User Guide for Heterogeneous Subset Sampling Library

October 9, 2011

Contents

1 Outline

Given a domain set $S = \{0, 1, \ldots, n-1\}$ and an associated inclusion probability function $I : S \rightarrow [0, 1]$, a subset drawing algorithm is used to draw a subset R, possibly many subsets, of S where $\Pr[e \in R] = I(e)$. We use C++ to implement the algorithms in the form of **class**. There are two main member functions in the classes, which are the **constructor** and **drawing** function. The algorithms are preprocessed in the **constructor** and a drawn subset can be obtained by invoking their **drawing** function. Note that the HSS library requires a open third-party library GSL and we show how to install it in the appendix.

In this article, **bold text** is used for explaining the terms in the programs and code segments are written in `typewriter text`. The remaining is organized as follows. In Section 2, we details the steps how the algorithms are used. For inspecting the internal codes, we state the module purpose and the parameters' description of the building blocks in Section 3. Then, in Section 4, a list of testing platforms is shown and the testing programs are briefly explained.

2 Subset Drawing Algorithms

The subset drawing algorithms introduced in this section are

- **naive**<**size**>,
- **sieve**<**size**>,
- **partition**<**size**>,

where **size** is an integral type to denote $n = |S|$. There are two main member functions in each subset drawing algorithm, which are the **constructor** and **drawing** function.

2.1 constructor

naive	**naive(size** *n***)**
sieve	**sieve(double const ***_prob_**, size** *n***)**
partition	**partition(double const ***_prob_**, size** *n***, size const ***_cut_points_**, size** *k***)**
partition	**partition(double ***_prob_**, size** *n***, int** *algorithm_type***)**

Table 1: Class constructors of subset drawing algorithms.

prob[0 .. n-1] refers to n inclusion probabilities and **cut_points[0 .. k]** denote how to partition the original single **prob[0 .. n-1]** into k ones. For example,

```
partition(prob, 5, cut_points={0, 2, 5}, 2)
```

divides **{prob[0 .. 4]}** into **{prob[0 .. 1]}** and **{prob[2 .. 4]}**. In the partition algorithm, an user can give **cut_points** to divide **prob** as shown above or merely specifies the preferred dividing algorithm among **HSS_OPTIMUM, HSS_FIX_APPROX, and HSS_DYN_APPROX (Table 2).**

HSS_FIX_APPROX	A fixed partition method of approximation factor 2.
HSS_OPTIMUM	The optimum algorithm of the fittest cover problem.
HSS_DYN_APPROX	The approximation algorithm of the fittest cover problem.

Table 2: Preprocessing algorithms for the partition algorithm.

Note

The 4-th constructor in Table 1 modifies the content in **prob**.

Example

```
#include <hss.h>

int main(){

    double prob[5] = {0.1, 0.2, 0.3, 0.7, 0.9};
    int cut_points[3] = {0, 3, 5};

    naive<int> drawer1(5);
    sieve<int> drawer2(prob, 5);
    partition<int> drawer3(prob, 5, cut_points, 2);
    partition<int> drawer4(prob, 5, HSS_DYN_APPROX);

    return 0;
}
```

See Also

include/naive.h, include/sieve.h, include/partition.h, test/test[5-8].cpp

2.2 drawing member function

Three classes have the same member function used to draw a subset, which is

size drawing(double **prob*, size_t **subset*).

To reduce the amount of used space, the class objects do not have a duplicate of **prob**. Hence, **prob** should be given for each **drawing**. During execution, the value of elements in **prob** can only be decreased down, otherwise the drawn sample

R does not satisfies the condition that $\Pr[e \in R] = I(e)$. The return value *ret* of **drawing(prob, subset)** is the size of drawn sample placed in **subset[0 ..** *ret*-**1].**

Example

```
#include <cstdio>
#include <hss.h>

int main(){

    double prob[5] = {0.1, 0.2, 0.3, 0.7, 0.9};
    int subset[5];

    partition<int> drawer(prob, 5, HSS_DYN_APPROX);

    for(int i=0; i<10; ++i){
        int ret = drawer.drawing(prob, subset);
        for(int j=0; j<ret; ++j){
            printf("%lf ", prob[subset[j]]);
        }
        printf("\n");
    }

    return 0;
}
```

See Also

include/naive.h, include/sieve.h, include/partition.h, test/test[5-8].cpp

3 Building Blocks

In this section, the building blocks used in the constructors of the partition algorithm are introduced.

3.1 The cuckoo hashing

The cuckoo hashing is a hash table used to store a series of keys and their associated data, which supports amortized $O(1)$ time insertion, $O(1)$ time lookup, and $O(1)$ time deletion [1]. We follow the convention of the well known STL map[1] to implement

cuckoo_hash<**key, data, size**>

[1] http://www.sgi.com/tech/stl/Map.html

and its the member functions, including

- **cuckoo_hash(size), ∼cuckoo_hash()**
- **iterator begin(), iterator end(), size size()**
- **iterator find(key), data &operator[ ](key)**
- **iterator, const_iterator**

Note

1. **key** is a native C++ data type of size no more than 8 bytes.
2. **data** is the type of associated data.
3. **size** is a signed integral data type, which counts the number of keys in a hash table. The maximum number of stored keys should not exceed the maximum of **size** minus 1.
4. Users can define the initial table size by assigning the parameter **ptabsize** in the constructor **cuckoo_hash(ptabsize)**. Once the space run out, the cuckoo_hash automatically double the table size.

Example

```
#include <iostream>
#include <hss.h>

int main(){

    double input[5] = {0.1, 0.1, 0.2, 0.2, 0.2};

    cuckoo_hash<double, int, int> s(100);
    cuckoo_hash<double, int, int>::const_iterator ite;

    for(int i=0; i<5; ++i){
        double key = input[i];
        if(s.find(key) == s.end()){ // if key is not found
            s[key] = 1;
        }else{
            ++ s[key];
        }
    }

    for(ite=s.begin(); ite!=s.end(); ++ite){
        std::cout << ite->first << " " << ite->second << std::endl;
```

```
    }
}
```

See Also

include/cuckoo.h, test/test1.cpp

3.2 The radix sorting

radixsort(data * *first***, data *** *last***, std::vector**<**int**> *lsb***)** is a sorting procedure used to sort the elements of fixed precision between **first** and **last** by the least significant bits, the second least significant bits, ..., the most significant bits defined in **lsb**. The required computation time is $O(nb) = O(n)$ [2] where n is the number of elements and b is a constant, due to fixed precision, denoting the number of bits in an element.

data	feasible **lsb**
int	{8, 8, 8, 8} or {16, 8, 4, 4}
double	{13, 13, 13, 13, 12} or {26, 26, 12}
long long	{16, 16, 16, 16} or {15, 15, 15, 15, 4}

Table 3: **data** and the corresponding **lsb**.

Note

1. **data** a data type that support right-shiftment » and bitwise-and &.
2. The elements between **first** and **last** should be non-negative.
3. In the case of **double**, use **long long** as an adapter and invoke **radixsort(first, last, lsb, adapter)**.

Example

```
#include <iostream>
#include <vector>
#include <hss.h>

int main(){

    long long s[3] = {3, 1, 2};
    int lsb1[4] = {16, 16, 16, 16};

    radixsort(s, s+3, *new std::vector<int>(lsb1, lsb1+4));
```

```
    double r[4] = {5.0, 1.0, 3.0, 2.0};
    int lsb2[5] = {13, 13, 13, 13, 12};

    radixsort(r, r+4, *new std::vector<int>(lsb2, lsb2+5), *new long long);
}
```

See Also

include/radix.h, test/test2.cpp test/test3.cpp

3.3 The fittest covering

Given a non-decreasing histogram H of n sorted values `value[0]` $\leq \cdots \leq$ `value[n-1]` and their frequencies `freq[0], ..., freq[n-1]`,

fittest_cover(data ****value*, data ****freq***, size** n**, size ****cut_points***, size** k**)**

can be applied to find a k-stepwise function which covers H with least under area in $O(nk)$ time and uses $O(n+k)$ space [3], as Figure 1 shown. The thicker line in the middle and the right histogram denote, respectively, a one-stepwise function and a two-stepwise function which cover H with least under area. The red dotted line in the right histgram is a two-stepwise functions that covers H but whose under area is not the least.

The calculated k-stepwise function is placed in `cut_points`. For example, in the right histogram of Figure 1, the returned `cut_points` is $[0,1,3]$. The return value of function is the least under area.

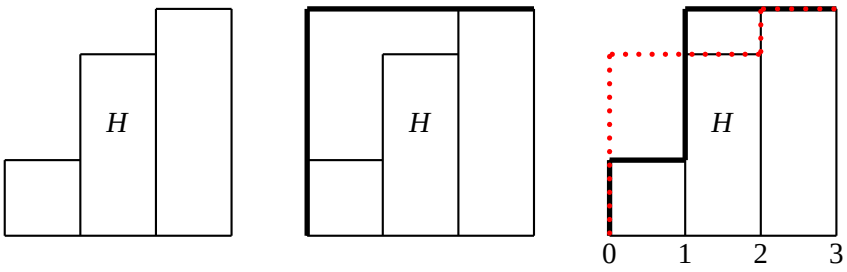

Figure 1: Sketch of k-stepwise functions which cover H.

Note

1. The values in **value** are distinct, non-negative and sorted. The sum $\sum_{x \in \textbf{value}} x$ should not exceed the maximum of **data** divided by 2.
2. The types of **value** and **freq** should be identical.
3. **size** is a signed integral type.

Example

```
#include <iostream>
#include <hss.h>

int main(){

    double value[6] = {0.1, 0.2, 0.3, 0.4, 0.5, 0.6};
    double freq[6] = {1, 5, 1, 9, 3, 7};
    int n = 6;

    int k = 3;
    int *cut_points = new int [k+1];

    double ret = fittest_cover(value, freq, n, cut_points, k);

    std::cout << "under area = " << ret << std::endl;

    for(int i=0; i<=k; ++i){
        std::cout << cut_points[i] << " ";
    }
    std::cout << std::endl;
}
```

See Also

include/smawk.h, test/test4.cpp

4 Testing Platforms and Programs

To use the HSS library, an open third-party library GSL[2] should be properly installed in advance. The installation guide for GSL is shown in the appendix. We test the sanity of this library on different platforms listed in Table 4 by a series of testing programs as Table 5 shown.

CPU	Memory	Operating System	Compiler
Intel Xeon X5690	48GB	FreeBSD 8.2	GCC 4.2.1
Intel Xeon X5365	48GB	Ubuntu 10.04	GCC 4.4.3
Intel Core2 Quad Q6600	8GB	SunOS 5.11	GCC 3.4.3

Table 4: Description of testing platforms.

[2]GNU Scientific Library http://www.gnu.org/software/gsl/

Program	Description
test1.cpp	checks the behavior of cuckoo hashing is correct.
test2.cpp	checks whether the radixsort sort an array of floating-point numbers correctly.
test3.cpp	checks whether the radixsort sort an array of integral number correctly.
test4.cpp	checks the calculated result of the fittest cover is the same as that of a naive dynamic programming.
test5.cpp	compares the drawn samples generated from the naive and sieve algorithms.
test6.cpp	compares the drawn samples generated from the sieve and partition algorithms.
test7.cpp	compares the drawn samples generated from two variations of the partition algorihtm.
test8.cpp	compares the drawn samples generated from other two variations of the partition algorithm.

Table 5: Description of testing programs.

References

[1] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[2] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[3] Meng-Tsung Tsai, Da-Wei Wang, and Tsan-sheng Hsu. Approximating the fittest cover problem. Manuscript, 2011.

A GSL Installation Guide

Step1: Fetch the GSL archive from the official site, http://www.gnu.org/s/gsl/ and place it on /tmp. The file name should look like 'gsl-*.*.tar.gz'.

Step2: Decompress the archive by

```
/tmp$ tar xvfz gsl-1.15.tar.gz
```

Step3: Enter the decompressed directory and configure the installation setting for 32-bit machine

```
/tmp/gsl-1.15$ ./configure
```

or for 64-bit machine

```
/tmp/gsl-1.15$ ./configure CC=cc CFLAGS="-64" LDFLAGS="-64"
```

Configure as the above will make the files installed in the directory /usr/local. To

install it on a different directory, say /tmp/install, you should

```
/tmp/gsl-1.15$ ./configure -prefix=/tmp/install
```

or

```
/tmp/gsl-1.15$ ./configure -prefix=/tmp/install
CC=cc CFLAGS="-64" LDFLAGS="-64"
```

Step4: Compile the source files and build the installation binary by

```
/tmp/gsl-1.15$ make
```

If you encountered any problems in Step3 or Step4 and wish to restart the installation procedure, issue the following command before you restart

```
/tmp/gsl-1.15$ make clean
```

Step5: Install the files to the default directory, /usr/local, by

```
/tmp/gsl-1.15$ sudo make install
```

or simply by

```
/tmp/gsl-1.15$ make install
```

if it does not require the superuser privilege to access the directory you specify in Step3.

The GSL should be properly installed now. If you install GSL in the default directory, ensure the sanity by check the files
/usr/local/include/gsl/gsl_rng.h
/usr/local/lib/libgsl.a
are existing and you can use our HSS library now.

Or, if you install GSL in /tmp/install, check the files
/tmp/install/include/gsl/gsl_rng.h
/tmp/install/lib/libgsl.a
are existing and change the compiler flag in the Makefile with

```
g++ -ansi -Wall -O3 -I/tmp/install/include -L/tmp/install/lib
-I../include -o test test.cpp -lgsl -lgslcblas
```

before using the HSS library.
You might notice that there is no space between -I and /tmp/install/include; neither is -L.

To uninstall GSL, it can be done by

```
/tmp/gsl-1.15$ sudo make uninstall
```

or, if the superuser privilege is unnecessary,

```
/tmp/gsl-1.15$ make uninstall
```