# Organizare jucator virtual

Numarul total de jocuri de sah posibile este de 10^44 care este un numar mai mare decat numarul particulelor din univers (probabil).

*A ply* is a word for a single move by one of sides. In graph terminology, it

corresponds to one edge in the game tree. Therefore one chess move (one of white and one

of black) consists of two plies.

## Number of moves

Number of edges from the root to nodes depth 1 in the initial position is 20 (there

are twenty legal white’s moves initially), and from each node depth 1 to node depth 2 is also 20 (there are also twenty legal black’s responses). However, in the chess middlegame

it is assumed that on average 35 legal moves exist for each side.

If we consider a tree of depth D, and branching factor B, then number of all its

nodes N (except root) can be calculated with formula:

N=B^D.

Length of an average chess game can be considered to be about 40 moves. 40

moves mean 80 plies. As the branching factor (number of edges from each node down the

tree) is 35 on average, we can estimate the number of nodes in a tree corresponding to a

game of such length to be N = 35^80.

## General approach

You do not explore the entire tree, because it is impossible. The general approach to writing chess

programs is to explore only a sub-tree of the game tree.

The sub-tree is created by pruning everything below certain depth. Instead of

assigning to the root a value from a set {win, draw, lose} (that could be assigned if the

whole game tree was explored down to the leaves) there is a value assigned according to

some heuristic evaluation function.

Value of the function is found at some intermediate

nodes (usually closer to the root than to the leaves) of the total game tree. These nodes are

leaves of the considered sub-tree.

As empirical tests show [Schaeffer et al., 1996, p. 1], a chess program searching

one move deeper into the game tree than its opponent (other factors being equal) wins

about 80% games played.

It means that playing strength increases almost

linearly with search depth, but after exceeding some threshold (in case of draughts – 11

plies) gain diminishes, and is negligible for greater depths (19 in case of draughts). Similar

behaviour has been observed in chess [Schaeffer et al., 1996, p. 12] – difference in playing

strength between the same programs searching 3 and 4 plies is much more apparent than

that between programs searching 8 and 9 plies. However, this phenomenon does not

manifest in current chess programs as even the best programs currently available have not

yet reached the threshold, beyond which the benefits of additional search depth are

negligible.

### Two basic approaches: forward and backwar prunning

These two strategies were first described by [Shannon, 1950] in his original work.

He called them *strategy type A* and *type B.*

Strategy type A (backword pruning), is used in most chess-playing programs (and in many other

computer implementations of games) since 60’s. Original Shannon’s work considered

basically browsing through the tree for a certain depth, calculating the evaluation function

at leaf nodes of the sub-tree. The best move (promising highest evaluation) can be

determined in that way. Since computers at that time were extremely slow, therefore

browsing through the sub-tree of depth only as low as 4 (predicting for 4 plies ahead – it

means about 35^4 = 1.5 million nodes) was possible in reasonable time.

It was a discovery of an a-b algorithm (covered later) that allowed more efficient analysis of the tree by

means of so called *backward pruning.* This technique allowed relatively efficient computer

implementation of strategy type A programs. The basic idea of the a-b algorithm is to skip

branches that cannot influence the choice of the “best move” using data already gathered.

Another approach – strategy type B (*forward pruning)* – tries to use

expert knowledge to prune moves that seem poor and analyse only a few promising

branches (just as human players do). Programs following this approach were written before

a-b dominated the arena. There were attempts using this approach later as well, but they

did not fulfil expectations – analysis performed to prune branches *a priori* was too

computationally extensive.

## Search vs Knowledge

There is quite a lot of expert knowledge concerning chess around. For example,

**openings** (the first stage of a chess game) have been studied and there is a lot of literature

available.

**Opening databases** are present in all good chess programs.

Once the program cannot follow the opening book anymore, it switches to its

normal mode (strategy type A).

By feeding the function with more and more factors one

must unavoidably reduce speed of program operation – computation of evaluation function

value must take some time.

One more example of using knowledge to minimize need for tedious search is

usage of **endgame tablebases**. When there are not so many pieces left on the board, it

turns out to be possible to precompute the optimal line of play and put it in a database.

When the program reaches a position that exists in the endgame base, it can switch from

search mode to tablebase-read mode, with tablebase serving as an oracle. It saves time.

## Typical structure of chess program

Therefore, each chess program must have a data structure to hold the **position** and

**moves.** Moreover, it must contain some kind of a **move generator**. Finally, it must

have some kind of a **search algorithm** and an **evaluation function**.

## Data structures

### Chess move

Definitely, a structure describing a chess move should contain the source and

destination square. Information contained in the *move* structure must be enough for the

program to be able to take that move back. Therefore, it should also describe the captured

piece (if any). If the move is a promotion, it must contain the piece that the pawn is

promoted to. Sometimes (it depends on a type of a chessboard representation), also type of

the piece that is moving should be included.

The above data may be put together into a structure having several one byte fields

(e.g.: *source, destination, captured, promoted, moved*). However, the efficiency of the

program often increases if the structure is contained within one 32-bit integer (for example,

6 bits per source / destination field, 4 bits per captured / promoted / moved piece) as it

reduces the memory access overhead for transferring/accessing particular fields of

instances of this often used structure (assuming that arithmetic and logical operations are

performed relatively much faster than memory access – in most hardware environments,

including the most popular PC platform, it is exactly the case).

### Game history entry

At the moment the move is made, data contained in the *move* structure is enough.

The procedure that makes moves has enough information to update locations of pieces as well as castling rights, *en passant* status and number of moves before 50-moves

rule applies. However, more information is needed to undo the move. For example, it is

often impossible to recover the castling rights using only information from the *move*

structure. Therefore, the game history entry should contain all the information needed to

take a move back. It includes: data from the *move* structure (except *promoted* field),

castling rights before the move was made, *en passant* capture field (if any) and number of

moves without capture and pawn movement.

### Representing chess position

The most natural way seems to be an array 8x8, with each element corresponding

to one of chessboard squares.

Later, the idea was improved by adding two square sentinels at the edges. Sentinel

squares were marked *illegal*. It speeded up move generation, as no check had to be done

each time a move was to be generated to verify if edge of the board was not reached.



Another possible representation is 0x88. It is friendly to bishop like moves and is suitable for environments where cache memory is not present, since it requires less memory access.

### Bitboards

Idea of using bitboards (64-bit integers) came into being when 64-bit mainframes

became available in 1960’s. Sixty four happens to be the number of squares on the

chessboard. Therefore, each bit in the 64-bit variable may contain a binary information for

one square. Such binary information might be, for example, whether there is a white rook

on the given square or not. Following this approach, there are 10 bitboards (+ one variable for each king) necessary to completely describe location of all pieces (5 bitboards for white

pieces (pawns, knights, bishops, rooks, queens) and 5 for black).

There are 10 bitboards to represent a chess position:

White pawns, white knights, white bishops, white rooks, white queens

Black pawn, black knights, black bishops, black rooks, black queens

Each bitboard is like: It is a 64 bit integer.

Therefore, each bit in the 64-bit variable may contain a binary information for

one square. Such binary information might be, for example, whether there is a white rook

on the given square or not.

For example, for white pawns bitboard, the 64 number bitboard contains a value of ‘1’ where there is a pawn on the chess table.

There is no need to maintain bitboards for kings as there is always only one king on the board – therefore a simple one byte variable is enough to indicate location of the king.

An addition to the bitboard representation of the actual chess position is a database

containing bitboards representing square attacked by a given piece located on a given

square.

For example, *knight*[A4] entry would contain a bitboard with bits corresponding to squares

attacked by a knight from A4 square set.



**Fig. 5. Graphical representation of bitboards representing a knight (or any other piece)**

**standing on square A4 (left) and squares attacked by this knight (right)**

This idea with additional bitboards can be used to generate possible moves for each piece.

Using the bitboard and the precomputed database, it is very fast to generate moves

and perform many operations using processor’s bitwise operations.

For example, verifying

whether a white queen is checking black’s king looks as follows (an example taken from

Laramée 2000):

1. Load the "white queen position" bitboard.

2. Use it to index the database of bitboards representing squares attacked by

queens.

3. Logical-AND that bitboard with the one representing "black king position".

If the result is non-zero, then white queen is checking black’s king. Similar analysis

performed when a traditional chessboard representation (64 or 144 byte) was used would

require finding the white queen location (by means of a linear search throughout the array)

and testing squares in all eight directions until black king is reached or we run out of legal

moves. Using bitboard representation, only several processor clock cycles are required to

find the result.

## Implementarea procedurilor

### Move generation

3 approaches:

1. **Selective generation** – carefully analyse the position in order to find a few

promising moves, and discard all the rest.

1. **Incremental generation** – generate a few moves at a time, hoping that one of

them turns out to be good enough for the rest to become insignificant – they

would not need being generated, leading to time savings. The savings may be

significant, since move generation in chess is not trivial (there are castlings, en

passant captures, each kind of piece moves differently).

1. **Complete generation** – generate all possible moves for the given position in one

batch.

Incremental and complete generation are commonly used in modern chess

programs. There are benefits and penalties from both schemes.

Search routines basing on alpha-beta algorithm for backward pruning that are used

in chess programs are very sensitive to move ordering – the rule here is simple: best move

first. It means that best moves should be analysed first, and in that case most moves may

never need being analysed (and therefore even generated). Obviously, the program never

knows whether the generated move is best in the given position – if it knew, it would not

need analysing it anymore.

However, there are some rules that allow us to indicate best (or

at least good) moves – for example, in the game of chess a best move is often a capture, or

a checking move.

Therefore, since we can achieve a reasonably good move ordering, it would seem

more logical to employ an incremental generator – in the first batch it might return only

captures (preferably sorted by an expected material gain), then checking moves etc. In such

case, often one of the moves generated in the first batch would turn out to be the best (or at

least good enough) one, and the rest would not have to be generated. The drawback of the

incremental scheme is that it takes it longer to generate all possible moves than it would

take a generator that returns all possible moves at once.

A complete generation is the fastest way to obtain all moves for a position.

However, usually most of those moves (assuming reasonable move ordering) turn out to be

useless for the engine – they never get to being analysed. An advantage of generating all

possible moves at once is that it enables some search tricks (e.g. enhanced transposition

cutoff (ETC) – one of the search enhancements described later), that may give benefits

outweighing the cost of excessive generation.

Obviously, even when using complete generation a good move ordering remains

crucial – it may lead to many branches being skipped by the search routine. Therefore, move generator should return moves according to the “first-best” rule wherever possible,

reducing the need for reordering later.

### Improvement of the evaluation function

Currently, some programs use other techniques to improve work of the evaluation

function. One approach is given by George and Schaeffer (1990). The approach aimed to

add some “experience” to the program. As most chess programs do not remember past

games, they lack a factor that is natural for humans – possibility of using past experience in

new competitions. The approach shown in that publication is to supply the program with a

database of positions taken from thousands of grandmaster games. During play, the

database is queried for identical or similar positions. The database is indexed with some

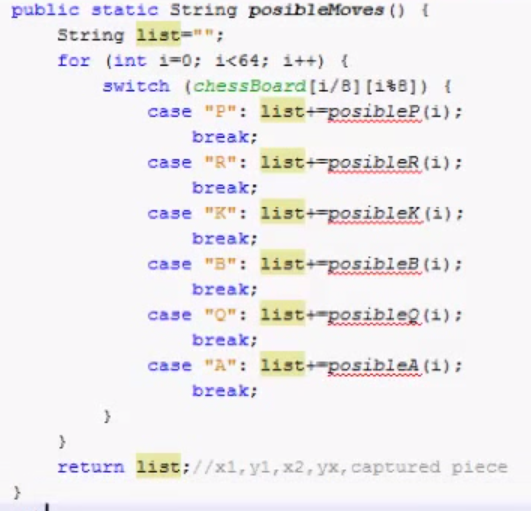
patterns. The program extracts some patterns from the current position, queries the

database, and is given back some information about the position. This information is used

to modify value of the evaluation function, therefore increasing probability of choosing the

move that seems proper according to the database.

## Other ideas

* Move generation
  + - Abordare: minmax with apha-beta pruning
    - The evaluation function should be calculated only for the leaves. The reason is the following: suppose that on the next move, the opponent takes my queen, and this gives me the opportunity to give him check mate in the following move.
  + Verificarea daca o mutare e valida
  + 
  + Evaluation function
    - Avantajul material (numarul de piese in avantaj)
    - Avantajul pozitional
    - Situatiaa in care da sah si poate lua o piesa (gen da sah cu calul si cand oponentul muta regele, jucatorul virtual sa ia o alta piesa)
  + Cautarea urmatoarei mutari
  + Make move
  + Undo move