CMPT 435 - Fall 2021 - Dr. Labouseur

Semester Project - Covid Pool Testing Simulation

Augusto Gonzalez-Bonorino
augusto.gonzalezbonorino1@marist.edu

December 15, 2021

1 Introduction

In 2020, the world was struck by a highly contagious virus: SARS COVID-19. Infections started rising by the hundreds, then by thousands. Hospitals were at full capacity, people were being laid off and businesses were going bankrupt. The increasing contagion rate required thousands and thousands of tests every day to keep track of Covid's evolution. Clearly, the method was not working at scale. Enter pool testing...

Pool testing is a testing method where a sample of a group (i.e. pool) of individuals is obtained and tested. If the test comes back positive we have no clue who of the individuals in the group is infected but we can proceed to a second testing phase, in which the original group is divided into k subgroups and tested separately. Now, what is the advantage then? you may ask. Well, if the first test comes back negative we know none of the individuals is infected and we have save ourselves a whole bunch of tests, and therefore valuable time and money. This is the objective of the project described in this documentation: Simulate a simple pool testing. Let's get into it!

2 Specifics of our simulation

For our purposes we have simplified the real pool-testing protocol to be able to program it with the limited knowledge of statistics and computer science that we possess. We will consider four experiments (different sizes of population), with pools of eight people, assuming 100% test accuracy and a fixed infection rate of 2%. So, we need a method for infecting people given a population and the infection rate. A method to group the population into groups of eight and a method to check if someone in a given group is infected. Here is how I implemented these methods in Java:

```java
public static ArrayList<Integer> infectPopulation(int popSize)
    {

      // to infect random people
      Random rand = new Random();

      // set to store unique ids
      Set<Integer> idxs = new LinkedHashSet<Integer>();

      // list to populate the full population with 1s and 0s
          ArrayList<Integer> patients = new ArrayList<Integer>();

          for (int i = 0; i < popSize; i++)
        {
              patients.add(0);

        } // for

          int idx = 0;

      while (idxs.size() < patients.size() * infectionRate)
        {
            idx = rand.nextInt(patients.size());

            idxs.add(idx);

        } // for


      for (int id : idxs)
        {

            // set infected patient to 1
            patients.set(id, 1);

        } // for each unique random id in idxs

          return patients;

      } // infectPopulation
```

First, let's see how we go about infecting people. I promise these are not real people. or are they?... Anyways, we do not get to play God for now. The method takes in a population size and randomly infects 2% of the individuals. I achieve this by generating random indices with the help of the *Random* package. Then, we populate an *ArrayList* of size equal to the given population size and set those randomly infected indices to 1 (to represent an infection). Otherwise, uninfected people are represented by 0s. Finally, we return the full list of patients. Next, we need to group them. Here is my function for that:

```java
public static ArrayList<ArrayList<Integer>> groupBy8(ArrayList<
    Integer> patients)
      {
            ArrayList<ArrayList<Integer>> grouped_pop = new ArrayList
    <>();
            ArrayList<Integer> group8 = new ArrayList<>();

            for(int i = 0; i < patients.size(); i++)
              {

                  if (group8.size() == 7)
                    {
                        // add an 8th person
                        group8.add(patients.get(i));

                        // clone group8
                        @SuppressWarnings("unchecked")
                ArrayList<Integer> group8_temp = (ArrayList<Integer>)
     group8.clone();

                        // add it to the ArrayList of ArrayLists
                        grouped_pop.add(group8_temp);

                        // clear group8 to add more people
                        group8.clear();

                    } // if it is a group of 7

                  else
                    {

                        group8.add(patients.get(i));

                    } // else

              } // for every patient

            return grouped_pop;
```

Given the array containing the patients generated with *infectPopulation* this method returns an *ArrayList* of *ArrayList* containing the grouped population. The method is straightforward and thus I will not dive into much detail. First, we instantiate a temporary *ArrayList* to store the groups of eight people as we traverse the patients and an *ArrayList* of *ArrayList* which contains the grouped population to be returned. Last but not least we define the method to check for an infection on a given group as follows:

```java
public static String checkInfection(ArrayList<Integer> group8)
    {

      String caseObserved = " ";

        if(!group8.contains(1))
        {
            // Case 1
          caseObserved = "Case 1";

        } // if no one is infected

        else if(group8.contains(1))
        {

          try
            {
                // split in half for first pool
                ArrayList<Integer> subGroup_1 = new ArrayList<Integer> (group8.subList(0, 4));
                ArrayList<Integer> subGroup_2 = new ArrayList<Integer> (group8.subList(4, 8));


                if(subGroup_1.contains(1) & subGroup_2.contains(1))
                {
                    //it is case 3
                  caseObserved = "Case 3";

                } // if both group have an infected person


                else if(subGroup_1.contains(1) & !subGroup_2.contains(1) || (!subGroup_1.contains(1) & subGroup_2.contains(1)))
                {
                  caseObserved = "Case 2";

                } // else if only one of the groups has an infected person

            } // try

          catch (IndexOutOfBoundsException ex)
            {
                  System.out.println("Exception thrown : " + ex);

              } // check index not out of bounds

          catch (IllegalArgumentException ex)
            {
                  System.out.println("Exception thrown : " + ex);

              } // check arguments are not illegal

        } // else if there is an infected person

        return caseObserved;
```

We have three possible cases: no infections, only one infection or more than one infection. I have conveniently denoted them as case 1, 2 and 3 respectively in order to keep track of the number of cases (which we will use to compute the number of tests needed). Very well, to check for an infection we simply need to check if the given list (group of 8) contains a 1 or not. First, we check for the case 1. Then, if we find an infection we must split the list in half, meaning two subgroups of 4 which I have denoted (again very conveniently) *subGroup_1* and *subGroup_2*. Then, if both groups contain a 1 then we have a case 3, else if only one of the subgroups has an infected patient we have a case 2. Java makes checking for the existence of an element in a list very easy by the use of the *contains* method. Great! All there is left to do is simulate, which is conducted in *Main* in the following manner:

```java
public class MainGonzalezBonorino {

  // fixed infection rate
  final static double infectionRate = 0.02;

  // number of experiments to conduct
  final static int numExperiments = 4;

  public static void main(String[] args)
  {

    // array to hold different population sizes to test
    ArrayList<Integer> populationsSize = new ArrayList<Integer>();

    // size of first population
    int size = 1000;

    for (int i = 0; i < numExperiments; i++)
      {
        populationsSize.add(size);

        size = size * 10;

      } // for

    System.out.println("SEMESTER PROJECT - COVID POOL TESTING");

      for ( int j = 0; j < numExperiments; j++)
      {

        int popSize = populationsSize.get(j);

          ArrayList<Integer> population = infectPopulation(popSize)
    ;

          ArrayList<ArrayList<Integer>> grouped_pop = groupBy8(
    population);

      System.out.println(" ");

      System.out.println("************* TEST " + (j+1) + " 
    ***************");

          System.out.println("Population size " + popSize + " with 
    " + infectionRate + " infection rate");

      System.out.println(" ");


          // number of cases

          double case1 = 0.0;
          double case2 = 0.0;
          double case3 = 0.0;

          for(int i = 0; i < grouped_pop.size(); i++)
          {
              String cases = checkInfection(grouped_pop.get(i));
              if(cases.equals("Case 1"))
              {
                  case1++;

              } // case 1

              else if(cases.equals("Case 2"))
              {
                  case2++;

              } // case 2

              else if(cases.equals("Case 3"))
              {
                  case3++;

              } // case 3

          } // inner for

    //print cases and percentages
    System.out.println("Case (1): " + grouped_pop.size() + " x " +
    case1 / (population.size() / 8) + " = " +
    grouped_pop.size() *
    Math.round( (case1 / (population.size() / 8) * 100.0))
    / 100.0 + " instances requiring "
    + (int) (case1 * 1) + " tests");

      } // outer for

  } // main
```

We have a fixed infection rate of 2% and so I set it as a static global variable, same as the number of experiments we would like to simulate to simplify the code. Then, we initialize a population size of 1000 patients and use the aforementioned methods to infect, group and check iteratively. This process is repeated *numExperiments* times, each iteration increasing the size of the population by a factor of 10. As expected, the statistics get increasingly more accurate as the population size increases, in other words the simulation becomes ever more statistically significant. Right? Well, there is a little detail to consider which I will discuss in the next section. To conclude the specific of the simulation section note that all we do at the end is print out the results (instances of the specific case and the number of tests required) of each experiment. Note that I have only included the sample of code for case 1 in the listing with the hope of improving legibility, just keep in mind that the same exact process is conducted for the two remaining cases and then they are summed to compute the total number of tests required in the simulation. For the full code please refer to the GitHub repository.

3 Further Thoughts

There are many components of this implementation which we can improve upon: relaxing our assumption of 100% test accuracy and a fixed infection rate, determine the size of pools (groups) according to the population size since ours will break if the population size is not divisible by eight, among others. One interesting improvement to consider, at least interesting to me, is the probability distribution implemented. We base our simulation on a binomial distribution, sampling with replacement where the events are independent with same probability of success or failure, but what if we considered a hyper-geometric distribution instead where there is no replacement and the probability of success changes every time we draw a patient from the population? At large scale, meaning a very large population, the probabilities differ by an arbitrary amount since the probability of recording x infections from a hyper-geometric distribution approximates the probability of recording them from a binomial distribution, but at a low scale these may differ significantly. These are all very interesting variations of our simulation that would be worth considering in future work. But for now, this is all folks. It has been an extremely productive semester full of knowledge and learning experiences. I look forward to improving my software development skills and thus improve upon the plethora of algorithms and data structures we learnt these past four months.