Gamescrafters Documentation
GENERIC HASH API

Max Delgadillo

Version History

2006.10.02 - Version 1.0 - First version created.
2006.10.17 - Version 1.1 - Changed from .doc to .tex, added Custom Contexts Mode explanation.

Contents

1 Overview

For new module writers as well as veteran ones, defining a hash that encompasses the entire game can be a very daunting task. Not only must the hash be completely free of collisions - that is, there is one and only one game state (or simply a "board") for every `POSITION` - it must also be packed in as tight as possible so that there are few illegal `POSITION`s. And even when that is done, the number of positions in that hash- the game's `gNumberOfPositions` - must be calculated and set, which is not always easy.

Luckily, Gamesman provides an interface to create a board hash for the module automatically! By providing a few key arguments, such as the boardsize and the type and amount of pieces on the board, a "generic hash" will be created which can take in `POSITION`s and return the appropriate board (given by a `char*`) and player's turn on that board. This hash is guaranteed to be perfectly packed (such that all unique permutations of the pieces are possible, and there's no gaps in between them) and have no collisions. Plus, it returns the hash's total positions, thus automatically giving you the number with which to set `gNumberOfPositions`!

The Generic Hash obviously doesn't work for all games (such as games which require more information than simply a board and the player's turn) but can be adjusted to at least do part of the hashing (for example, you can have Generic Hash hash just the board and player's turn, and then you tack on additional multipliers for any other attributes you must keep track of).

All in all, Generic Hash is a great asset both to module coders who need a hash for their entire game, and more complicated games (and all Tier-Gamesman games!) which require more than one hash to be initialized and switched between as the game progresses.

2 Goals and Non-Goals

This document will explain how to use `generic_hash`'s different functions, as well as provide basic examples. After reading this document, readers should have a clear grasp of how to go about setting up a global hash for their game, and how to call its methods to attain its information.

That said, this document will not go into any other aspects of programming a module; it is assumed that the reader already knows the basics of the process to add a module to Gamesman as well as knowing which API functions (such as `Primitive()`, `GenerateMoves()`, etc.) require these functions. Of course, there is plenty of code in the existing modules which give very good examples of the different ways you can use the `generic_hash` code in coding a module.

3 Functionality Overview

3.1 The Basics

- Call `generic_hash_init()` to initialize the hash.
- Use its function pointer argument to help only hash legal boards.
- Call `generic_hash_init()` again to initialize new contexts.
- For the current context (the active hash):
  - Unhash boards with `generic_hash_unhash()` (and player's turn with `generic_hash_turn()`). The boards will be a char*, of length specified in `generic_hash_init()`, and player's turn will be either 1 or 2.

  - Hash boards with `generic_hash_hash()`.
  - Get the maximum number of boards (the `gNumberOfPositions`) with `generic_hash_max_pos()`.
  - Change the current hash's custom context number with `generic_hash_set_context()`.

- Switch between using custom context numbering or standard with `generic_hash_custom_contexts_mode()`.
- Find out what hash context is active with `generic_hash_cur_context()`.
- Use `generic_hash_context_switch()` to change between the different hash contexts.
- Finally, clear all the contexts and free the memory used by the hash(es) with `generic_hash_destroy()`.

3.2 For Those Games With Only One Hash

Don't worry about contexts - if you're only using one hash, they don't help you.

Simply pretend that a subsequent call to `generic_hash_init()` throws away the current hash and makes another one, so that all of the above functions refer to just one global hash. Or, probably better, always call `generic_hash_destroy()` before calling `generic_hash_init()` again, to ensure that there really is only one hash.

Ignore usage of `generic_hash_cur_context()`, `generic_hash_context_switch()`, `generic_hash_custom_contexts_mode()`, and `generic_hash_set_context()`.

3.3 For Those Games With Many Hashes (And Tier-Gamesman Games)

Here's an in-depth look at contexts:

If never been called before (or called after `generic_hash_destroy()`), `generic_hash_init()` will initialize a brand new hash with the second set of given parameters.

If called again, this will create a new HASH CONTEXT, so that both hashes will exist simultaneously, though only one (in this case, the newly created one) will be active at any time. The new hash will be context 1, and the older will be context 0. Every subsequent call of `generic_hash_init()` will create a new context and automatically set it as the active context, each time receiving an incremented context number.

Note that specific contexts can't be deleted; only by calling `generic_hash_destroy()` will you remove all the current contexts.

Every other function listed (except for, obviously, `generic_hash_destroy()`) depends on what the current context is. Hashing, unhashing, max positions, etc. is all in terms of the current context, which can obviously be very different for different contexts.

Thus, make use of `generic_hash_context_switch()` to always switch to the right context before using any other functions (it's very low overhead). You can also use `generic_hash_cur_context()` to check what the current context is.

Finally, keep in mind that the contexts are numbered in the order in which you initialize them by calling `generic_hash_init()`. Thus, a good initialization order can make context bookkeeping/switching a lot easier. If, however, using this system gets too messy/complicated to keep track of, there is a way to refer to the hash contexts via your own numbering system by using "Custom Context" numbering:

3.3.1 About Custom Context Numbering

To help Tier-Gamesman module writers with non-straightforward tier numbering, I have introduced Custom Context Mode into `generic_hash`. Essentially, it allows a user to define his or her own context numbers for every hash context. Since the mode is FALSE by default and is only switched on by the user, the majority of users won't need to worry about this mode. Nevertheless, if you are interested, read on:

Think of every hash context as having *two* context numbers: a normal context number, which it receives upon creation and is equal to the number of times `generic_hash_init()` has been called before it (as seen above), and a *custom* context number set by the user. By default, a hash context's custom context number is the same as its normal context number: it's up to the user to change it.

To change the custom context number of the current hash, use `generic_hash_set_context()`. The custom context number can be any non-negative integer.

To use the custom numbers instead of the normal ones (and vice versa), turn on Custom Context Mode with `generic_hash_custom_contexts_mode()`, setting its argument to TRUE to turn it on and FALSE to turn it off. The mode will be reflected in `generic_hash_cur_context()` and `generic_hash_context_switch()`, which will be in terms of either custom numbers or normal ones, depending on the mode.

That said, there's two caveats regarding `generic_hash_context_switch()` in this new mode:

First of all, `generic_hash_context_switch()` is $O(1)$ normally, but is $O(N)$ in Custom Context Mode, where N is the number of contexts that have been initialized. Thus, having freedom of numbering contexts arbitrarily comes at a price at run-time, since `generic_hash` now has to search through all the contexts to find the right number.

Second, and possibly more important, no error checking is done to ensure that all hashes have a different custom context number. That means that two (or more) hashes could potentially have the same custom context number, either by an oversight by the programmer or simply by having a later hash start with a default custom context number that happens to be the same as an already-defined custom context number (for example, calling `generic_hash_init()` once, setting that hash's custom context to 1, and then calling `generic_hash_init()` again; now both contexts have a custom context number of 1). To avoid this, a good methodology is to always call `generic_hash_set_context()` literally *right* after the `generic_hash_init()` call, to ensure that all hash contexts receive a unique context number by the user right as they are created. If this is done in tandem with tier numbering, and it's been assured that all tier numbers are unique, then this system should work well for all Tier-Gamesman games defining all contexts in terms of the tiers they represent.

4 Generic Hash API

Here is a full list of all the functions that the Global Hash provides for you. Game modules must adhere to the function name and prototype.

4.1 `generic_hash_init()`

```
POSITION generic_hash_init(int boardsize, int pieces_array[],
int (*vcfg_function_ptr)(int* pieces)), int player)
```

This initializes a Generic Hash, and makes new contexts if called multiple times.

- `boardsize` is the total length of the game board, a positive integer value. In tic-tac-toe, for example, `boardsize` = 9.

- `pieces_array[]` is an array of the form
  $\{p_1, L_1, U_1, p_2, L_2, U_2, ..., p_n, L_n, U_n, -1\}$ where:
  - The p_i's are the characters associated with the pieces (including blanks)
  - The L_i's are the minimum allowable number of occurrences of each piece type on the board.
  - The U_i's are the maximum allowable number of occurrences of each piece type on the board.
  - The -1 is used to mark the end of the array.

  In tic-tac-toe, this array is:
  $\{'o', 0, 4, 'x', 0, 5, '-', 0, 9, -1\}$
  provided that player 'x' moves first (since then there'd be more 'x's than 'o's).

  Note that:
  - There are at least 0 'o's on the board (this can occur on a totally empty board, or when 'x' has made 1 move).
  - There are at least 0 'x's on the board (this occurs on a totally empty board).
  - There are at most 4 'o's on the board (this can occur on a board that's full or that has one blank square).
  - There are at most 5 'x's on the board (this occurs on a totally full board).
  - Blank '-' occurrences range from 0 (full board) to 9 (empty board).

  Also note that the order in which the pieces are given doesn't matter; only the values for the pieces matter. Additionally, this says nothing about who Player 1 and Player 2 are: the hash will create two copies of each possible board, one which corresponds to Player 1's turn and the other corresponding to Player 2's turn on that board. (For details, consider the discussion of the `player` argument further below.)

- `*vcfg_function_ptr` is a pointer to a function vcfg described below (meaning that you have to write the function, and pass in a pointer to it). This function takes in `int pieces[]` as an argument, where `pieces[i]` counts number of occurrences of the ith piece in this configuration. Thus, for the example `pieces_array` above, `pieces[]` would be of length 3, with `pieces[0]` = 'o', `pieces[1]` = 'x', and `pieces[2]` = '-'.

  It returns 1 (or TRUE) if this configuration is "valid", 0 (or FALSE) otherwise.

  Most users will not need to write this function, and instead set the third argument of `generic_hash_init()` to NULL.

  This feature is necessary if the user needs extra optimizations like "dartboard hash". If, for example, the user needs a dartboard hash for the above example of tic-tac-toe, here is one way to write it:

```
int vcfg(int pieces[])
{
  /* if number of 'o's is equal to or one less than number of 'x's
  then this configuration is valid*/

  return pieces[0] == pieces[1] || pieces[0] + 1 == pieces[1];
}
```

As a final note, keep in mind that `generic_hash` makes sure that the pieces perfectly fit the board every time, so you don't have to check that. That is, if `pieces[]` has size n, then:
`(pieces[0] + pieces[1] + ... + pieces[n-1] == boardsize)`
is ALWAYS true.

- `player` usage: Normally, the hash should initialize boards for two players - that is, there is essentially two of every possible board, each corresponding to a different player's turn. In this case, `player` should be 0. If, however, your hash only needs to include one player's turn (such as with certain non-loopy tier hashes), then you may pass in either a 1 or a 2 and this initializes a hash with boards only for that player.

Finally, `generic_hash_init()` returns the number of different possible boards (a.k.a the "maxpos" of the hash, or the "`gNumberOfPositions`") for the hash just initialized.

4.2 generic_hash_hash()

`POSITION generic_hash_hash(char *board, int player)`

Hashes a board and player's turn into a `POSITION`.

- `board` is any valid physical board, represented as an array. (No error-checking is done on the board, so if you enter a bad board, it messes up.) The board is not freed, so you should make sure to call `SafeFree(board)` after calling `generic_hash_hash()` if you have no further use for the board.
- `player` denotes whose turn it is (either 1 or 2).

This returns the `POSITION` value of that board and player's turn.

4.3 generic_hash_unhash()

`char *generic_hash_unhash(POSITION hash_number, char *empty_board)`

Unhashes a `POSITION` into a board. Note that the player whose turn it is on that board is given by `generic_hash_turn()` below; this simply returns the actual board.

- `hash_number` must be between 0 and maxpos (where maxpos = the number of different possible boards in the current hash context, given by `generic_hash_max_pos` as described further below).
- `empty_board` must be a pre-defined (either a local variable or `SafeMalloc()`'d) `char*` of length `boardsize`.

The contents of the board whose index is `hash_number` are stored in `empty_board`. This function also returns the board as well.

4.4 generic_hash_turn()

`int generic_hash_turn (POSITION hashed)`

Usually called in tandem with `generic_hash_unhash()`, this unhashes a `POSITION` into a player's turn on the board that would be returned by calling `generic_hash_unhash()` on that `POSITION`.

- `hashed` is the hash value of a board, with the same constraints as `hash_number` above.

This returns a 1 or a 2, corresponding to player 1 or player 2. If this hash was initialized with a non-zero `player` argument, this always returns the player that was given in the argument.

4.5 generic_hash_max_pos()

`POSITION generic_hash_max_pos()`
This returns the current hash's "maxpos" (`gNumberOfPositions`).

4.6 generic_hash_custom_context_mode()

`void generic_hash_custom_context_mode(BOOLEAN on)`
This switches the hash mode to Custom Context Mode, or turns it off.

- `on`: TRUE if setting to Custom Context Mode, FALSE if setting to normal mode.

4.7 generic_hash_set_context()

`void generic_hash_set_context(int context)`
This sets the custom context number of the current hash context.

- `context` will be set as the hash's context number. It must be non-negative, or else Gamesman will exit.

No error checking is done on `context` (apart from confirming it is non-negative), so it could be being set to an already existing context number for some other context. Care must be taken by the programmer to ensure that this isn't so.

4.8 generic_hash_cur_context()

`int generic_hash_cur_context()`
This returns the context number of the currently active hash context.
If in Custom Context Mode, this will return the custom context set by the user (by default, it is simply the normal context). Otherwise it returns the normal context, which indicates how many times `generic_hash_init` was called before this context was created.

4.9 generic_hash_context_switch()

`void generic_hash_context_switch(int context_number)`
This switch the current context to the argument context.

- `context_number` indicates which context to switch to. It must be an actual existing context number, or else Gamesman will exit.

If in Custom Context Mode, `context_number` refers to a custom context number, otherwise it must refer to a context between 0 (inclusive) and the number of times `generic_hash_init` has been called (exclusive). It should be noted that if two hash contexts have the same custom context number via an oversight of the programmer, this will switch to the OLDEST context with that context number (that is, the context which was created first).

4.10 generic_hash_destroy()

`void generic_hash_destroy()`

Frees all the (possibly enormous) memory used by the hash function, deletes all the contexts (so that the next call to `generic_hash_init` will create context 0), and disables Custom Context Mode if it was active.

Call this at a point when you have no further use for the hash function, or simply when you are defining a new set of hash functions (such as when a boardsize or variant is changed in `GameSpecificMenu()`).

5 Examples

There are examples all over the modules on how these functions are to be used. Try looking at any of these source files:

m9mm.c, mabalone.c, masalto.c, mataxx.c, mbaghchal.c, mblocking.c, mcambio.c, mdao.c, mdinododgem.c, mdodgemhash.c, mfandam.c, mfoxes.c, mhshogi.c, miceblocks.c, mkono.c, mlewth.c, mloa.c, mnuttt.c, mothello.c, mparadux.c, mpylos.c, mqland.c, mquickchess.c, mrcheckers.c, mseega.c, mshogi.c, msquaredance.c, mswans.c, mtilechess.c, mtopitop.c, mtore.c, mttc.c, mtttier.c, mwinkers.c, mwuzhi.c, mxigua.c

Notice that most of these are basic rearranger or dartboard-based games, which fit nicely into the use of `generic_hash`. Most of these do not need to use contexts and so will show how you can work with simply one main global hash. A few, such as mtopitop.c, are non-Tier-Gamesman games which nonetheless work with multiple contexts.

For a good simple example of how to initialize and switch between multiple hash contexts, try looking at mtttier.c. Skip the Tier-Gamesman stuff if you're not interested in the tier aspects (or study it if you are!). The snippet of code near the middle of `SetupTierStuff()` initializes 10 hash contexts. Correspondingly, `PositionToBlankOX` (a.k.a. "unhash") and `BlankOXToPosition` (a.k.a. "hash") deal with switching to the appropriate context and calling the appropriate function to hash/unhash the values.