Fall 2012 BIOSTAT 615/815 Problem Set #3

Due is Saturday October 20th, 2012 11:59PM by google document (shared to hmkang@umich.edu and atks@umich.edu) containing the source code and answers to the questions. Also email of the compressed tar.gz file containing all the source codes.

BIOSTAT615 students need to solve two of the three problems, and BIOSTAT815 students need to solve all three problems.

BIOSTAT615 students who attempted to solve all three problems will be graded based on the two problems with higher scores. 5% of extra credit will be given if all three problems were scored above 90%.

Problem 1. Binomial Coeffficient

Write a program that calculates binomial coefficient $\binom{n}{k}$, based on the following recursive rule.

$$\begin{aligned}\binom{n}{k} &= \binom{n-1}{k} + \binom{n-1}{k-1} \\ \binom{n}{0} &= \binom{n}{n} = 1\end{aligned}$$

A suggested skeleton of the program is given below.

(a) (30 pts) Write down the full function `choose()`

(b) (10 pts) compute the value $\binom{30}{15}$ and $\binom{60}{30}$ using the implemented program.

(c) (10 pts) What happens if all `double` in the program is replaced into `int`? Breifly explain why.

```
#include <iostream>
#include <vector>

double choose(int n, int k, std::vector< std::vector<double> > & stored) {
   // NEED TO FILL THIS FUNCTION
}

int main(int argc, char** argv) {
  if ( argc != 3 ) {
    std::cerr << "Usage: " << argv[0] << " [n] [k] " << std::endl;
    return -1;
  }
  int n = atoi(argv[1]);
  int k = atoi(argv[2]);
  if ( k > n ) {
    std::cerr << "n = " << n << " is smaller than k = " << k << std::endl;
    return -1;
  }

  std::vector< std::vector<double> > v(n+1);
  for(int i=0; i < n+1; ++i) {
    v[i].resize(k+1,0);
  }

  double nCk = choose(n, k, v);
  std::cerr << "choose(" << n << "," << k << ") = " << nCk << std::endl;
  return 0;
}
```

Problem 2. Manhattan Tourist Problem

(a) (25 pts) Write a program that calculates the optimal cost and print an optimal path for Manhattan tourist problem. Follow the skeleton below. You may copy `~hmkang/Public/615/include/Matrix615.h` instead of typing it yourself. An example run of the program will take two input files - the horizonal cost matrix and the vertical cost matrix. For $r \times c$ Manhattan tourist problem, the horizontal cost matrix must be $r \times (c-1)$ matrix, and the veritcal cost matrix must be $(r-1) \times c$ matrix. The expected output from the example from the lecture note should print the optimal cost and the path to the optimal cost as appeared below.

```
$ ./bin/hw-3-2 hw.txt vw.txt
Optimal cost is 21
A path to the optimal cost is:
Node (0,0) - Cost = 0
Node (0,1) - Cost = 4
Node (0,2) - Cost = 6
Node (0,3) - Cost = 6
Node (1,3) - Cost = 8
Node (2,3) - Cost = 8
Node (3,3) - Cost = 16
Node (4,3) - Cost = 16
Node (4,4) - Cost = 21
```

The two input matrices used for this run is as follows.

`hw.txt`

```
4 2 0 7
7 4 5 9
6 8 1 0
1 6 4 7
1 5 8 5
```

`vw.txt`

```
0 6 6 2 4
9 7 1 0 6
1 8 4 8 9
3 6 6 0 7
```

Your `hw-3-2.cpp` is given as below.

```cpp
#include <iostream>
#include "Matrix615.h"
#include "MTP.h"

int main(int argc, char** argv) {
  if ( argc != 3 ) {
    std::cerr << "Usage: " << argv[0] << " [hw.txt] [vw.txt]" << std::endl;
  }

  MTP mtp(argv[1], argv[2]);

  int cost = mtp.computeOptimalCost();
  std::cout << "Optimal cost is " << cost << std::endl;
  std::cout << "A path to the optimal cost is:"<< std::endl;
  mtp.printOptimalPath();
  return 0;
}
```

The `Matrix615.h` file uses `boost` library to load matrix from file. The file is available at `~hmkang/Public/615/include/Matrix615.h`

```cpp
#ifndef __MATRIX_615_H
#define __MATRIX_615_H
```

```cpp
#include <vector>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <boost/tokenizer.hpp>
#include <boost/lexical_cast.hpp>

template <class T>
class Matrix615 {
public:
  std::vector< std::vector<T> > data;

  Matrix615() {}

  Matrix615(int nrow, int col, T val = 0) {
    resize(nrow, col, val);
  }

  Matrix615(const char* fileName) {
    readFromFile(fileName);
  }

  void resize(int nrow, int ncol, T val = 0) {
    data.resize(nrow); // make n rows
    for(int i=0; i < nrow; ++i)
      data[i].resize(ncol,val); // make n cols with default value val
  }

  int rowNums() { return (int)data.size(); }

  int colNums() { return ( data.size() == 0 ) ? 0 : (int)data[0].size(); }

  void readFromFile(const char* fileName);
};

template <class T>
void Matrix615<T>::readFromFile(const char* fileName) {
  // open input file
  std::ifstream ifs(fileName);
  if ( ! ifs.is_open() ) {
    std::cerr << "Cannot open file " << fileName << std::endl;
    abort();
  }

  // set up the tokenizer
  std::string line;
  boost::char_separator<char> sep(" \t");
  // typedef is used to replace long type to a short alias
  typedef boost::tokenizer< boost::char_separator<char> > wsTokenizer;

  // clear the data first
  data.clear();
  int nr = 0, nc = 0;

  // read from file to fill the contents
  while( std::getline(ifs, line) ) {
    if ( line[0] == '#' ) continue; // skip meta-lines starting with #
    wsTokenizer t(line,sep);
    data.resize(nr+1);
```

```
    for(wsTokenizer::iterator i=t.begin(); i != t.end(); ++i) {
      data[nr].push_back(boost::lexical_cast<T>(i->c_str()));
      if ( nr == 0 ) ++nc;  // count # of columns at the first row
    }
    if ( nc != (int)data[nr].size() ) {
      std::cerr << "The input file is not rectangle at line " << nr << std::endl;
      abort();
    }
    ++nr;
  }
}
#endif
```

What you need to do is to fill `MTP::printOptimalPath(int r, int c)` function in `MTP.h` below.

```
#ifndef __MTP_H__
#define __MTP_H__

#include <iostream>
#include <cstdlib>
#include "Matrix615.h"

class MTP {
 public:
  Matrix615<int> cost;
  Matrix615<int> move;
  Matrix615<int> hw;
  Matrix615<int> vw;

  MTP(const char* hwf, const char* vwf);

  int computeOptimalCost();
  int computeOptimalCost(int r, int c);

  void printOptimalPath();
  void printOptimalPath(int r, int c);

  int computeOptimalCost2();
};

MTP::MTP(const char* hwf, const char* vwf) {
  hw.readFromFile(hwf);

  int nr = hw.rowNums();
  int nc = hw.colNums()+1;

  vw.readFromFile(vwf);

  if ( ( vw.rowNums() != nr-1 ) || ( vw.colNums() != nc ) ) {
    std::cerr << "ERROR: " << hwf << " and " << vwf << " are not compatible" << std::endl;
    std::cerr << nr << " " << nc << std::endl;
    abort();
  }

  cost.resize(nr, nc, -1);
  move.resize(nr, nc, -1);
}

int MTP::computeOptimalCost() {
  return computeOptimalCost(cost.rowNums()-1, cost.colNums()-1);
```

```
}

int MTP::computeOptimalCost(int r, int c) {
  // if cost is stored already, skip the cost evaluation
  if ( cost.data[r][c] < 0 ) {
    if ( ( r == 0 ) && ( c == 0 ) ) cost.data[r][c] = 0; // terminal condition
    else if ( r == 0 ) { // only horizontal move is possible
      move.data[r][c] = 0; // 0 means horitontal move to (r,c)
      cost.data[r][c] = computeOptimalCost(r,c-1) + hw.data[r][c-1];
    }
    else if ( c == 0 ) { // only vertical move is possible
      move.data[r][c] = 1; // 1 means vertical move to (r,c)
      cost.data[r][c] = computeOptimalCost(r-1,c) + vw.data[r-1][c];
    }
    else { // evaluate the cumulative cost of horizontal and vertical move
      int hcost = computeOptimalCost(r,c-1) + hw.data[r][c-1];
      int vcost = computeOptimalCost(r-1,c) + vw.data[r-1][c];
      if ( hcost > vcost ) { // when vertical move is optimal
        move.data[r][c] = 1; // store the decision
        cost.data[r][c] = vcost; // and store the optimal cost
      }
      else {
        move.data[r][c] = 0;
        cost.data[r][c] = hcost;
      }
    }
  }
  return cost.data[r][c];
}

int MTP::computeOptimalCost2() {
  // NEED TO FILL IN FOR PART (b)
}

void MTP::printOptimalPath() {
  printOptimalPath(cost.rowNums()-1, cost.colNums()-1);
}

void MTP::printOptimalPath(int r, int c) {
  // NEED TO FILL IN FOR PART (a)
}
#endif
```

(b) (25 pts). Write a function `MTP::computeOptimalCost2()` that fills `move` and `cost` matrix as `MTP::computeOptimalCost()` does, but without using recursion. You can accomplish the same thing just using a nested loop. After implementing `MTP::computeOptimalCost2()`, replace `MTP::computeOptimalCost()` in the `main()` function into `MTP::computeOptimalCost()`, and confirm that the output is identical.

Problem 3. Hidden Markov Model

(a) (25 pts) Write a C++ program that performs a general Hidden Markov Model inference, from any transition and emission matrix, initial state distribution, and sequence of observed output. All inputs are whitespace or tab-delimited matrices. For HMM with $|S|$ states, $|O|$ outputs, and T time slots, the transition matrix should be $|S| \times |S|$ matrix of `double`, emission matrix should be $|S| \times |O|$ matrix of `double`, pi should be $|S| \times 1$ matrix (or vector) of `double`, and outputs should be $T \times 1$ matrix of `int`.

For example, the input parameters for `biasedCoin` example can be rewritten as

Transition matrix (`trans2.txt`):

```
0.95 0.05
0.20 0.80
```

Emission Matrix (`emis2.txt`):

```
0.5 0.5
0.9 0.1
```

Priors (`pis2.txt`):

```
0.5
0.5
```

```
$ ./hw-3-3 trans2.txt emis2.txt pis2.txt ~hmkang/Public/615/data/obs.20.txt
TIME    TOSS    Pr(0)   Pr(1)   MLSTATE
1       0       0.5950  0.4050  0
2       1       0.8118  0.1882  0
3       0       0.8071  0.1929  0
4       1       0.8584  0.1416  0
5       0       0.7613  0.2387  0
6       0       0.7276  0.2724  0
7       1       0.7495  0.2505  0
8       0       0.5413  0.4587  1
9       0       0.4187  0.5813  1
10      0       0.3533  0.6467  1
11      0       0.3301  0.6699  1
12      0       0.3436  0.6564  1
13      0       0.3971  0.6029  1
14      1       0.5028  0.4972  1
15      0       0.3725  0.6275  1
16      0       0.2985  0.7015  1
17      0       0.2635  0.7365  1
18      0       0.2596  0.7404  1
19      0       0.2858  0.7142  1
20      0       0.3482  0.6518  1
```

Below is another example run with 3 states.

Transition matrix:

```
0.98 0.01 0.01
0.01 0.98 0.01
0.01 0.01 0.98
```

Emission matrix:

```
0.5 0.5
0.9 0.1
0.1 0.9
```

The priors are

```
0.50
```

```
0.25
0.25
```

And the expected output is

```
$ ./hw-3-3 trans3.txt emis3.txt pis3.txt ~hmkang/Public/615/data/obs.30.txt
TIME    TOSS    Pr(0)   Pr(1)   Pr(2)   MLSTATE
1       1       0.1982  0.0014  0.8004  2
2       1       0.1911  0.0003  0.8086  2
3       1       0.1891  0.0003  0.8106  2
4       1       0.1915  0.0014  0.8071  2
5       1       0.1982  0.0103  0.7915  2
6       1       0.2065  0.0850  0.7085  2
7       0       0.1887  0.7307  0.0806  1
8       0       0.1666  0.8240  0.0094  1
9       0       0.1529  0.8459  0.0011  1
10      0       0.1460  0.8538  0.0002  1
11      0       0.1439  0.8561  0.0000  1
12      0       0.1458  0.8541  0.0000  1
13      0       0.1526  0.8473  0.0000  1
14      0       0.1666  0.8333  0.0001  1
15      0       0.1929  0.8070  0.0001  1
16      0       0.2407  0.7590  0.0002  1
17      0       0.3271  0.6719  0.0010  1
18      0       0.4827  0.5105  0.0068  1
19      1       0.7619  0.1807  0.0574  0
20      1       0.8228  0.1201  0.0571  0
21      1       0.8445  0.1105  0.0450  0
22      0       0.8639  0.1228  0.0133  0
23      0       0.8704  0.1215  0.0081  0
24      0       0.8779  0.1136  0.0085  0
25      1       0.8894  0.0924  0.0181  0
26      0       0.8888  0.0916  0.0196  0
27      0       0.8839  0.0812  0.0349  0
28      1       0.8460  0.0165  0.1375  0
29      1       0.8213  0.0088  0.1699  0
30      1       0.8062  0.0096  0.1843  0
```

The `Matrix615.h` is the same as the Problem 2, and `HMM615.h` is implemented as follows (which can be found at `hmkang/Public/615/include/HMM615.h`

```
#ifndef __HMM_615_H
#define __HMM_615_H

#include "Matrix615.h"

class HMM615 {
 public:
  // parameters
  int nStates;  // n : number of possible states
  int nObs;     // m : number of possible output values
  int nTimes;   // t : number of time slots with observations
  std::vector<double> pis; // initial states
  std::vector<int> outs;   // observed outcomes
  Matrix615<double> trans;   // trans[i][j] corresponds to A_{ij}
  Matrix615<double> emis;

  // storages for dynamic programming
  Matrix615<double> alphas;
  Matrix615<double> betas;
  Matrix615<double> gammas;
```

```
 Matrix615<double> deltas;
 Matrix615<int> phis;
 std::vector<int> path;

HMM615(int states, int obs, int times) : nStates(states), nObs(obs), nTimes(times),
   trans(states, states, 0), emis(states, obs, 0), alphas(times, states, 0), betas(times, states, 0),
   gammas(times, states, 0), deltas(times, states, 0), phis(times, states, 0)
 {
   pis.resize(nStates);
   path.resize(nTimes);
 }

 void forward() {
   for(int i=0; i < nStates; ++i) {
     alphas.data[0][i] = pis[i] * emis.data[i][outs[0]];
   }
   for(int t=1; t < nTimes; ++t) {
     for(int i=0; i < nStates; ++i) {
       alphas.data[t][i] = 0;
       for(int j=0; j < nStates; ++j) {
         alphas.data[t][i] += (alphas.data[t-1][j] * trans.data[j][i] * emis.data[i][outs[t]]);
       }
     }
   }
 }

 void backward() {
   for(int i=0; i < nStates; ++i) {
     betas.data[nTimes-1][i] = 1;
   }
   for(int t=nTimes-2; t >=0; --t) {
     for(int i=0; i < nStates; ++i) {
       betas.data[t][i] = 0;
       for(int j=0; j < nStates; ++j) {
         betas.data[t][i] += (betas.data[t+1][j] * trans.data[i][j] * emis.data[j][outs[t+1]]);
       }
     }
   }
 }

 void forwardBackward() {
   forward();
   backward();

   for(int t=0; t < nTimes; ++t) {
     double sum = 0;
     for(int i=0; i < nStates; ++i) {
       sum += (alphas.data[t][i] * betas.data[t][i]);
     }
     for(int i=0; i < nStates; ++i) {
       gammas.data[t][i] = (alphas.data[t][i] * betas.data[t][i])/sum;
     }
   }
 }

 void viterbi() {
   for(int i=0; i < nStates; ++i) {
     deltas.data[0][i] = pis[i] * emis.data[i][ outs[0] ];
   }
   for(int t=1; t < nTimes; ++t) {
```

```
      for(int i=0; i < nStates; ++i) {
        int maxIdx = 0;
        double maxVal = deltas.data[t-1][0] * trans.data[0][i] * emis.data[i][ outs[t] ];
        for(int j=1; j < nStates; ++j) {
          double val = deltas.data[t-1][j] * trans.data[j][i] * emis.data[i][ outs[t] ];
          if ( val > maxVal ) {
            maxIdx = j;
            maxVal = val;
          }
        }
        deltas.data[t][i] = maxVal;
        phis.data[t][i] = maxIdx;
      }
    }

    // backtrack viterbi path
    double maxDelta = deltas.data[nTimes-1][0];
    path[nTimes-1] = 0;
    for(int i=1; i < nStates; ++i) {
      if ( maxDelta < deltas.data[nTimes-1][i] ) {
        maxDelta = deltas.data[nTimes-1][i];
        path[nTimes-1] = i;
      }
    }
    for(int t=nTimes-2; t >= 0; --t) {
      path[t] = phis.data[t+1][ path[t+1] ];
    }
  }
};
#endif // __HMM_615_H
```

(b) (25 pts) If you run this program for a very large input (e.g. 20,000), you will notice that the inference is incorrect due to numerical precision issue. Below is an example you can try.

```
$ ./hw-3-3 trans2.txt emis2.txt pis2.txt ~hmkang/Public/615/data/obs.20k.txt
```

Modify `HMM615.h` to avoid the limitation of numerical precision. (Hint: in Viterbi algorithm, storing log-likelihood instead of likelihood will solve the problem. In forward-backward algorithm, you don't need to use log transformation. A simple normalization of `alpha` and `beta` at each time slot will be sufficient, and more efficient).