialized hardware.

The experience does not confirm the clear priority of any mentioned techniques. Both approaches have enabled building machines and achieving the grandmaster strength. The most powerful brute-force machine IBM Deep Blue calculated 200 million positions per second and won the match again the World Chess Champion GM Garry Kasparov in 1997. Last two matches between Chess Champions GM Vladimir Kramnik and GM Garry Kasparov against PC based parallel chess computers Deep Junior and Deep Blue respectively both ended draw. The machines used selective, null-move-based algorithms achieving about 3-4 million positions per sec.

3.4. Alpha-Beta pruning technique

The combinatory explosion problem connected with game decision trees was noticed early in Shannon works (Shannon, 1949,1950). The Alpha-Beta algorithm is the first significant effort and contribution with the purpose of reducing the number of positions that has to be searched. This technique enables achieving greater depths in the same amount of time without the decreasing of a play quality. The main idea is that large parts of a tree do not influence to the best move decision and that could be pruned without any affect on final result. The algorithm does not require all terminal positions to be evaluated because cut-offs are generated earlier in game tree. The value of the position along the principal variation has to be exactly determined and other ones are interested only if they are better or worse than evaluation that has been done before.

(**Definition** - Principle variation is the alternation of the best own moves and best opponent moves from the root to the depth of the tree).

The Alpha-Beta search procedure gets two main arguments (Alpha,Beta) indicating the bounds where the exact values for a position have to be searched. The illustration of the standard Alpha-Beta algorithm is presented in sequel:

```
procedure AlphaBeta (position, depth, alpha, beta):integer;
var …
  { Position – Current position }
  { Depth – Depth in chess tree }
  { Alpha – Upper bound }
  { Beta – Lower bound}

begin

if (depth=0) then
     begin
          Evaluate(Position);    { Base node evaluation, exit from procedure }
          Exit;
     end;
  best: = -INFINITY;                  { Setup for maximizing }
  succ: = Successors(Position);    { Find successor }

while ((not Empty(succ)) and ( best < beta)) do  {Searching while move list is not empty,
                                                   or best value is lower then bound}
```

```
begin
    position: = RemoveOne(succ);            {Removing one move}
    if (best > alpha) then alpha: = best;
    value: = -AlphaBeta(Position, depth-1, -beta, -alpha);  {Recursive call with negated
                                                              values of bounds}
     if (value > best) then best: = value;    {New best value}
end;
  AlphaBeta:= best;   {The exit from the procedure is best value}
end;
```

Figure 3: Fail-soft Alpha-Beta recursive algorithm

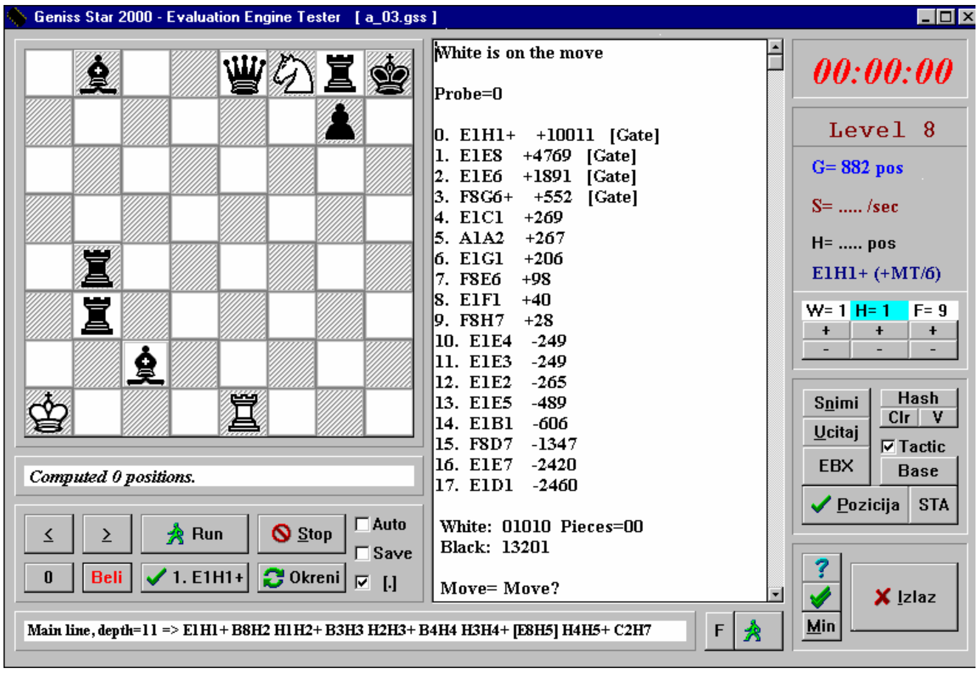
This version of algorithm is also known as fail-soft Alpha-Beta. It can return values outside the Alpha - Beta range that can be used as upper or lower bounds for researching. In combination with different memory hash schemas it could achieve high pruning efficiency. The gain from Alpha-Beta proceeds from the earlier exit of the move list-scanning loop. The best value that exceeds (or equals) Beta bound is called a *cut-off*. These cut-offs are completely safe because they indicate that this branch of the tree is worse than the principal variation and could never be played. *The largest gain is reached when at each level of the tree the best successor position is searched first.* The main reason for this conclusion is that the position generated by best successor will either be a part of the principal variation or it will cause a cut-off to be as early as possible. Under favorable circumstances Alpha-Beta has to search $W^{(D+1)/2} + W^{D/2} - 1$ positions (W is average number of legal moves per position and D is search depth). Of course, this is exponential dependence but much less than basic MiniMax search algorithm (W^D). It allows reaching about twice the depth in the same amount of time. To achieve maximum pruning efficiency move ordering has to be near perfect at every generated position in the game tree.

4. THE BASIC PROCEDURES

This Section reflects some functions and procedures that are implemented in G.M.S. application. Three basic procedures are abstracted: *move generator, searcher* and *evaluation function*. The theory and implementation reclines on methods and algorithms presented in previous sections.

4.1. Move Generator

One of the most important procedures in every chess program is move generator. This function generates all legal moves from current position into the move list. When the move list is formed the next step is to calculate move weights to achieve efficient move ordering. In G.M.S. application these two steps are integrated; all moves are generated simultaneously with their weights. The moves with greater weights will be examined firstly. Weights are tuned thus power king attacks, check moves, captures and mate treats get maximal values.



```
White is on the move

Probe=0

0.  E1H1+   +10011  [Gate]
1.  E1E8   +4769  [Gate]
2.  E1E6   +1891  [Gate]
3.  F8G6+   +552  [Gate]
4.  E1C1   +269
5.  A1A2   +267
6.  E1G1   +206
7.  F8E6   +98
8.  E1F1   +40
9.  F8H7   +28
10.  E1E4   -249
11.  E1E3   -249
12.  E1E2   -265
13.  E1E5   -489
14.  E1B1   -606
15.  F8D7   -1347
16.  E1E7   -2420
17.  E1D1   -2460

 White:  01010  Pieces=00
 Black:  13201

 Move= Move?

Main line, depth=11 => E1H1+ B8H2 H1H2+ B3H3 H2H3+ B4H4 H3H4+ [E8H5] H4H5+ C2H7
```

Figure 4: Move generator function, developing and testing application

Unsafe moves get negative weights. The secondary parameters for move ordering are pion promotions, double attacks (forks), opponent king proximity (especially for knights and queens) and piece position on the board. The Figure 4. illustrates how move generator performs one simple position. The key move, power check E1H1+ that leads to mate in 6, has maximum weight (+10011) and it is examined first. Otherwise, move weights are 16-bit integers within the range -32768..+32767. The good move ordering has direct positive consequence on searching speed. This implicates usage of large portions of heuristic knowledge into the move generator. Nevertheless, the move generator must be simple enough to attain high position per second rate. This confronted requests determine need to balance knowledge and performance in generator procedure. The G.M.S. application has good equilibrium between those two requirements. The assembly coded and manually improved move generator contains sophisticated heuristics and high execution speed at the same time. On test 1Ghz Celeron PC platform, move generator overtakes 7 million positions per seconds.

4.2. Searcher

Main procedure in every chess application is game tree searcher. It performs searching round the game tree in purpose to find best continuation for each side, using minimizing function. The G.M.S. tree searcher has the following characteristics:

- It is full-width Shannon type-A searcher. The orientation to brute-force type of searching is based on high move generator performances, described previously. This type of searching insures that all mate combinations, if exist, will be found within the pre-defined horizon. Nevertheless, the shortcomings in move ordering are not fatal - they influence only to the search efficiency. The desire for simplicity and efficiency was also additional arguments for choosing such a type of searcher.

- The quiescence search (Beal, 1984) beyond the horizon is not performed. This shallow search orientation enables to find mates precisely in pre-defined number of moves.

The variation of non-recursive Alpha-Beta pruning algorithm is embedded into the searcher. This technique is absolutely safe and in combination with fast and simple terminal evaluation function as well as near-perfect move ordering from move generator, it achieves very high pruning percentage. Because of the non-material nature of the evaluation function, Alpha-Beta cut-off and MinMax procedure could be integrated.

The searcher uses stack as the basic memory structure, which coupled with fast machine code access routines contributes to the high execution efficiency. The external hash structures are not supported. Some experiments in that direction confirm that external hash memory does not conduce to the searching efficiency because of the specific conception of the evaluation function. The searcher nucleus code (in Pascal) is enclosed in Appendix. This type of searcher performs deep first, full-width searching round the game tree. Procedure uses stack as the main data structure and assembly sub-procedures (GENERATOR, EVALUATOR, MOVE_FORWARD, MOVE_BACK) to accelerate tree searching. The searcher used in G.M.S. application is one variant of the illustrated algorithm therewith the machine code is used instead of Pascal for concrete implementation.

4.3. The Evaluation Function

An adequate circumstance with chess problem solver is that complicated evaluation of terminal positions could be avoided. Namely, in a chess game, mate is defined as the situation where king is in check and has not even one legal move to play. Searching for mates adds up finding those situations. In G.M.S. search procedure has responsibility to handle mate situations because it has information about the number of legal moves at each node in game tree provided by move generator procedure. Due to the fact that searching result must be "mate found" or "mate not found" *all terminal positions are evaluated to zero value*. There are two main implications of the terminal zero-evaluation. Firstly, evaluation function is extra-fast because it is reduced to only one machine instruction. Secondly, Alpha-Beta pruning mechanism becomes extremely efficient because all terminal positions have equivalent values. The status of non-terminal nodes is slightly different because searcher could find some mates upwards the game tree. MinMax node variable could get 3 different values -MATE, 0, +MATE, where MATE value indicates that mate for black/white was found. Experiments prove that usage of the rough evaluation with a few different values significantly increase number of pruning in game tree generated by Alpha-Beta algorithm. In classic chess application, the complex evaluation has to be used in purpose of computing the static and other non-material factors that add much more nodes into the game tree. The combination of extremely efficient zero-evaluator, machine coded near-perfect move ordering provided by move generator and Alpha-Beta pruning mechanism enables the realization of the G.M.S full-width searcher possible.

5. THE IMPLEMENTATION

Geniss Mate Solver (G.M.S) is the author's contribution to the problem chess programming. The most of the chess applications, amateur or professional are concerned with classic chess and only a few have implemented routines for chess problem solving. Some of the mate solver engines are embedded in the standard environments (Deep Fritz).

The G.M.S. application consists of two major parts. The interface is written in Borland Delphi programming language under *MS Windows* operating system. Program is compatible with all Windows OS-s up from *Windows 3.11*. This solution enables the usage of all accommodations provided by graphic operating system. The largest segments of the searching engine are implemented in assembly language containing about 10000 code lines. The core functions (move generator and searcher) are fully manually coded and debugged. Machine routines are connected with main Pascal program interface using ASM directives. Data transmission between two languages is realized through common (shared) memory structures. Thus, all system, interface and engine could be mutually developed and compiled using Delphi environment. The example of the G.M.S. hybrid software organization is presented in following listing:

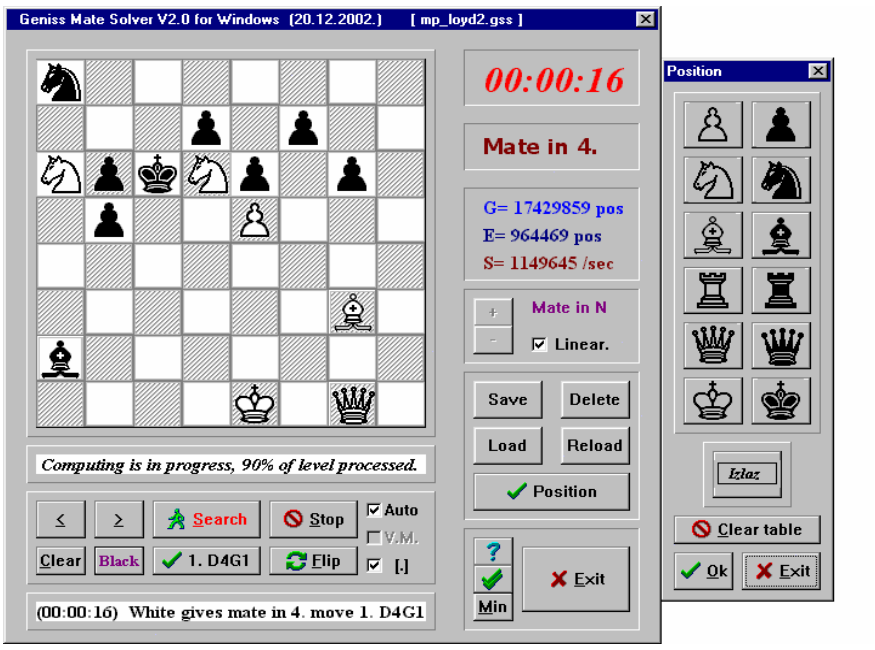
```
function save_combination(i:integer):integer;  {  I is input variable…  }
var s:integer;                                  { S is accessory variable… }
begin
asm             { From this point assembly language is used… }
  push ds       { Segment registers are pushed to the stack… }
  push es
    MOV AX,I    { 16-bit Pascal input variable is transferred directly to AX register}
  pop es
  pop ds        { Segment registers are loaded from the stack… }
  MOV S,AX      { The contains of the AX register is transferred to Pascal S variable…}
end;            { Back to Pascal code… }
  save_combination:=s;    {Now, the function gets the value of S variable…}
end;
```

Figure 5: Direct data transfer between Pascal and machine language (example).

This example is the prototype of connection between two languages used in realization of the G.M.S. and illustrates the transfer of 16-bit integer from the input variable I to SAVE_COMBINATION function via the assembly-coded segment. Of course, the assembly part of the procedure could process the input values using machine code instructions providing high efficiency in executing. The presented programming strategy that use Delphi Pascal for interface, organization and data structure handling, and the assembly language for chess engine implementation, proved to be appropriate for the realization of G.M.S. application.

5.1. The Interface and Main Application Organization

Application interface is presented in the following figure:



Geniss Mate Solver V2.0 for Windows (20.12.2002.) [mp_loyd2.gss]

00:00:16

Mate in 4.

G= 17429859 pos
E= 964469 pos
S= 1149645 /sec

Computing is in progress, 90% of level processed.

(00:00:16) White gives mate in 4. move 1. D4G1

Figure 6: Geniss Mate Solver v2.0 – main application interface

The main window is separated into the four main sections: chessboard, move control buttons, report arrays and piece palette. The chess engine is integrated with the interface in common executable file. All standard options connected with position and combination management are supported. The use of application generally follows 3 phases:

- Set-up of the start position using piece palette (Figure 6, right window). Positions also could be stored or loaded from disk,
- Starting the solver engine,
- Analyzing of the results (if the combination was found).

The example in Figure 6. shows the solution of the Sam Loyd's problem, Qd4-g1!!. This extraordinary move is the only solution for that mate-in-4 problem.

5.2. The Mate Solver Engine

The many details of the realization and implementation of the G.M.S. solver engine are specified in previous sections so only the most important characteristics will be emphasized:

- G.M.S. uses full-width, Shannon type-A searching algorithm. All legal moves are examined to the pre-defined depth that increases in each iteration. Program solves all type of chess problems in minimal number of moves.
- Alpha-Beta pruning technique is used. In combination with zero-evaluation function it achieves high pruning ratio.
- Zero evaluation function includes only one machine instruction so it is extremely fast. Move generator function has responsibility to generate all legal moves with their weights enabling the efficient move ordering. If move generator detects check and no legal moves existing, the mate situation is signalized.
- The engine is completely programmed in machine language maximizing the usage of CPU. The code is the mixture of 16 and 32-bit instructions using the segment-oriented addressing. The tests demonstrate that AMD Athlon XP family of processors slightly dominates over Intel P4 running that kind of code.

6. EXPERIMENT AND RESULTS

Some comparative results confirm that G.M.S. running on average PC platform easily exceeds strong human opponents in chess problem solving (limited time conditions). It is the main reason why this or similar application (Lindner, 1983,1985) will not be allowed to participate and compete against human opponents in regular problem solving tournaments.

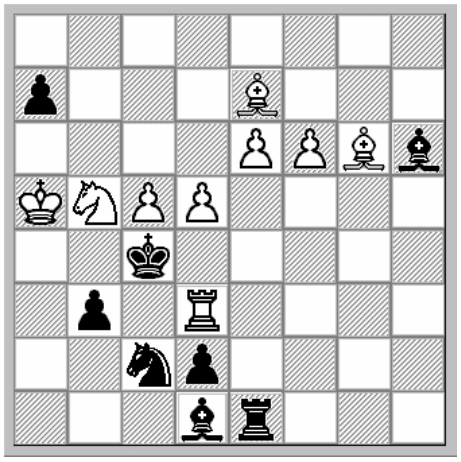


Figure 7: Mate-in-6 problem (example)

The efficiency and accuracy of G.M.S. is tested on many chess problems (*Chess Informant, 2000*). The program has solved all samples, finding some extra solutions, if they were existed. For the illustration, one mate-in-6-moves problem is chosen (Figure 7):

Table 1: Experiment results

PLY	GEN	EVL	S (G/T)	T (sec)
1	922	28	-	<< 1 sec
2	24034	678	-	<< 1 sec
3	320008	8463	-	< 1sec.
4	3973630	103514	4077144	1
5	44689880	1141950	7638638	6
6	99323877	2457682	7270111	14
-	-	-	-	-
1-6	148332046	3712315	7240207	21

The solution (1. Nd6+ Kc5 2. Nb5 Kc4 3. Bf5! Re3 4. Nd6+ Kc5 5. Ne4+ Kc4 6. Nd2++) is complicated containing some unexpected, maneuvering moves. The experiment is conducted using G.M.S. v2.0 application. The searching depth is increased in each iteration, from 1 to 6. The first found mate sequence is the problem solution. The hardware used in experiment is PC 128Mb platform with 1Ghz Pentium Celeron CPU. The results of experiment are shown in Table 1. The symbols denoted in table header have following signification:

- PLY - search depth in moves (depth=2*ply-1),
- GEN - number of generated positions,
- EVL - number of evaluated terminal positions,
- S (G/T) - generated positions per second,
- T - total time for computing (in seconds).

The solution is found on depth 6, computing 10% from the total chess tree owing to good move ordering. The analyze of the experiment results refers to some major conclusions:

- The overall application efficiency is very high. On depth 6, the program achieves about 7.3 million generated positions per second and about 175548 terminal positions/evaluations per second.
- Depending on position, all 2-4 movers are solved within a seconds and 5-6 movers rarely exceeds a minute to be solved. For the complex (tournament) set of problems G.M.S. attains much better time solving all proposed problems then strong human problemist.
- The chess tree grows exponentially. Due to the efficient pruning techniques and near-perfect move ordering the exponent is decreased but the combination explosion problem still remains. For instance, for the complete computation on ply=5 (depth=9), G.M.S. has generated about 44.7 million positions and 1.1 million terminal nodes in game tree. If we suppose that each node of game tree has 20 legal moves; the total terminal nodes at depth 9 will be roughly 20^9=512 billions! Due to the efficient pruning only about 0.0002% of the total tree size is processed.

- All legal moves are examined to the fixed depth, so all mate sequences will be found in that range, if they exist. If the searching depth is increased gradually from 1-6, the mates in minimal number of moves will be found. Essentially, the first found mate sequence in that case is the minimal solution of the problem.

7. CONCLUDING REMARKS

The novelty presented in this paper is the author's conception of *the zero evaluation function*, presented in Section 4.3. The new idea is to use zero function instead of classical material/positional evaluator commonly used in different chess engines. In combination with efficient searcher techniques coded in assembly language, the high performance mate solver searcher is constructed.

The *Geniss Mate Solver v2.0* is the last successor of the G.M.S. v1.0 author's mate solver application. The application is full-width, Shannon type-A problem searcher completely coded in machine language. The approach and techniques used for the realization are designed efficiently to solve all kinds of chess problems.

Application has basic intention to support human chess problemists and composers as an automatic chess problem-solving tool. In that purpose, G.M.S. v2.0 is equipped with human-friendly interface including all standard options for position editing, searching and report analyzing. Program has not special hardware requests but it is recommended minimum 800Mhz Pentium II/Celeron PC platform to achieve appropriate program strength. Memory and disk capacity and speed are not crucial because algorithm does not support hash tables.

The latest version of G.M.S. is embedded into the author's *Geniss Axon XP* application designed for classic chess. The idea is to use G.M.S. abilities to find mate combinations in minimal number of moves when main searcher has already detected mate-in-N-move sequence. The other possible research directions will be implementation of some other type of chess problems like helpmates and self-mates.

Acknowledgements. The author wishes to thank the anonymous referee for the very useful suggestions that improved the quality of this paper.

REFERENCES

[1] Althöfer, I., "An additive evaluation function in chess", *ICCA Journal*, 14 (3) (1991) 137-141.
[2] Althöfer, I., "On telescoping linear evaluation functions", *ICCA Journal*, 16 (2) (1993) 91-94.
[3] Beal, D.F., "Mating sequences in the quiescence search", *ICCA Journal*, 7 (3) (1984) 133-137.
[4] Beal, D.F., and Botvinnik, M.M., "Computers in chess - solving inexact search problems", *ICCA Journal*, 9 (2) (1986) 88-89.
[5] Bruin, A., Pijls, W., and Plaat, A., "Solution trees as a basis for game-tree search", *ICCA Journal*, 17 (4) (1994) 207-219.
[6] *Chess Informant*: *Anthology of Chess Problems*, http://www.sahovski.co.yu/acp.htm., 2000.
[7] Donninger, C., "Null move and Deep Search: selective-search heuristics for obtuse chess programs", *ICCA Journal*, 16 (3) (1993) 137-143.

[8] Fray, P.W., *Chess Skill in Man and Machine, Texts and Monographs in Computer Science*, Springer-Verlag, New York, U.S.A, 1978.

[9] Grand, H., "The impact of computers on chess-problem composition", *ICCA Journal*, 9 (3) (1986) 152-153.

[10] Kaindl, H., Horacek, H., and Wagner, M., "Selective search versus brute force", *ICCA Journal*, 9 (3) (1986) 140-145.

[11] Lindner, L., "Experience with the second human-computer problem test", *ICCA Journal*, 6 (3) (1983) 10-15.

[12] Lindner, L., "A test to compare human and computer fairy-chess problem solving", *ICCA Journal*, 8 (3) (1985) 150-155.

[13] Shannon, C.E., "Programming a computer for playing chess", *National IRE Convention*, March 9, New York, U.S.A, 1949.

[14] Shannon, C.E., "A chess-playing machine", reprinted with permission, *Scientific American*, U.S.A., 1950.

[15] Schlosser, M., "Computers and chess-problem composition", *ICCA Journal*, 11 (4) (1988) 151-155.

[16] Wiereyn, P.H., "Inventive problem solving", *ICCA Journal*, 8 (4) (1985) 230-234.

APPENDIX

The nucleus of the searcher in Pascal with assembly extension directives is presented in sequel:

```
procedure searcher;      {Performing full-width Shannon type-A chess tree search}
label l1,l2,l3,lup,lend;
var move,alter:word;
begin
    depth:=0; analized_moves:=0; stack[0].alter:=0;   { Reset basic variables…}
L1:  GENERATOR;   { <- Move generator. Move weights are automatically generated
                     by the same routine. }
       if stack[depth].full=0 then
          begin
            stack[depth].minmax:=CUTOFF;   {If base nod, CUTOFF constant is
                                              moved to minmax variable}
            goto L3;
          end;
 L2:  alter:=stack[depth].alter;   { ALTER is index of current move }
      move:=stack[depth].moves[alter].move;    { MOVE is current move }
      combination[depth]:=move;  { Combination (principal variation) are stored. }
    asm
      mov dx,move
      call MOVE_FORWARD;      { Assembly routine for playing one moves forward }
    end;
    inc(depth); fluid:=fluid+next_raw;   { Increase depth }
    if depth<max_depth then goto L1;  { If depth overcome maximum depth, go to
evaluation }
    EVALUATOR;   { Terminal evaluation is performing. }
LUP:     if depth=0 then goto LEND;  {If evaluation is performed on base node, exit !}
L3:  MOVE_BACK;                       {Assembly routine for playing back one move.}
     dec(depth); fluid:=fluid-next_raw;    {Decrease depth, go upwards the game tree }
    alter:=stack[depth].alter;
    stack[depth].moves[alter].evaluation:=stack[depth+1].minmax;
    inc(stack[depth].alter);
    if stack[depth].alter>=stack[depth].full then goto LUP;  {If all moves are processed
                                              at this node, go upwards the game tree}
    goto L2;
LEND:   { End of program. }
end;
```

Figure 8: Geniss Mate Solver v2.0 – Shannon type-A searcher nucleus