CMPT 435 - Fall 2020 - Dr. Labouseur

Semester Project
Pooled Infection Testing Simulator

Sam Alcosser
Samuel.Alcosser1@Marist.edu

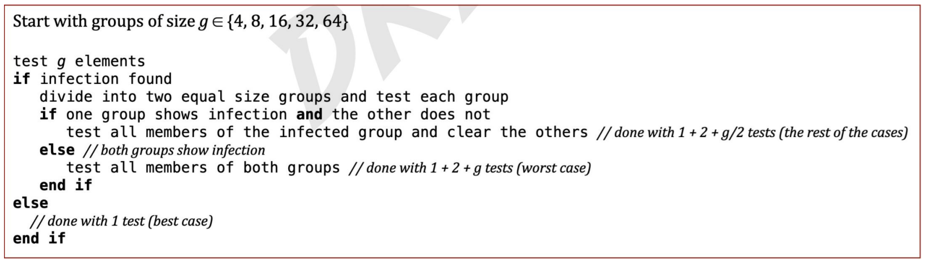
December 9, 2020

Contents

1 Introduction

1.1 History and Background

With current events in late 2020, we are currently past the point where a test for the novel disease is hard to come by. Although despite this, there is something to be said for the efficiency of the tests. As described by Dr. Alan Labouseur, during the time of WWII, there was a need for a more efficent way to test for infections. One idea was pooled testing. With this approach, instead of handing out individual tests, a set of people would all be swabbed or sampled, all of the samples would be compiled into one testable container, and the container would be tested as a whole. If the test comes back negative, everyone is assumed to be clear, and if not, the sample set is split into two sub samples and repeat the process. After a certain point, the testing facility will decide to go to individual tests to ensure who is actually positve. With this approach, instead of using 1000 tests on 1000 people, the number of tests can sometimes be reduced down to under 250 for the same sample size. Dr. Labouseur's pseudo-code algorithm for this process can be seen below.



```
Start with groups of size $g \in \{4, 8, 16, 32, 64\}$

test $g$ elements
if infection found
   divide into two equal size groups and test each group
   if one group shows infection and the other does not
      test all members of the infected group and clear the others // done with 1 + 2 + g/2 tests (the rest of the cases)
   else // both groups show infection
      test all members of both groups // done with 1 + 2 + g tests (worst case)
   end if
else
  // done with 1 test (best case)
end if
```

1.2 Implementation and Logic Design

To simulate this situation, I needed three main parts.

- A way to set up the sample size with a simulated infection rate, and put them into pools of 8
- A way to implement the pooled testing algorithm on each pool
- Some way to abstract the use of both of the other parts

Also, A very simple UI was created to assist the user in running the simulations. To see how this works, look to the Appendix subsection titled "`main.cpp` : Making The Simulator Usable".

2 Implementation

2.1 `PooledTesting::setupPools()` : Setup

This first function gets called at the beginning of the process. The function takes a population size and a percentage of infection, randomly infects the correct number of people, and loads up the `testPools` vector with sets of eight people.

```cpp
void PooledTesting::setupPools(double size, double pcPos)
{

    tPeople = size;
    std::vector<int> population(tPeople, 0);
    int numPos = (int)(static_cast<double>(tPeople) * pcPos);

    std::vector<int> indexes;
    int chosen = 0;
    while (chosen < numPos)
    {

        std::random_device dev;
        std::mt19937 rng(dev());
        std::uniform_int_distribution<std::mt19937::result_type> dist6(0, (tPeople - 1));
        int attempt = dist6(rng);
        if (std::find(indexes.begin(), indexes.end(), attempt) != indexes.end())
        {
            continue;
        }
        else
        {

            indexes.push_back(attempt);
            population[attempt] = 1;
            chosen++;
        }
    }

    int remainder = tPeople % poolSize;

    int pools = (tPeople - remainder) / poolSize;

    for (int i = 0; i < pools; i++)
    {
        std::vector<int> pool;
        for (int j = 0; j < poolSize; j++)
        {
            int subInd = (i * poolSize) + j;
            pool.push_back(population[subInd]);
        }
        testPools.push_back(pool);
43      }
44      if (remainder > 0)
45      {
46          int processedCounter = pools * poolSize;
47          std::vector<int> remVect;
48          for (int x = processedCounter - 1; x < tPeople; x++)
49          {
50              remVect.push_back(population[x]);
51          }
52          testPools.push_back(remVect);
53      }
54      std::cout << testPools.size() << " pools have been set up." << std::endl;
55  }
```

1. (lines 4-9): Setup the local variables for this method including:
   (a) `tPeople` for the size of the population
   (b) `population` for set of all people in the population in a vector of ints initialized with the population size `tPeople` each as 0, to indicate starting as negative
   (c) `numPos` for the computed amount of positive cases based on population and the infection rate `pcPos` which was passed in
   (d) `indexes` for the indexes of the population set which are randomly decided to be positive (also useful to ensure that there are no repeats)
   (e) `chosen` to keep track of how many indexes have been chosen
2. While the number of chosen indexes is less than the number of needed positive cases (line 10), create random numbers in the range of the number of people (lines 13-16) and if the number has not been previously chosen, (line 17), add it to the list of selected indexes and mark that index as chosen. Each of the ”people” at these indexes will be marked positive by a ”1” for their value instead of a ”0”. (lines 24-26)
3. Decide the number of needed pools by first stripping off the people that would create an uneven split (line 30), and then divide the rest of the people by the pool size (line 32).
4. Create a vector and populate it with 8 ”people” of the whole population size an amount of times equivilent to the number of necessary pools.
   (a) Start by making an iterator that will run from 0 to right under the number of desired pools (line 34).
   (b) Make a vector of ints to hold the current working pool, and fill it with the next eight ”people's” values using another iterator from 0 to 7 (line 37). To find the correct index, multiply the index of the pool `i`, and multiply it by `poolSize` to account for the previous pools values.

    Then, add the value of the iterator for the person in the current pool `j`, and use that value to take the correct index, and push it onto the current pool (lines 39,40).

    (c) Once the pool is created, push it onto the vector of pools `testPools` (line 42).

5. Don't forget about the remaining people that couldn't fit into a complete pool. Take them and put them into one final test pool.

    (a) Start a counter for the correct index by computing the previously processed indexes by setting `processedCounter` to `pools * poolSize`(line 46)

    (b) Fill the final pool vector `remVect` by starting at the index right below the number of processed people, working until the last index (lines 47-51)

    (c) Push the final pool vector to `testPools` and indicate to the user that the pools have been set up. (lines 52,54)

2.2 `PooledTesting::itTest` : Testing the Pools

Now that we have our pools set up, it is time to test the pools for positive cases. Following the algorithm, If a case comes back as positve, it will be "recursively" split until the exact infected person(s) is/are found.

I say recursive in quotes, as my implementation is not recursive, but operates as if it is. This is done because in this simulation, we are using a fixed pool size of 8. This way, we know that 8 will always split into two groups of four, and then from there there would be individual tests. If there was the intention of having varying sizes of pools, recursion could be used to easily split the test sizes depending on the pool size. Although again, this was built with the intention of a fixed size of eight.

This function knows which pool to operate on as it is passed `sPoolIndex` as an index within the vector that holds all of the pools.

Note: Any time an if statement with the condition of `readOut` is seen, this means that the contained print statement will only execute if the user states that they want full print out in their console window. This is seen throughout.

```
1  void PooledTesting::itTest(int sPoolIndex, bool readOut)
2  {
3      int poolCases = 0;
4      int poolTests = 0;
5
6      bool initPos = false;
7      testCounter++;
8      poolTests++;
9      for (int i = 0; i < poolSize; i++)
10     {
        if (testPools[sPoolIndex][i] == 1)
        {
            initPos = true;

        }
    }
    if (initPos)
    {
        bool s1 = false;
        bool s2 = false;
        testCounter++;
        poolTests++;
        for (int j = 0; j < 4; j++)
        {

            if (testPools[sPoolIndex][j] == 1)
            {
                s1 = true;

            }
        }
        testCounter++;
        poolTests++;
        for (int k = 4; k < poolSize; k++)
        {

            if (testPools[sPoolIndex][k] == 1)
            {
                s2 = true;

            }
        }
        if (s1)
        {
            for (int n = 0; n < 4; n++)
            {
                testCounter++;
                poolTests++;
                if (testPools[sPoolIndex][n] == 1)
                {
                    activeCases++;
                    poolCases++;
                    if (readOut)
                    {
                        std::cout << "Positive case found in pool #" << sPoolIndex << " for person " << n << "."
                    }
                }
            }
        }
```

```
61          if (s2)
62          {
63              for (int m = 4; m < poolSize; m++)
64              {
65                  testCounter++;
66                  poolTests++;
67                  if (testPools[sPoolIndex][m] == 1)
68                  {
69                      activeCases++;
70                      poolCases++;
71                      if (readOut)
72                      {
73                          std::cout << "Positive case found in pool #" << sPoolIndex << " for person " << m << "."
74                      }
75                  }
76              }
77          }
78          if (readOut)
79          {
80              std::cout << "POOL #" << sPoolIndex << ":Tests used:" << poolTests << std::endl;
81              std::cout << "POOL #" << sPoolIndex << ":Cases found:" << poolCases << std::endl;
82          }
83      }
84      else
85      {
86          if (readOut)
87          {
88              std::cout << "Pool #" << sPoolIndex << " came back negative." << std::endl;
89              std::cout << "POOL #" << sPoolIndex << ":Tests used:" << poolTests << std::endl;
90              std::cout << "POOL #" << sPoolIndex << ":Cases found:" << poolCases << std::endl;
91          }
92
93  }
94  }
```

As this is an iterative approach to this process, much of the code is mirrored between both halves of the set, and the descriptions will be grouped as such.

1. Start by Initializing variables to hold the `poolCases` for the cases for logging, `poolTests` to hold the tests that were used for the single pool. Again, these are only used for logging. The real counts are held within the `PooledTesting` object itself.(lines 3,4).

2. Initialize a flag for a positive pool as `false` (line 6).

3. Complete the initial whole pool test.

   (a) Increment the local `poolTests` counter and the object's `testCounter` variable. (lines 7-8)

   (b) Test each ”person” in the pool, and if they are positive, raise the `initPos` flag. (lines 12-17)

4. Initialize flags for both halves of the pool called `s1` and `s2` respectivley, and set them to false (lines 20,21)
5. For each sub pool:
    (a) Increment the local `poolTests` counter and the object's `testCounter` variable. (lines 22-23, 33-34)
    (b) Test each "person" in the sub pool, and if they are positive, raise the `s1` or `s2` flag depending on which sub pool. (lines 24-32, 35-43)
6. If either or both sub pools `s1` or `s2` come back positive, as indicated by their flags:
    (a) Iterate over each "person" in the sub pool. At the beginnning of each iteration, increment the local `poolTests` counter and the object's `testCounter` variable. (lines 48-49, 65-66)
    (b) Test each "person" in the sub pool, and if they are positive, increment both the local variable `poolCases` and the object's attribute `activeCases`. (lines 52-53, 69-70)
7. in the (not so) rare case that the whole pool comes back negative, do nothing, or output the results to the user if they desire. (lines 84-93)

2.3 `PooledTesting::testThePools` : Simplifying the Run Process

At this point we have a way to setup all of the population, put them into pools, and test each pool. Although If this were to be the ending point, much of the organizational work would be left to the `main()` function, which is less than desirable. For this reason, `testThePools` was written to simplify the process.

```cpp
void PooledTesting::testThePools(int sampleSize, double perCent, bool readOut)
{
    setupPools(sampleSize, perCent);
    for (int i = 0; i < testPools.size(); i++)
    {

        itTest(i, readOut);
        if (readOut)
        {
            std::cout << "_____________________________" << std::endl;
            std::cout << "Beginning testing on pool #" << i + 1 << std::endl;

            std::cout << "Testing for pool #" << i + 1 << " is complete." << std::endl;
            std::cout << "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" << std::endl;
        }
    }
    std::cout << "Total cases found:" << activeCases << std::endl;
    std::cout << "Observed infection percentage:" <<std::to_string(((double)activeCases / (double)sampleSize) *
19
20      std::cout << "Total tests used:" << testCounter << std::endl;
21      activeCases = 0;
22      testCounter = 0;
23      tPeople = 1000;
24      testPools.clear();
25      std::cout << "Data has been reset." << std::endl;
26  }
```

1. Run the `setupPools()` method using the `sampleSize` and `perCent` positive as passed into the method from `main()`(line 4).

2. Being that `setupPools` sets the number of pools, we can use that as a limit for our iterator. For each pool, run `itTest()`. Pass the function the value of the iterator `i` which will operate as an index for which pool to operate on. Also, pass `readOut` to alert the method whether or not to give detailed logs. (lines 4-7)

3. If the user desires full read outs, print out some information about the current state of the processing. (lines 10 - 14)

4. At the end of the processing, print out the total `activeCases` found, the computed observed infection percentage, and `testCounter`, the number of tests used. To find the observed infection percentage divide `activeCases` by `sampleSize`, and multiply by 100.0 to get a percentage (This was cut off by the page). Make sure to type cast both numbers to doubles so that a decimal value is given. (lines 17-20)

5. Reset `activeCases` and `testCounter` to 0, and set `tPeople` to 1000.(lines 21-23)

6. Finally, clear the `testPools` vector, and inform the user that the data has been reset. (lines 24,25)

3 Results

Here are the results running the simulation with different settings.

Population Size	Infection %	Found Cases	Observed %	Tests Used
125	85%	107	85.6%	176
125	14.96%	19	15.2%	96
125	.04%	0	0.0%	16
1000	2%	20	2%	243
10000	2%	200	2%	2394
100000	2%	2000	2%	24030
1000000	2%	20000	2%	240112

4 Appendix

4.1 `main.cpp` : Making The Simulator Usable

It should go without saying that the nature of this project is different than all of the previous projects within Algorithms Analysis and Design. Other projects would focus on demonstrating a working knowledge of how a data structrure or algorithm works, by implementing it and creating a small demo. This project however, could be seen as a full program. Because of this, it would not be complete to just create a class that has the functionality to do a single demo, or even swap out input files. Instead, it is more fitting to have a more polished way to interact with the program.

For this reason, Alongside the logic of the simulation, there was a very simple user interface to aid in utilizing the simulation that was created. This makes it possible to both do quick tests, such as `.launch quick 1000 .02` which will run a simulation of 1000 people at an infection percentage of 2%, and simply return the cases found, observed infection percentage, and tests used. With the simple cli accessed by running `./launch cli` after compiling, the user is guided through setting up their simulations, and can run multiple simulations in succession. For those who want to have all of the logging, but to only do one simulation, they can run `./launch quick 1000 .02 printAll` and simulate 1000 people with a positivity of 2%, with full print out.

Example Run Configurations

- `./launch quick 1000 .02`
  - run a single simulation of 1000 people with a 2% random positivity
- `./launch quick 1000 .02 printAll`
  - run a single simulation of 1000 people with a 2% random positivity, and print out the results of all of the simulated tests
- `./launch cli`
  - run the simulator with the command line interface

```
int main(int argc, char **argv)
{
    std::string type = std::string(argv[1]);
    bool readOut = false;
    if( argc == 5 && std::string(argv[4]) == "printAll"){
        readOut = true;
    }
    if (type == "quick")
    {
        PooledTesting *pt = new PooledTesting();
        pt->testThePools(std::stod(argv[2]), std::stod(argv[3]), readOut);

    }

    else if (type == "cli")
    {



        int sampSize = 0;
        double pcPos = 0;
        bool running = true;

        std::cout << "****************************************" << std::endl;
        std::cout << "POOLED TESTING SIMULATOR" << std::endl;
        std::cout << "Welcome to the Pooled Testing Simulator!" << std::endl;

        while (running)
        {
            bool havePercent = false;
            std::cout << "****************************************" << std::endl;
            std::cout << " either type 'run' follwed by a single space and your total case size, or type 'quit' t
            std::cout << "Example: 'run 1000' or 'quit'" << std::endl;
            std::string command;

            std::getline(std::cin, command);
            if (command == "quit")
            {
                std::cout << "Goodbye." << std::endl;
                running = false;
            }
            else if (command.find("run") != std::string::npos && (command.substr(4).find_first_not_of("012345678
            {
                sampSize = std::stoi(command.substr(4));
                std::cout << sampSize << " Was your defined sample size." << std::endl;

                while (!havePercent)
                {
                    std::cout << "What is your simulated percentage of positive cases as a decimal number (i.e. 
                    std::string preProcessedPer;
                    std::getline(std::cin, preProcessedPer);
                    if (preProcessedPer.find("quit") != std::string::npos)
                    {
                        std::cout << "Goodbye." << std::endl;
                        running = false;
                        break;
                    }
                    else if ((preProcessedPer.find_first_not_of("0123456789.") == std::string::npos) && preProce
                    {
                        if (std::stod(preProcessedPer) <= 1)
                        {
                            pcPos = std::stod(preProcessedPer);

```

```
64                              PooledTesting *pt = new PooledTesting(); //actually doing the tests
65                              pt->testThePools(sampSize, pcPos, true);
66
67                              std::cout << "completed test of " << sampSize << " people with an overall positivity
68
69                              havePercent = true;
70                          }
71                          else
72                          {
73                              std::cout << "ERROR: Please only use positive decimal numbers between 0 and 1 (i.e.
74                              std::cout << "---------------------------------------------------------" << std::endl
75                          }
76                      }
77                      else
78                      {
79                          std::cout << "ERROR: Please only use positive decimal numbers between 0 and 1 (i.e. 3% w
80                          std::cout << "---------------------------------------------------------" << std::endl;
81                      }
82                  }
83              }
84              else if (running)
85              {
86
87                  std::cout << "ERROR: Sorry, that command was not recognized.\a" << std::endl;
88              }
89          }
90      }
91
92      return 0;
93  }
```

Due to the nonlinear nature of this function, a list of the actions of the functions is listed below in detail.

- Quickly allow for a ”quick” run to simply take the console inputs and plug them into a simulation. This is usable, as it is only for a quick run of the test.(lines 8-13)
- Handle whether or not the user wants full print out when they input ”printAll” as an argument. (lines 5-7)
- If the user asks to quit the cli by writing `quit`, stop the program (lines 37-41, 52-57)
- If the user desires to use the cli, as indicated by their input (line 15)
  1. Start a while loop to run until indicated that `running` has been set to false (line 28)
  2. Have the user input a desired population. (lines 31-33)

3. If the input was in the correct format (run [population size]) (line 42), ask for the user to give a positivity percentage. (lines 49-51)
4. Keep asking for a percentage until their input matches the desired format of a decimal number from 0-1. (lines 58-69, 58, , caught on lines 71-75, 77-81)
5. If the values were both correct, run the simulation with those inputs, and mark the flag denoting that a correct percentage has been inputted. (lines 64, 65, 69)

- If anything does not work in the outside levels of the nesting (failure to use the run command) print an error. (lines 84-88)

4.2 `PooledTesting.h` The Pooled Testing Class

```cpp
#pragma once
#include <iostream>
#include <vector>
class PooledTesting
{
public:
    int tPeople = 0;
    const int poolSize = 8;
    int activeCases = 0;
    int testCounter = 0;
    void setupPools(double size, double pcPos);
    void testThePools(int sampleSize, double perCent, bool printOut);
    void itTest(int sPoolIndex, bool readOut);
    std::vector<std::vector<int>> testPools;
};
```

- `tPeople` holds the population size
- `poolSize` holds the size of the pools as a constant
- `activeCases` holds the current active cases as observed
- `testCounter` holds a counter of the amount of tests used
- `setupPools()` is used to initialize the pools with randomly assigned active cases
- `testThePools()` is used to implement the testing function `itTest()`
- `itTest()` tests the pool which it is designated to
- `testPools` holds a two dimensional vector of vectors, the inner vectors being the pools

Closing Thoughts

This final project was overall very easy, yet enjoyable for a different reason. The fact that we were able to see a real world application of computer science (beyond just a linked list or even a website) was extremely interesting. This class overall was so helpful to my understanding of concepts that at one time I thought it would take years to learn. With this class, scary sounding topics like binary search trees sound easy, and completely approachable. These documents that I have created over the course of this semester will surely be of great help later on in my progression.

as a wise person once said, " give a person a program and they'll be frustrated for a day, teach a person to program and they'll be frustrated for the rest of their life.". $\{$*S.A.*$\}$

References

Labouseur, Alan G. *Lots of Group Testing and a Little Computer Science Can Improve COVID-19 Screening* .