CMPH - C Minimal Perfect Hashing Library

Motivation

A perfect hash function maps a static set of n keys into a set of m integer numbers without collisions, where m is greater than or equal to n. If m is equal to n, the function is called minimal.

Minimal perfect hash functions are widely used for memory efficient storage and fast retrieval of items from static sets, such as words in natural languages, reserved words in programming languages or interactive systems, universal resource locations (URLs) in Web search engines, or item sets in data mining techniques. Therefore, there are applications for minimal perfect hash functions in information retrieval systems, database systems, language translation systems, electronic commerce systems, compilers, operating systems, among others.

The use of minimal perfect hash functions is, until now, restricted to scenarios where the set of keys being hashed is small, because of the limitations of current algorithms. But in many cases, to deal with huge set of keys is crucial. So, this project gives to the free software community an API that will work with sets in the order of billion of keys.

Probably, the most interesting application for minimal perfect hash functions is its use as an indexing structure for databases. The most popular data structure used as an indexing structure in databases is the B+ tree. In fact, the B+ tree is very used for dynamic applications with frequent insertions and deletions of records. However, for applications with sporadic modifications and a huge number of queries the B+ tree is not the best option, because practical deployments of this structure are extremely complex, and perform poorly with very large sets of keys such as those required for the new frontiers database applications.

For example, in the information retrieval field, the work with huge collections is a daily task. The simple assignment of ids to web pages of a collection can be a challenging task. While traditional databases simply cannot handle more traffic once the working set of web page urls does not fit in main memory anymore, minimal perfect hash functions can easily scale to hundred of millions of entries, using stock hardware.

As there are lots of applications for minimal perfect hash functions, it is important to implement memory and time efficient algorithms for constructing such functions. The lack of similar libraries in the free software world has been the main motivation to create the C Minimal Perfect Hashing Library (gperf is a bit different, since it was conceived to create very fast perfect hash functions for small sets of keys and CMPH Library was conceived to create minimal perfect hash functions for very large sets of keys). C Minimal Perfect Hashing Library is a portable LGPLed library to generate and to work with very efficient minimal perfect hash functions.

Description

The CMPH Library encapsulates the newest and more efficient algorithms in an easy-to-use, production-quality, fast API. The library was designed to work with big entries that cannot fit in the main memory. It has been used successfully for constructing minimal perfect hash functions for sets with more than 100 million of keys, and we intend to expand this number to the order of billion of keys. Although there is a lack of similar libraries, we can point out some of the distinguishable features of the CMPH Library:

- Fast.
- Space-efficient with main memory usage carefully documented.
- The best modern algorithms are available (or at least scheduled for implementation :-)).
- Works with in-disk key sets through of using the adapter pattern.
- Serialization of hash functions.
- Portable C code (currently works on GNU/Linux and WIN32 and is reported to work in OpenBSD and Solaris).
- Object oriented implementation.
- Easily extensible.
- Well encapsulated API aiming binary compatibility through releases.
- Free Software.

Supported Algorithms

- CHD Algorithm:
  - It is the fastest algorithm to build PHFs and MPHFs in linear time.
  - It generates the most compact PHFs and MPHFs we know of.
  - It can generate PHFs with a load factor up to *99 %*.
  - It can be used to generate *t*-perfect hash functions. A *t*-perfect hash function allows at most *t* collisions in a given bin. It is a well-known fact that modern memories are organized as blocks which constitute transfer unit. Example of such blocks are cache lines for internal memory or sectors for hard disks. Thus, it can be very useful for devices that carry out I/O operations in blocks.
  - It is a two level scheme. It uses a first level hash function to split the key set in buckets of average size determined by a

parameter *b* in the range *[1,32]*. In the second level it uses displacement values to resolve the collisions that have given rise to the buckets.
    - It can generate MPHFs that can be stored in approximately *2.07* bits per key.
    - For a load factor equal to the maximum one that is achieved by the BDZ algorithm (*81 %*), the resulting PHFs are stored in approximately *1.40* bits per key.
- BDZ Algorithm:
    - It is very simple and efficient. It outperforms all the ones below.
    - It constructs both PHFs and MPHFs in linear time.
    - The maximum load factor one can achieve for a PHF is *1/1.23*.
    - It is based on acyclic random 3-graphs. A 3-graph is a generalization of a graph where each edge connects 3 vertices instead of only 2.
    - The resulting MPHFs are not order preserving.
    - The resulting MPHFs can be stored in only *(2 + x)cn* bits, where *c* should be larger than or equal to *1.23* and *x* is a constant larger than *0* (actually, x = 1/b and b is a parameter that should be larger than 2). For *c = 1.23* and *b = 8*, the resulting functions are stored in approximately 2.6 bits per key.
    - For its maximum load factor (*81 %*), the resulting PHFs are stored in approximately *1.95* bits per key.
- BMZ Algorithm:
    - Construct MPHFs in linear time.
    - It is based on cyclic random graphs. This makes it faster than the CHM algorithm.
    - The resulting MPHFs are not order preserving.
    - The resulting MPHFs are more compact than the ones generated by the CHM algorithm and can be stored in *4cn* bytes, where *c* is in the range *[0.93,1.15]*.
- BRZ Algorithm:
    - A very fast external memory based algorithm for constructing minimal perfect hash functions for sets in the order of billions of keys.
    - It works in linear time.
    - The resulting MPHFs are not order preserving.
    - The resulting MPHFs can be stored using less than *8.0* bits per key.
- CHM Algorithm:
    - Construct minimal MPHFs in linear time.
    - It is based on acyclic random graphs
    - The resulting MPHFs are order preserving.
    - The resulting MPHFs are stored in *4cn* bytes, where *c* is greater than 2.
- FCH Algorithm:
    - Construct minimal perfect hash functions that require less than 4 bits per key to be stored.
    - The resulting MPHFs are very compact and very efficient at evaluation time
    - The algorithm is only efficient for small sets.
    - It is used as internal algorithm in the BRZ algorithm to efficiently solve larger problems and even so to generate MPHFs that require approximately 4.1 bits per key to be stored. For that, you just need to set the parameters -a to brz and -c to a value larger than or equal to 2.6.

News for version 2.0

Cleaned up most warnings for the c code.

Experimental C++ interface (--enable-cxxmph) implementing the BDZ algorithm in a convenient interface, which serves as the basis for drop-in replacements for std::unordered_map, sparsehash::sparse_hash_map and sparsehash::dense_hash_map. Potentially faster lookup time at the expense of insertion time. See cxxmpph/mph_map.h and cxxmph/mph_index.h for details.

News for version 1.1

Fixed a bug in the chd_pc algorithm and reorganized tests.

News for version 1.0

This is a bugfix only version, after which a revamp of the cmph code and algorithms will be done.

News for version 0.9

- The CHD algorithm, which is an algorithm that can be tuned to generate MPHFs that require approximately 2.07 bits per key to be stored. The algorithm outperforms the BDZ algorithm and therefore is the fastest one available in the literature for sets that can be treated in internal memory.
- The CHD_PH algorithm, which is an algorithm to generate PHFs with load factor up to *99 %*. It is actually the CHD algorithm without the ranking step. If we set the load factor to *81 %*, which is the maximum that can be obtained with the BDZ algorithm, the resulting functions can be stored in *1.40* bits per key. The space requirement increases with the load factor.
- All reported bugs and suggestions have been corrected and included as well.

News for version 0.8

- An algorithm to generate MPHFs that require around 2.6 bits per key to be stored, which is referred to as BDZ algorithm. The algorithm is the fastest one available in the literature for sets that can be treated in internal memory.
- An algorithm to generate PHFs with range m = cn, for c > 1.22, which is referred to as BDZ_PH algorithm. It is actually the BDZ algorithm without the ranking step. The resulting functions can be stored in 1.95 bits per key for *c = 1.23* and are considerably faster than the MPHFs generated by the BDZ algorithm.
- An adapter to support a vector of struct as the source of keys has been added.
- An API to support the ability of packing a perfect hash function into a preallocated contiguous memory space. The computation of a packed function is still faster and can be easily mmapped.
- The hash functions djb2, fnv and sdbm were removed because they do not use random seeds and therefore are not useful for MPHFs algorithms.
- All reported bugs and suggestions have been corrected and included as well.

News log

Examples

Using cmph is quite simple. Take a look.

```
#include <cmph.h>
#include <string.h>
// Create minimal perfect hash function from in-memory vector
int main(int argc, char **argv)
{

    // Creating a filled vector
    unsigned int i = 0;
    const char *vector[] = {"aaaaaaaaaa", "bbbbbbbbbb", "cccccccccc", "dddddddddd", "eeeeeeeeee",
        "ffffffffff", "gggggggggg", "hhhhhhhhhh", "iiiiiiiiii", "jjjjjjjjjj"};
    unsigned int nkeys = 10;
    FILE* mphf_fd = fopen("temp.mph", "w");
    // Source of keys
    cmph_io_adapter_t *source = cmph_io_vector_adapter((char **)vector, nkeys);

    //Create minimal perfect hash function using the brz algorithm.
    cmph_config_t *config = cmph_config_new(source);
    cmph_config_set_algo(config, CMPH_BRZ);
    cmph_config_set_mphf_fd(config, mphf_fd);
    cmph_t *hash = cmph_new(config);
    cmph_config_destroy(config);
    cmph_dump(hash, mphf_fd);
    cmph_destroy(hash);
    fclose(mphf_fd);

    //Find key
    mphf_fd = fopen("temp.mph", "r");
    hash = cmph_load(mphf_fd);
    while (i < nkeys) {
        const char *key = vector[i];
        unsigned int id = cmph_search(hash, key, (cmph_uint32)strlen(key));
        fprintf(stderr, "key:%s -- hash:%u\n", key, id);
        i++;
    }

    //Destroy hash
    cmph_destroy(hash);
    cmph_io_vector_adapter_destroy(source);
    fclose(mphf_fd);
    return 0;
}
```

Download vector_adapter_ex1.c. This example does not work in versions below 0.6. You need to update the sources from GIT to make it work.

```
#include <cmph.h>
#include <stdio.h>
#include <string.h>
 // Create minimal perfect hash function from in-disk keys using BDZ algorithm
int main(int argc, char **argv)
{
       //Open file with newline separated list of keys
      FILE * keys_fd = fopen("keys.txt", "r");
      cmph_t *hash = NULL;
      if (keys_fd == NULL)
      {
        fprintf(stderr, "File \"keys.txt\" not found\n");
        exit(1);
      }
```

```
        // Source of keys
        cmph_io_adapter_t *source = cmph_io_nlfile_adapter(keys_fd);

        cmph_config_t *config = cmph_config_new(source);
        cmph_config_set_algo(config, CMPH_BDZ);
        hash = cmph_new(config);
        cmph_config_destroy(config);

        //Find key
        const char *key = "jjjjjjjjjj";
        unsigned int id = cmph_search(hash, key, (cmph_uint32)strlen(key));
        fprintf(stderr, "Id:%u\n", id);
        //Destroy hash
        cmph_destroy(hash);
        cmph_io_nlfile_adapter_destroy(source);
        fclose(keys_fd);
        return 0;
}
```

Download file_adapter_ex2.c and keys.txt. This example does not work in versions below 0.8. You need to update the sources from GIT to make it work.

Click here to see more examples

The cmph application

cmph is the name of both the library and the utility application that comes with this package. You can use the cmph application for constructing minimal perfect hash functions from the command line. The cmph utility comes with a number of flags, but it is very simple to create and to query minimal perfect hash functions:

```
$ # Using the chm algorithm (default one) for constructing a mphf for keys in file keys_file
$ ./cmph -g keys_file
$ # Query id of keys in the file keys_query
$ ./cmph -m keys_file.mph keys_query
```

The additional options let you set most of the parameters you have available through the C API. Below you can see the full help message for the utility.

```
usage: cmph [-v] [-h] [-V] [-k nkeys] [-f hash_function] [-g [-c algorithm_dependent_value][-s seed] ]
            [-a algorithm] [-M memory_in_MB] [-b algorithm_dependent_value] [-t keys_per_bin] [-d tmp_dir]
            [-m file.mph] keysfile
Minimum perfect hashing tool

  -h   print this help message
  -c   c value determines:
        * the number of vertices in the graph for the algorithms BMZ and CHM
        * the number of bits per key required in the FCH algorithm
        * the load factor in the CHD_PH algorithm
  -a   algorithm - valid values are
        * bmz
        * bmz8
        * chm
        * brz
        * fch
        * bdz
        * bdz_ph
        * chd_ph
        * chd
  -f   hash function (may be used multiple times) - valid values are
        * jenkins
  -V   print version number and exit
  -v   increase verbosity (may be used multiple times)
  -k   number of keys
  -g   generation mode
  -s   random seed
  -m   minimum perfect hash function file
  -M   main memory availability (in MB) used in BRZ algorithm
  -d   temporary directory used in BRZ algorithm
  -b   the meaning of this parameter depends on the algorithm selected in the -a option:
        * For BRZ it is used to make the maximal number of keys in a bucket lower than 256.
          In this case its value should be an integer in the range [64,175]. Default is 128.

        * For BDZ it is used to determine the size of some precomputed rank
          information and its value should be an integer in the range [3,10]. Default
          is 7. The larger is this value, the more compact are the resulting functions
          and the slower are them at evaluation time.

        * For CHD and CHD_PH it is used to set the average number of keys per bucket
          and its value should be an integer in the range [1,32]. Default is 4. The
          larger is this value, the slower is the construction of the functions.
          This parameter has no effect for other algorithms.

  -t   set the number of keys per bin for a t-perfect hashing function. A t-perfect
```

```
        hash function allows at most t collisions in a given bin. This parameter applies
        only to the CHD and CHD_PH algorithms. Its value should be an integer in the
        range [1,128]. Defaul is 1
    keysfile     line separated file with keys
```

Additional Documentation

FAQ

Downloads

Use the project page at sourceforge: http://sf.net/projects/cmph

License Stuff

Code is under the LGPL and the MPL 1.1.

Enjoy!

Davi de Castro Reis

Djamel Belazzougui

Fabiano Cupertino Botelho

Nivio Ziviani

Last Updated: Sat Jun 9 04:07:31 2012