The Hoffman Reference Guide

Brent Baccala

September 13, 2018

Hoffman is a program to solve chess endgames using retrograde analysis, which is much different from conventional computer chess programs. Retrograde analysis is only useful in the endgame, runs very slowly, and produces enormous amounts of data. Its great advantage lies in its ability to completely solve the endgame. In a very real sense, a retrograde engine has no "move horizon" like a conventional chess engine. It sees everything. For those not up on Americana, the program is named after Trevor Hoffman, an All Star baseball pitcher who specializes in "closing" games. It was written specifically for The World vs. Arno Nickel game.

Hoffman uses XML extensively both for configuring its operation and for labeling the resulting tablebases. In fact, a completed Hoffman tablebase (typically in an `.htb` file) is just a `gzip`-ed file that contains an XML prefix followed by binary tablebase data (in a format specified by the XML). Basically, to operate Hoffman, you write an XML file that specifies the analysis you want done, then feed it to the program. As output, it produces a modified version of the XML input that includes binary tablebase data appended at the end.

1 Parallel Processing with Hoffman

A Hoffman analysis can be quite compute-intensive. The program can be compiled to use POSIX threads (if available), with the number of threads specified at run-time using the `-t` option. The program is also designed to use multiple computers in parallel, all working simultaneously on an analysis. This is accomplished by breaking the analysis up into smaller pieces, each with its own XML configuration file.

2 Propagation tables

A Hoffman analysis can also be quite space-intensive. Since its memory utilization pattern is basically random, Hoffman will begin to swap dramatically and suffer a disastrous drop in performance once its working set size exceeds the machine's available memory. To alleviate this, the program can be operated in a mode where it fills a series of *propagation tables*, writing each one out to disk when full, then reads them back in sequentially during the next pass. Although less efficient than when the working set can be contained in memory, propagation tables allow the program to build tablebases of essentially unlimited size with no swapping and reasonable CPU utilization. This mode is activated at run-time by specifying the size of the propagation tables (in MB) with the `-P` switch. Temporary files will be written to the current directory.

For example, the command `hoffman -g -t 2 -P 1024 kqqkqq.xml` will trigger a Hoffman generation run with two threads, using one gigabyte (1024 megabytes) of memory.

3 XML Syntax

The root XML element in a Hoffman tablebase is always `<tablebase>`. Its only attribute (`offset`) is added by the program, should not be supplied by the user, and indicates a hexadecimal byte-offset into the file where the binary tablebase data begins.

Within a `<tablebase>` the following elements may occur in the listed order (deprecated elements and attributes are not documented):

3.1 `<prune-enable color="white|black" type="concede|discard"/>`

Specifies which kinds of pruning elements will be allowed in this tablebase and its futurebases. Both attributes are required. `concede` means wins may be conceded to the named color; `discard` means moves by the named color may be discarded. At most one `prune-enable` can be specified for each color. No `prune-enable` element is required, however, no `prune` elements are allowed without one and no futurebases may possess additional `prune-enable` elements beyond those specified for the current tablebase.

3.2 `<variant name="normal|suicide"/>`

The optional `<variant>` element specifies which version of the rules of chess apply to this tablebase.

Default: `normal`

3.3 `<index type="naive|naive2|simple|compact|no-en-passant|combinadic|combinadic2|combinadic3|combinadic4|pawngen" symmetry="1|2|4|8"/>`

The `<index>` element specifies the algorithm that will be used to compute the index numbers in the tablebase; i.e, the algorithm that will convert board positions into tablebase offsets and vice versa. It typically is not specified by the user (but can be).

`naive` uses 2^{6n+1} indices to store positions for n pieces. It assigns a single bit for the side-to-move flag, then assigns 6 bits to each piece, which is used to encode a number from 0 to 63, indicating the piece's position on the board.

`naive2` Differs from `naive` in its handling of multiple identical pieces, which it stores as a base and an offset, thus saving a single bit. Currently, only pairs of identical pieces are handled; a fatal error will result if there are more than two identical pieces.

`simple` Like `naive`, but only assigns numbers to squares that are legal for a particular piece. Slower to compute than `naive`, but more compact for tablebases with lots of movement restrictions on the pieces.

`compact` A combination of the delta encoding used for identical pieces in `naive2`, the encoding of restricted pieces used in `simple`, plus a paired encoding of the kings so they can never be adjacent.

`no-en-passant` An enhancement of `compact` that uses the paired encoding scheme for pawns restricted to the same file. Since they can never pass each other, we can encode them as if they were an identical pair, then assign their colors in the same order they were originally specified. En passant significantly complicates this and can not be handled with this scheme.

`combinadic` An enhancement of `naive2` that can encode more than two identical, overlapping pieces by using a combinadic encoding scheme (see the wikipedia page "Combinatorial number system").

`combinadic2` Like `combinadic`, but later pieces wholly contained within the semilegal range of earlier pieces are encoded using fewer positions. Piece order is significant. No attempt is made to reduce the encoding of pawns.

`combinadic3` Like `combinadic2`, but pawn encodings are also reduced, by reducing its encoding value (with en-passant factored in), while using its board position to reduce other pieces..

`combinadic4` Like `combinadic3`, but *color symmetric* tablebases, those invariant under swapping colors, are optimised by removing the side-to-move flag, cutting the tablebase size in half. (ex: `kqkq` is color symmetric, but `kqkr` is not)

`pawngen` Like `combinadic4`, but pawns are handled separately by building a table of all possible pawn positions that can result from a given initial pawn configuration.

The optional `symmetry` attribute can be used to encode multiple positions using a single entry, but its utility depends upon the exact analysis being done. A tablebase with no pawns and no movement restrictions can be encoded with 8-way symmetry, since the board can be rotated about a horizontal, vertical, or diagonal axis without affecting the behavior of the pieces. A tablebase with pawns can utilize at most 2-way symmetry, since only a reflection about a vertical axis preserves piece behavior. A tablebase with restrictions on the positions of the pieces (say, frozen pawns) can not use any symmetry at all. Not all symmetries are compatible will all index types; for example, 8-way symmetry can not be used with `naive` or `naive2` index types.

Default: `combinadic4` with automatically selected symmetry, unless a `pawngen` element is present in the XML, which triggers `pawngen`

3.4 Tablebase format

The next three elements specify the format of the tablebase entries. At most one of them can be specified.

`<dtm bits=`*integer*`/>` specifies a *distance to mate* metric. Zero is used for draws, -1 is used for positions where the moving side is checkmated, and 1 is used for positions where the moving side can capture the opposing king, so an eight bit dtm field can record mate-in distances up to 126. If a field size is not specified, it is selected automatically.

`<dtc bits=`*integer*`/>` specifies a *distance to conversion* metric, which is the number of moves required before reaching a different tablebase. Zero is used for draws, -1 is used for positions where the moving side is checkmated, and 1 is used for positions where the moving side can capture the opposing king, so an eight bit dtm field can record distances up to 126. If a field size is not specified, it is selected automatically.

`<basic/>` specifies a *bitbase* where two bits are used for each position, and no distance information is stored — only an indication of the ultimate outcome (win, lose, or draw). Such a format is more compact and requires less time to generate, but requires more effort to use, since care must be taken to avoid loops when following winning lines.

`<flag type="white-wins|white-draws"/>` specifies a *bitbase* where only a single bit is used for each position. `white-draws` includes both winning and drawing positions for white, so it is essentially `NOT black-wins`.

Default: DTM with automatically selected field size.

3.5 <piece color="white|black" type="king|queen|rook|bishop|knight|pawn" location="*string*"/>

Multiple `piece` elements are used to specify the chess pieces present in the tablebase. `color` and `type` are required and should be obvious. The ordering of `piece` elements is significant in that it directly affects the index algorithm, but there is no user-visible effect of the ordering.

The optional `location` attribute restricts the board positions available to this piece. It should be a list of squares, in algebraic notation, on which the piece is to be allowed. A single square results in a completely frozen piece. In addition, pawns may use an additional syntax consisting of a single starting square followed by a plus sign, indicating that the pawn may move forward as far as possible. This can be used, for example, to locate a black pawn on `"a7+` and a white pawn on `"a2+"`, indicating that both can move forward, but they can not "pass" each other.

3.6 <pawngen white-pawn-locations="*string*" black-pawn-locations="*string*" white-pawns-required="*number*" black-pawns-required="*number*" white-queens-required="*number*" black-queens-required="*number*" white-captures-allowed="*number*" black-captures-allowed="*number*"/>

Pawn configurations can be specified using a `pawngen` element instead of `piece` elements. A set of starting pawn locations is specified for each color using `*-pawn-locations` attributes, and all possible pawn moves from that initial configuration are calculated. Since the number and types of pieces is fixed for each run of the program, the `*-pawns-required` attributes must be specified to indicate how many pawns of each color are allowed. Optionally, the `*-queens-required` attributes can specified to force consideration of positions where a certain number of pawns have queened, and the `*-captures-allowed` attributes include consideration of positions where a certain number of non-pawn pieces have been captured (pawn captures are already considered).

The auxiliary Perl script `pawngen` can accept control files without `*-pawns-required` attributes, and generates new, interlinked control files with the required attributes added.

3.7 <futurebase filename="*string*" colors="invert"/>

One or more futurebases may be specified with this element. A `filename` must be specified to locate a futurebase, which must be another tablebase, in either Hoffman, Nalimov, or Syzygy format. The path to a Nalimov tablebase is ignored; it must be in the directory specified on the command line using the `-N` option.

The futurebase must be related to the current tablebase in one of the following ways:

It has exactly the same piece configuration as the current tablebase, and corresponds to movement by one of the restricted pieces, i.e, the current tablebase has a white pawn frozen on `e4` and the futurebase has a white pawn frozen on `e5`.

It has exactly the same piece configuration as the current tablebase except that a single piece is missing, i.e, a capture occurred.

It has exactly the same piece configuration as the current tablebase except that a single pawn has been replaced with a knight, bishop, rook or queen, i.e, a pawn promoted.

It has exactly the same piece configuration as the current tablebase except that a single pawn has been replaced with a knight, bishop, rook or queen, and a single non-pawn of the opposite color has been removed, i.e, a pawn captured and promoted in the same move.

The option colors="invert" attribute may be specified to indicate that the piece colors of the futurebase should be inverted as it is processed. This precludes the need to calculate, say, a tablebase with a white queen and a black rook as well as a tablebase with a black queen and a white rook. The first may be used (with this option) as a futurebase to calculate a tablebase with two white rooks and a black queen.

Note: Any futurebase prune-enable elements must be a subset of the current tablebase's prune-enable elements.

3.8 <prune color="white|black" move="*string*" type="concede|discard"/>

Futuremoves not handled by specifying futurebases must be pruned using one or more of these elements, or an error will result. The move is specified using regular expression syntax to match a move in a subset of standard algebraic notation. All of the following strings are examples of legal move strings in a prune element: Pe5, P=Q, RxQ, PxR=Q. The following regular expressions would all match Kd4: Kd?, K?4, K[a-d]4, K*. The type attribute specifies what should be done with matching moves: treated as wins for the moving side (concede), or completely ignored (discard). If multiple prune elements match a particular move, it is a warning if they have the same type, a fatal error if their types differ.

A single prune element may be specified with move="stalemate" and type="concede". In this case, the color attribute indicates to which side stalemates should be conceded as wins.

Note: prune elements do not affect moves within a tablebase. Specifying a prune element that only matches moves within a tablebase will do *nothing*.

Note: If a prune element is specified for a futuremove handled by a futurebase, then the futurebase takes precedence. However, this case is handled by tracking every futuremove in every position, so it is possible to specify futurebases that handle a subset of the possible futuremoves, then use prune elements to handle the rest by default.

Note: prune elements are only allowed if they match a prune-enable element. If no prune-enable elements were specified, then no prune elements will be permitted.

Note: Earlier versions of Hoffman allowed a pawngen-condition attribute that is no longer supported.

3.9 Generation Controls

These elements are all optional, but if output is not specified, an output filename must be specified on the command line using the -o switch.

3.9.1 <output filename="*string*"/>

At most a single output element should be used to specify where the finished tablebase should be written.

3.10 <tablebase-statistics> ... </tablebase-statistics>

This element is added by the program and should not be specified in the input. It contains statistics relating to the finished tablebase.

Element	Interpretation
indices	Total number of entries in the uncompressed tablebase
PNTM-mated-positions	Total number of positions in which *player not-to-move* is mated; i.e, illegal positions in which a kind can be immediately captured
legal-positions	Total number of legal positions; i.e, total number of entries, minus illegal entries where two pieces occupy the same space, minus PNTM-mated positions
stalemate-positions	Stalemate (not draw by repetition) positions
white-wins-positions	Positions from which White can force a win
black-wins-positions	Positions from which Black can force a win
forward-moves	Total number of forward moves from positions in this tablebase (including futuremoves)
futuremoves	Total number of forward moves from positions in this tablebase into futurebases or pruned
max-dtm	Largest *distance to mate* of all positions in this tablebase
min-dtm	Smallest *distance to mate* of all positions in this tablebase, i.e, a negative number indicating the longest forced loss

3.11 <generation-statistics> ... </generation-statistics>

This element is added by the program and should not be specified in the input. It contains statistics relating to the program run that generated the tablebase.

Element	Interpretation
host	Hostname of system that generated the tablebase
program	Name and version of the program that generated the tablebase
args	Command line used for the generation run
start-time	Time the program run initially started
completion-time	Time the program run finally ended
user-time	CPU time used by the run in user space
system-time	CPU time used by the run in system calls
real-time	Wall clock time used by the run
page-faults	Number of times the program had to wait for a memory page to be swapped in from disk
page-reclaims	Number of times the program reclaimed a page from the free list; this will typically be program instruction pages
proptable-writes	If proptables are in use, the number of proptables written to disk
proptable-write-time	If proptables are is use, the total real time required for all proptable writes
pass	Per-pass statistics, including real-time and user-time

4 Some Confusing Error Messages

4.1 Futurebases can't be less symmetric than the tablebase under construction

Symmetric tablebases collapse multiple positions into one, so futurebases must have the same symmetry (at least), or the futurebase might handle differently two positions that the more symmetric tablebase treats as one.

4.2 Doubled pawns must (currently) appear in board order in piece list

Currently, doubled pawns using "plus" locations (ex: `location="a2+"`) on the same file must have their `piece` elements listed in the XML in the order that the pawns appear on the board, counting in algebraic notation from row 1 to row 8. I mean, row 2 to row 7.

4.3 Piece restrictions not allowed with symmetric indices (yet)

You can't specify an `index symmetry` attribute and also specify `piece location` attributes, even if the restrictions on the piece locations might be compatible with the requested symmetry.

4.4 Non-identical overlapping piece restrictions not allowed with this index type

For the `naive`, `naive2`, and `simple` index types, you can't specify two identical pieces with different `location` restrictions unless those restrictions are completely distinct. For example, you can't have a free white rook and another white rook restricted to the a-file. If you think about it, this situation would allow the rooks to "trade places" — both could move to the a-file and then either one could move off. The simpler index types can't handle this situation. You could, however, have a white rook restricted to the a-file and another restricted to the d-file (or use a more sophisticated index type, like `compact`).

4.5 Futurebase doesn't match prune-enables!

Remember that futurebase `prune-enable` elements must be a subset of the current tablebase's `prune-enables`.

4.6 No futurebase or pruning for ... Futuremoves not handled ...

If one or more futuremoves are not handled by specifying either a futurebase or a prune statement, then a fatal error will result either immediately or after the initialization pass. To aid in diagnosis, the error message includes the FEN of the offending position.

4.7 pawngen doesn't support output elements in generation-controls

The `pawngen` script automatically generates all of its filenames. Remove the output element. After generation, all of the resulting `htb` files can be loaded together into Hoffman's probe mode.