Generic Hash Symmetries

Documentation for Programmers

by

Albert Shau

Last updated 2008-2-12

Table of Contents:

1. Objective of this Document 2
2. API between Game Modules and Generic Hash 2
   2.1 `generic_hash_init_sym` 2
   2.2 `generic_hash_add_sym` 3
3. Implementation Details 5
   3.1 Overview 5
   3.2 How Symmetries are Represented 5
   3.3 How the Canonical Position is Found 5

1. Objective of this Document

This document targets two types of developers. The first type is seeking to add symmetries to games that use generic hash. The second type is seeking to gain an understanding of the details behind how symmetries are handled in case the implementation needs to be changed.

This document assumes the reader is familiar with the C programming language and the general concept behind symmetries. There is a Symmetries doc that gives an overview of what symmetries are and which games have them.

After reading through this document you should be able to add symmetries to any generic hash game by adding just a few lines of code. You should also gain an understanding of what the solver requires for symmetries, and how that is abstracted away from game module writers.

2. API Between Game Modules and Generic Hash

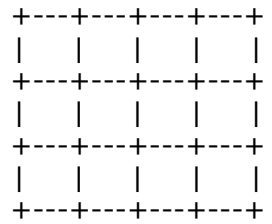
2.1 `generic_hash_init_sym`

The only essential function the game module needs to call to support symmetries is:

```
void generic_hash_init_sym(int boardType, int numRows, int numCols,
int* reflections, int numReflects, int* rotations, int numRots);
```

This function should be called after `generic_hash_init` is called.

Rectangular and hexagonal boards are supported. The `boardType` variable should be 0 for rectangular boards and 1 for hexagonal boards. Rows go from left to right whereas columns go from up to down. The following table:



has 3 rows and 4 columns, which should be the values of numRows and numCols respectively. For a hexagonal board, `numCols` should be equal to the number of columns in the very first row.

The `reflections` variable is an array of integers that describe the axes of rotation. If there is a reflection across the x axis, the integer 0 should be included in this array. If there is a reflection across the y axis, the integer 90 should be included. There is no difference between 0 and 180, 90 and 270, 45 and 225, etc. The `rotations` array is an

array of integers that describe the clockwise rotations present in the game. If there are 90, 180, and 270 degree rotations, than the `rotations` array should consist of those three integers. These arrays can also be set to NULL and their lengths set to 0 there are none of those symmetries present.

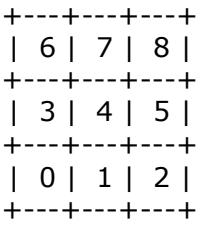
The standard game of tic-tac-toe has three rotations (90, 180, 270) and four reflections (0, 45, 90, 135). To add symmetries to tic-tac-toe, all one needs to do is write

```
int rotations[3] = {90, 180, 270};
int reflections[4] = {0, 45, 90, 135};
generic_hash_init_sym(0, 3, 3, reflections, 4, rotations, 3);
```

These three lines are all it takes!

2.2 `generic_hash_add_sym`

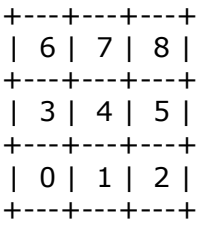
There is also support for symmetries that are not reflections or rotations. Suppose you are playing a variant of tic-tac-toe where for some reason, if you switch the bottom middle piece with the top right piece, you get the same board. With the board numbered like below:



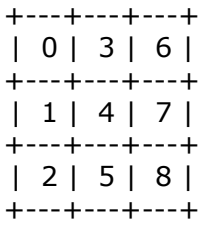
This means the piece at position 1 can be swapped with the piece at position 8. Suppose that is the only symmetry present in this game. In order to add symmetries to this game, you need to use the following function:

```
void generic_hash_add_sym(int* symToAdd);
```

This function takes in an array that describes a symmetry. It is easiest to describe how the array does this through an example. Suppose we wanted to add a 90 degree rotation using `generic_hash_add_sym`. This means we know this board:



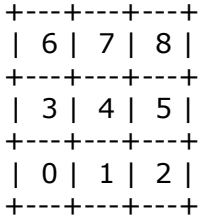
Is equivalent to this board:



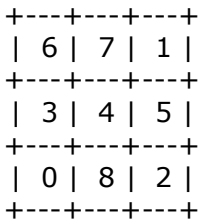
This means we want to write the following code:

```
int sym[9] = {2, 5, 8, 1, 4, 7, 0, 3, 6};
generic_hash_add_sym(sym);
```

The array, `sym`, says that whatever piece was in the 2nd position should now be moved to the 0th position, whatever piece was in the 5th position should now be moved to the 1st position, etc. Going back to our original example, we know this board:



is equivalent to this board:



To add symmetries to this game, we would write the following:

```
int sym[9] = {0, 8, 2, 3, 4, 5, 6, 7, 1};
generic_hash_init_sym(0, 4, 4, NULL, 0, NULL, 0);
generic_hash_add_sym(sym);
```

This may seem like an unnecessary feature, but there are some games (I believe Nine Men's Morris is one of them), that have symmetries that are not reflections or rotations.

3. Implementation Details

We will now go over some of the implementation details in case there are still bugs that need to be worked out or the implementation eventually needs to be changed.

3.1 Overview

The solver requires a `gCanonicalPosition` function to be defined in order to solve with symmetries. This function pointer is set to `generic_hash_canonicalPosition` in `generic_hash_init_sym`. This function loops through all the symmetries, applies them to the standard board, hashes each of these applied symmetries, and returns the lowest position as the canonical position.

3.2 How Symmetries are Represented

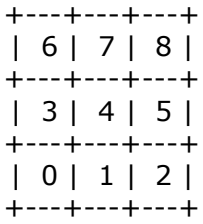
A symmetry is represented by a structure, `symEntry`, that includes its `type` (reflection, rotation, or unknown), its `angle`, and an array `sym`. The array is exactly the same as the array described in the section for `generic_hash_add_sym`. These structures are all nodes in a linked list, `symmetriesList`. The first node of this list is the identity. Knowing this, we can now look at pseudocode for `generic_hash_canonicalPosition`.

3.3 How the Canonical Position is Found

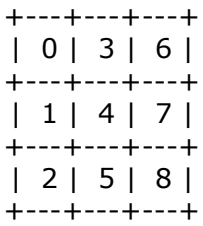
This isn't exactly how it is done, but it's the basic idea.

```
POSITION generic_hash_canonicalPosition(POSITION pos) {
      board = unhash(pos);
      struct symEntry* symIndex = symmetriesList->next;

      minPos = hash(board);
      while (symIndex != NULL) {
        for (i = 0; i < boardSize; i++) {
            tempBoard[i] = board[symIndex->sym[i]];
        }
        tempPos = hash(tempBoard);
        if tempPos < minPos,
            minPos = tempPos;
        symIndex = symIndex->next;
      }

      return minPos;
}
```

The for loop is where:



becomes:



The transformed board is then hashed and compared to the minimum position seen so far. Lastly, the minimum position is returned. Since this function is inside generic hash, it is possible to avoid some redundant work. These improvements are not described because they are difficult to explain without a good understanding of how generic hash works.

Before `generic_hash_canonicalPosition` can be run, the list of symmetries must be initialized. The `reflections array`, `rotations` arrays, board size, row size, and column size must be correctly translated into the `sym` arrays in each node of the symmetry list or the algorithm fails. This is done in `generic_hash_init_sym` and is a huge mess; quite frankly I don't remember how it works so it will not be described in detail here. There is a `printSymmetries` function for debugging purposes that can verify the correctness of that chunk of code.