Algorithms and Datastructures
Assignment 1 — COVID-19 Group Testing Report

Wouter Doeland — s1034816
Willem Medendorp — s1040947
Vamsi Yerramsetti — s1032599

November 14, 2020

1 Introduction

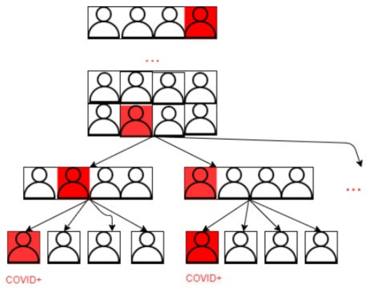
The Corona Virus has devastated the world for the past 10 months. Despite many efforts, testing quickly, efficiently, and accurately has been a big problem. In this report, we aim to explain our journey and our algorithm that we made to solve this testing-problem for COVID-19.

2 Failed Attempts

Let's start with what didn't work:

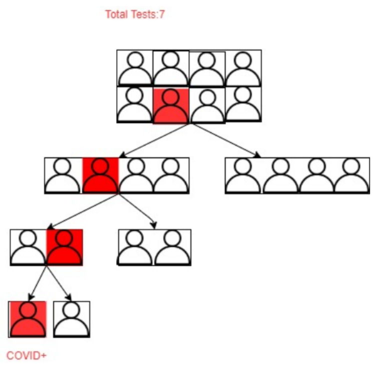
2.1 Groups of four

We first started out with a very simple algorithm that splits the population into groups of 4 and if a group tests positive, we then test them individually. This is accurate, however the number of tests is still very high and there is a lot of room for optimization. When we did this we got a point score of around 7 million.



2.2 Testing and splitting

We first tried implementing simple algorithms to get some results. One of our failed algorithms was to test in groups of 8 for anyone with covid and then if someone has covid we split the group into two groups of 4 and then test again and then divide into two groups of 2.



3 Algorithm

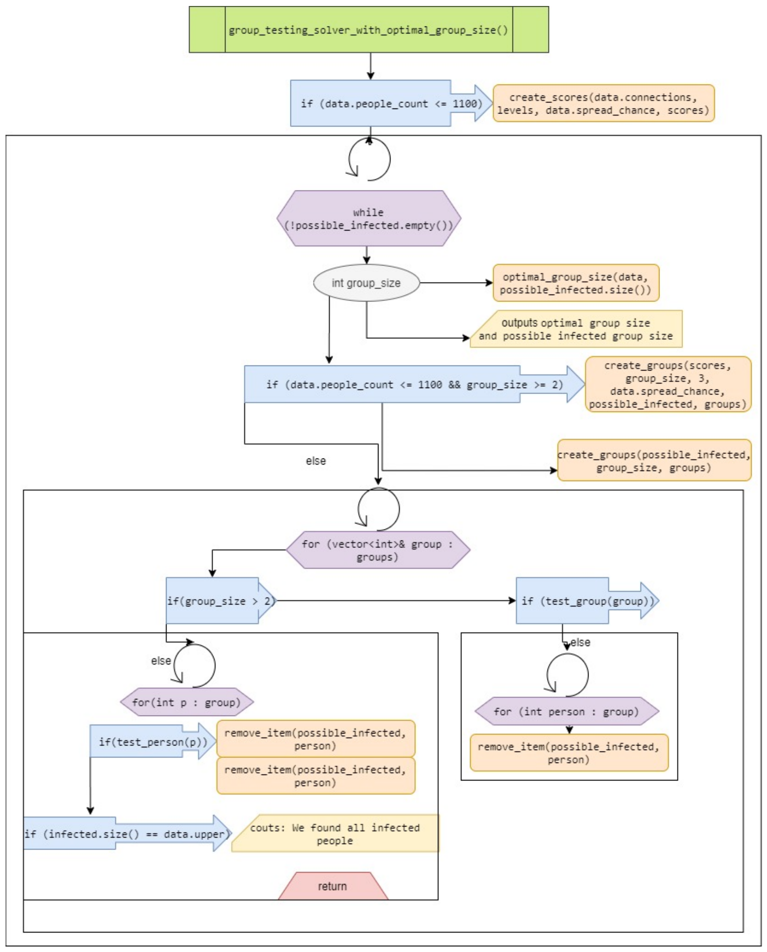
After learning that choosing arbitrary numbers such as 4, 8, 12 for our group sizes doesn't work that well, we found a paper[1] describing a method to find the optimal group-size based on the population. This paper includes an easy-to-use thumb rule:

$$\text{optimal group size} = \lceil \frac{0.693}{\text{prevalence}} \rceil$$

where prevalence is the percentage of infected people in a population at that time. An example: with 5% of the population infected the optimal group size would be $\lceil \frac{0.693}{0.05} \rceil = 14$.

This paper also describes that testing should be done in stages, where every stage removes the groups that were tested negative. Then for the next stage calculate the prevalence over the new population and create new groups for the recommended group size. This seems to work really well.

[1] `https://www.medrxiv.org/content/10.1101/2020.04.26.20076265v1.full.pdf`

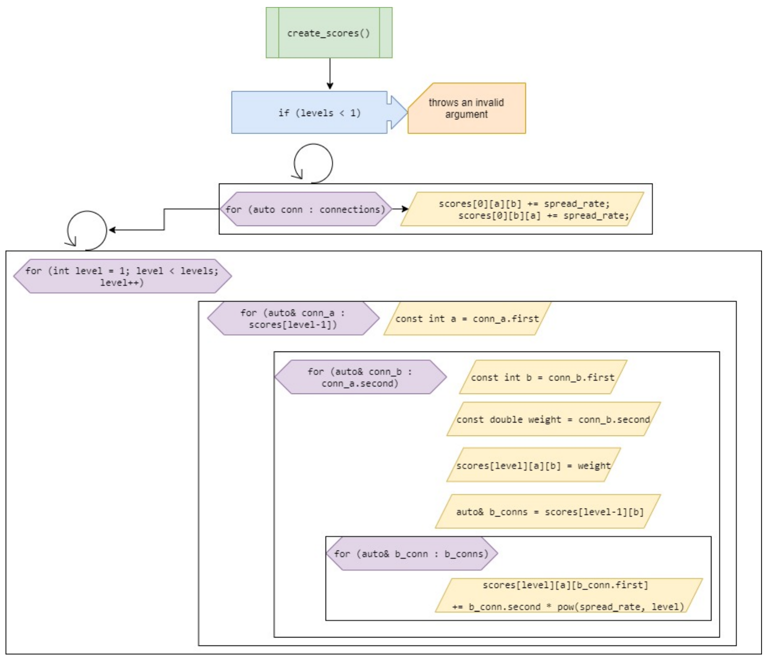

group_testing_solver_with_optimal_group_size()
if (data.people_count <= 1100)
create_scores(data.connections, levels, data.spread_chance, scores)
while (!possible_infected.empty())
int group_size
optimal_group_size(data, possible_infected.size())
outputs optimal group size and possible infected group size
if (data.people_count <= 1100 && group_size >= 2)
create_groups(scores, group_size, 3, data.spread_chance, possible_infected, groups)
else
create_groups(possible_infected, group_size, groups)
for (vector<int>& group : groups)
if(group_size > 2)
if (test_group(group))
else
else
for(int p : group)
if(test_person(p))
remove_item(possible_infected, person)
remove_item(possible_infected, person)
for (int person : group)
remove_item(possible_infected, person)
if (infected.size() == data.upper)
couts: We found all infected people
return

3.1 Finding groups

We still needed to do something with the connections. We thought of an algorithm were we first create some scores based on the connections and then create the most connected groups based on those scores.

3.1.1 Creating scores

Our idea was to create scores for every pair of persons a and b. We used an array of `map<int, map<int, double> scores` for this. The array contains a double map for every `level` we have. We currently have the amount of levels set to three. First we add the spread rate to `codes[0][a][b]` (and `codes[0][b][a]`) for every connection pair between a and b. Then for every next level (`1..levels-1`) we add all the connections of `scores[level-1][a][b]` to `scores[level][a][x]` as well! This way we can find connections between multiple people. With the `level` the amount of times this search is done is determined.


create_scores()
if (levels < 1)
throws an invalid argument
for (auto conn : connections)
scores[0][a][b] += spread_rate;
scores[0][b][a] += spread_rate;
for (int level = 1; level < levels; level++)
for (auto& conn_a : scores[level-1])
const int a = conn_a.first
for (auto& conn_b : conn_a.second)
const int b = conn_b.first
const double weight = conn_b.second
scores[level][a][b] = weight
auto& b_conns = scores[level-1][b]
for (auto& b_conn : b_conns)
scores[level][a][b_conn.first]
+= b_conn.second * pow(spread_rate, level)

3.1.2 Creating groups

Once we have the scores we can create groups out of them. We start with a collection of possibly infected people and we loop until it is empty. In every loop we first check if the amount of people still left is less or equal to the group size. If that is the case we can simply add the entire list as a new group and exit the loop. If that is not the case we first pick the highest scoring pair that is available with the `pick_best_pair` function. We add that pair to our new group and remove them from the list of people we can pick from. Then for every remaining spot in the group we pick the person that is the most connected to the current group and add them to the group and remove them from the list of people to pick from. When we've filled the group we add the group to the list of groups and continue the loop.

4 Correctness analysis

For the correctness of the algorithm, we consider two important parts, firstly that correct groups are created. That is all people are placed in groups and per stage, no groups overlap. Secondly that every infected person is found, thus successfully returning the correct result to the server.

4.1 Group correctness

All group creation happens inside the `create_groups` method. The relevant input for this method is the `group_size` and `possible_infected` vector. We have n possible infected which all need to be placed in groups. So we keep looping until `possible_infected` is empty. Thus ensuring everyone is placed in a group. Who is placed in the group is irrelevant, but they are only picked if they are in `possible_infected`. Once we pick a pair we remove them from `possible_infected` thus ensuring they are not picked again. So no overlapping groups are created. Once n is less than the `group_size` we specified, we can place all of them in that group and do not try to fill the group to the specified size. Thus again ensuring that everybody is placed in a group.

In `group_testing_solver` we also create groups using a while loop. Until `possible_infected` is empty we create groups. Thus ensuring that everyone is in a group. Once put in a group, a person is removed from `possible_infected`. Thus ensuring that no one is added twice to a group.

4.2 Finding all infected persons

To ensure that every infected person is found we need to look at a few cases. Everyone is checked either individually or in a group. No infected person is written off as not infected. Every found person is kept track of. No infected person is counted multiple times as infected. Every infected person is submitted. We have to check this for both of our algorithms: firstly `group_testing_solver_with_optimal_group_size` and secondly `group_testing_solver`.

To start with the first algorithm, every person is added to the possible infected to ensure that everyone is checked. People are placed in groups, every group gets tested either in one group test or every member gets tested individually if group size is not greater than two. Once a group tests negative we can assure that every member of that group is negative. Only then we can remove them from the possible infected. Thus ensuring no infected person is written of as not infected. This algorithm keeps on repeating until the group size is smaller than three, then everyone is tested individually. Thus finding all the infected persons. Once a test for a person is positive that person is removed from the possible infected and added to the infected. Ensuring that every infected person is kept track of and no infected person is counted twice.

The second algorithm works quite similar. Every person is added to the possible infected thus ensuring that everyone is checked. Groups are created, every group is tested. If a group tests positive it is investigated further. Every person in the group is tested to find the infected person(s). If a person tests positive they are added to the infected, thus ensuring that every infected person is kept track of and no infected person is written of as not infected.

Once the algorithms are done they both use `submit_answer` with the infected vector, which as described above contains all the infected. `submit_answer` loops over all infected persons and submits them. Thus ensuring that every infected person is submitted.

5 Complexity analysis

The complexity of the application will have to be determined based on the functions that it is calling, we have analysed all of those (see also the comments in the code) and we will highlight some of the most important functions:

5.1 Creating scores

In the `create_scores` function we first loop over all connections, this is an $\mathcal{O}(c)$ operation. The next loop we run into has three loops inside. This first loop loops over all levels l and is not that interesting since $l <<< n$. The loop inside that loops over all connections, which could be n, then in the next loop we loop over all n connections again. In that loop we loop through all n connections again. Thus we have a time complexity of $c + l \cdot n \cdot n \cdot n = \mathcal{O}(n^3)$.

5.2 Selecting groups

We have two functions for selecting groups, one which requires scores and one which does not, we will go over the one that requires scores first.

First we loop until we have placed every possible infected person in a group, this operation takes at max $\frac{n}{s}$ operations (with s = group size). In this loop we first pick the best pair, this function `pick_best_pair` is $\mathcal{O}(n^2)$. After that we remove the items we found from the possibly infected list ($\mathcal{O}(n)$). After that we select the other group members until we have filled the group. So we loop s times in which we find the most connected person ($\mathcal{O}(s \cdot n)$) and then remove this person from the list ($\mathcal{O}(n)$). So we have a time complexity of $\frac{n}{s} \cdot (n^2 + n + n + s \cdot (s \cdot n + n)) = \mathcal{O}(n^3)$ since $s <<< n$.

For selecting groups without the scores we loop n/s times (with s = group size). In that loop we loop over every member of the group so s times. In the end we get a time complexity of $\mathcal{O}(n)$.

5.3 Group testing with optimal group size

In the function `group_testing_solver_with_optimal_group_size` we test groups based on the algorithm discussed earlier. We start with calculating the scores first ($\mathcal{O}(n^3)$). Then we start to loop over $\log n$ different group size (discussed in the paper). In this loop we create the groups (either $\mathcal{O}(n^2)$ or $\mathcal{O}(n)$, so worst-time complexity is $\mathcal{O}(n^2)$). Next we loop over all groups ($\mathcal{O}(n/s)$, where $s =$ group size). In those loops we either have an $\mathcal{O}(n)$ or another $\mathcal{O}(n)$ algorithm. Thus in the end we have $n^3 + \log n \cdot (n^2 + (n/s) \cdot n) = \mathcal{O}(n^3)$.

5.4 Group testing with fixed group size

In the function `group_testing_solver` we start by creating scores ($\mathcal{O}(n^3)$) and creating groups ($\mathcal{O}(n^3)$). Then we continue by looping over n/s, with $s =$ group size, in which we also loop over s again. So in total we have $n^3 + n^3 + n/s \cdot s = \mathcal{O}(n^3)$.

5.5 Main function

In the main function the most complex functions we run are `group_testing_solver_with_optimal_group_size` and `group_testing_solver`. These functions both have a complexity of $\mathcal{O}(n^3)$ so our algorithm has an a complexity of $\mathcal{O}(n^3)$.