COVID-19 Scheduling Algorithm

Ruiming Li, Jiangxue Han, Isaac Chang, Rahul Palnitkar

December 2020

1 Abstract

In this paper, we discuss an algorithm that schedules Haverford classes in-person or online so as to maximize the number of students attending in person classes while keeping the number of students infected by COVID-19 under a certain threshold. Our finding from the algorithm includes: 1) we prefer removing large and medium-sized classes while leaving only small classes in-person 2) we prefer converting to online for classes taken by a diverse range of people, defined by a clustering method that is mostly representative of different majors.

We then analyze our algorithm's effectiveness to brute-force solution within a partial input, and we further consider modification to the problem, including, but not limited to, multiple initial infected people, students and professors having different rates of transmission. Finally, we analyze the human impact of our algorithm, paying special attention to its potential negative consequences.

2 Introduction

2.1 Motivation

In the best of times, scheduling college classes is unpleasant. With only so much time in a day, and with finite professors, it can be hard to make everyone happy with their schedule. These, however, are not the best of times. With the COVID-19 pandemic, schedulers face yet another problem: should classes be remote or should they continue to be in-person, or some hybrid of the two? Assuming, as many do, that in-person classes are preferable to online classes, an interesting question arises: How can we maximize the number of students in in-person classes, while keeping the number of people infected over a semester under a given threshold?

2.2 Problem Definition

Our goal is to allocate classes as in-person or online-only so that all students are in the same courses at the same times as given; in-person courses are considered infection opportunities and all students and the professor in the course may be infected by any student or professor in the course currently in the infectious period of SIR.

We define the total sum over all students of the number of courses they attend in-person the total number of in-person student seats. An allocation of in-person or online-only courses is considered optimal if it contains the maximum number of total student seats in in-person classes while remaining under or equal to the given acceptable number of infected students. The maximum such value is 4 times the number of students if all students take 4 courses.

2.3 Problem Parameters

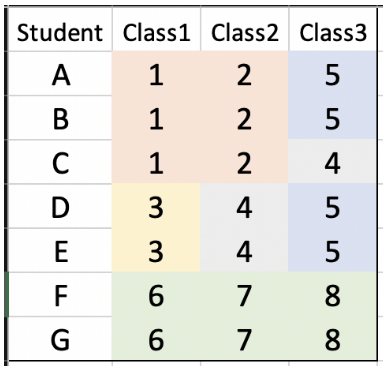
In this project, we considered the following parameters in the problem: 1) **Schedule** S: A list of classes with enrolled students, 2) **Contagion probability** p: the probability that the virus spreads from one person to another, 3) **Infection length** d: the length of time (in days) when people are in the infectious stage, 4) **Semester length** t: the number of days (the semester length), and 5) **Threshold** M: acceptable number of infected students and professors.

3 Algorithm Description

3.1 Construction of Graph from a Given Schedule

The input schedule S has a list of students and the classes they take. Initially we assume all classes are in person, so students are connected directly to people in the same class. We represent each student as a node and each class as a labeled edge in a graph G. Two students A, B can be connected by an edge labeled with multiple classes such as $1, 2$ if they take both classes together.

All students in a component C are connected in graph G, and are completely disconnected across components. This means that if one student is taking both class 1 and 2, all students in class 1 and 2 are connected in the same component. If there are no students taking both class 1 and 2, and there are no students from class 1 and 2 meeting directly in any class 3 or connected indirectly through any other class, students in class 1 and 2 are in two distinct components.



Student	Class1	Class2	Class3
A	1	2	5
B	1	2	5
C	1	2	4
D	3	4	5
E	3	4	5
F	6	7	8
G	6	7	8

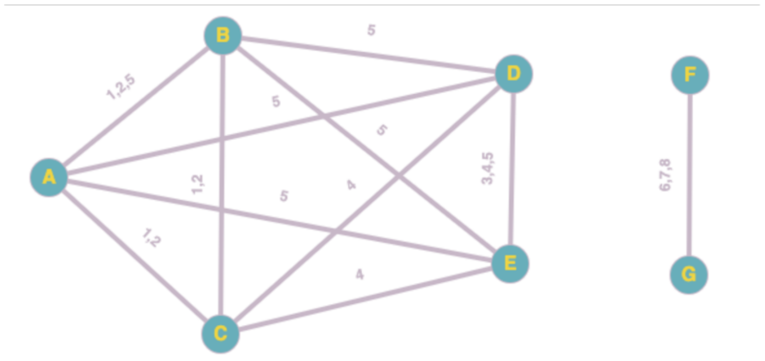


Figure 1: Left: a sample schedule S; Right: Graph G constructed from schedule S

3.2 Expected Number of Infection of Schedule Graph

Let t_k be the number of connected students in the k^{th} component and the number of professors teaching classes in that component. We assume all t_k people are equally susceptible to COVID, so given a probability of infection p, days of infectivity d, and length of semester t, we can use the available SIR formula[1] to calculate the maximum number of infected students within a component k when 1 student in component k gets infected.

$$I_k^* = max(\mathbf{SIR}(p, \frac{1}{d}, t_i, 1)),\ t_i = 0 \cdots t$$

Hence, in graph G, the maximum expected number of the infected population I in graph G at time is a weighted sum of the expected maximum infected population across all k components:

$$\mathbf{E}(max(I) \mid G) = \sum_{k=1}^{m} \frac{t_k}{T} I_k^*$$

where $T = \sum_{k=1}^{m} t_k$ is the total number of students and professors in in person classes and I_k^* is the maximum number of people infected during the SIR model simulation for component k with model population of t_k, infectious rate of p, recovery rate of $\frac{1}{d}$ and time of t.

[1]David Smith and Lang Moore. "The SIR Model for Spread of Disease - The Differential Equation Model." The SIR Model for Spread of Disease - The Differential Equation Model — Mathematical Association of America. Accessed December 7, 2020. https://www.maa.org/press/periodicals/loci/joma/the-sir-model-for-spread-of-disease-the-differential-equation-model.

3.3 Algorithm Motivation

By our construction of graph G and the definition of expected number of infection, we know that every component is disconnected from each other, and infection in one component is contained in that component. Within a component, the number of infections increases exponentially with the number of students, so the ideal situation is to have many small components in G so one infection in a component will not exceed the limit imposed and keep students and professors safe.

Our goal is therefore to find the best edges to remove on graph G so they lead to as many small clusters as possible. At the same time, we want these removed edges to represent a set of classes that have the smallest number of people.

3.4 Identify Potential Components via Clustering

By our motivation, we want to find classes that will result in many small components after making them online. By inspection of graph G constructed from various schedule, we discovered that initially almost every person is in a same, large component because of the high degree of indirect connections among students. Cutting from component is therefore not a good starting point.

Instead, we can first identify clusters of students based on the similarity of their schedule and remove classes that are connecting different clusters. Clustering is also intuitively motivated because we are likely to find people from the same major taking similar classes and having similar schedule. It is more helpful to remove classes between clusters and less helpful make classes within a cluster online, since people within a cluster (major students) are connected via many other (major) classes.

Hence, we use Jaccard distance [2] as a similarity measure for the schedule of two students. We consider each schedule as a set of classes, and calculates the ratio of the size of the intersection of two sets of classes to that of the union of the two sets of classes.

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

After calculating the Jaccard distance between every student, we then use a union find algorithm to group students into clusters whose Jaccard distance is above 0.5, a threshold from experimentation.

[2] Stephanie Glen. "Jaccard Index / Similarity Coefficient" From StatisticsHowTo.com: Elementary Statistics for the rest of us! https://www.statisticshowto.com/jaccard-index/

Similarity	A	B	C	D	E	F	G
A	1.00	1.00	0.50	0.20	0.20	0.00	0.00
B		1.00	0.50	0.20	0.20	0.00	0.00
C			1.00	0.20	0.20	0.00	0.00
D				1.00	1.00	0.00	0.00
E					1.00	0.00	0.00
F						1.00	1.00
G							1.00

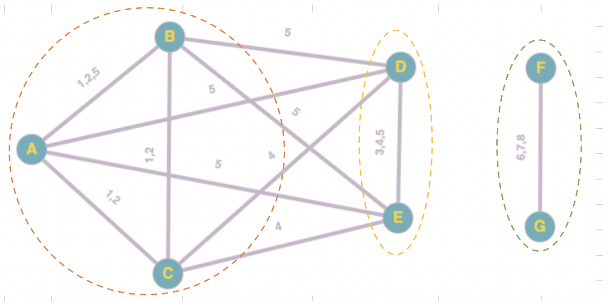


Figure 2: Left: Jaccard distance between students with schedule S; Right: Clusters found in graph G

An important distinction between cluster and component is that people within the same component can be in different clusters, but people in different components (not taking the same classes) cannot be in the same cluster (unless the threshold for Jaccard distance is 0). Our motivation is that these clusters can become separated components once we move enough classes online.

3.5 Identify Best Classes to Make Online based on Clusters

Once we identify clusters as potential components, we then need to identify classes between clusters to make online so we can separate a large component into many small components. For a class, we iterate through every cluster that the class is in, and look at ratio of the number of people in this class and outside the cluster, to the total number of people in the class, and we add the ratio to a score for each class.

$$score(class) = \sum_{cluster} \frac{|\text{students in } class \text{ outside } cluster|}{|\text{students in } class|}$$

class	in_out_cluster1	in_out_cluster2	in_out_cluster3	score
1	3,0	0,0	0,0	0
2	3,0	0,0	0,0	0
3	0,0	2,0	0,0	0
4	1,2	2,1	0,0	1
5	2,2	2,2	0,0	1
6	0,0	0,0	2,0	0
7	0,0	0,0	2,0	0
8	0,0	0,0	2,0	0

Figure 3: number of people in each classes inside and outside each cluster for graph G after clustering. Class 4 and 5 have a higher score and are better to be made online

This is also intuitively motivated because a higher score for a class means: 1) there are many people in the class outside a cluster so the ratio is high 2) people from this class are from many clusters so we added a ratio many times.

We then sort all classes by highest score to lowest and make the highest scored class online.

3.6 Iteration until Expected Infection Below Threshold

Since each time we make a class online, the graph G is updated. We have to repeatedly calculate cluster and score the edges based on the latest graph. We thus repeat step 3.4-3.5 and calculate expected infection based on 3.2, until it is lower than the threshold M.

4 Pseudocode

```
Function getGraph(course_teacher, course_students)
    (1 = in-person, 0 = online)
    courses, students, teachers = Hashmap()
    for course_id,students_list ∈ course_students do
        for name ∈ students_list do
            students[name] = Person(name = name,
                                    role = student,
                                    classes = [ ])
        end
        students_in_class = student object values of students dictionary
        courses[course_id] = Class(name = course_id,
                                   people = students_in_class,
                                   status = 1)
        for name ∈ students_list do
            add class object with id=course_id to students[name]'s list of classes
        end
    end
    for course_id, name ∈ course_teacher do
        teachers[name] = Person(name = name,
                                role = teacher,
                                classes = [ ])
        teacher_obj = teachers[name]
        if course_id ∉ classes then
            (no haverford student is taking this course)
            courses[course_id] = Class(name = course_id,
                                       people = [teacher_obj],
                                       status = 1)
        end
        else
            add class object with id=course_id to teacher[name]'s list of classes
        end
    end
    for student ∈ keys of students do
        get neighbors of student
    end
    for teacher ∈ keys of teachers do
        get neighbors of teachers
    end
    return classes, students, teachers
```

```
Function getComponents(people)
    components = hashmap() (k: person(), v: $|C|$ s.t $person() \in C$)
    people_visited = hashmap()
    for person $\in$ keys of people do
        people_visited[person] = False
    end
    for person in keys of people do
        if person is not visited then
            s = stack()
            push person onto s
            size = 0
            while s is not empty do
                p = pop person off of s
                people[p] = True (mark as visited)
                size += 1
                for neighbor of p do
                    if neighbor is not visited then
                        push neighbor onto s
                        people[neighbor] = False (mark visited to avoid pushing it
                          again)
                    end
                end
            end
            components[person] = size
        end
    end
    return components
```

```
Function SIR_maxInfected(size, p, p_recovery, time, init_infected)
    $S_{\text{cur}}$ = size - init_infected
    $I_{\text{cur}}$ = init_infected
    $R_{\text{cur}} = 0$
    $\Delta t = 1$
    $t = 0$
    while $t < time$ do
        $a = \frac{p}{\text{size}} \cdot I_{\text{cur}} \cdot S_{\text{cur}}$
        $b = p_{\text{recovery}} \cdot I_{\text{cur}}$
        (SIR model deq system)
        $S_{\text{next}} = S_{\text{cur}} - a$
        $I_{\text{next}} = I_{\text{cur}} + a - b$
        $R_{\text{next}} = R_{\text{cur}} + b$
        if $I_{next} < I_{cur}$ then
            return $I_{\text{cur}}$ ($I$ has 1 local max so if next is smaller, cur is global max)
        end
        else
            $S_{\text{cur}} = S_{\text{next}}$
            $I_{\text{cur}} = I_{\text{next}}$
            $R_{\text{cur}} = R_{\text{next}}$
            $t$ += $\Delta t$
        end
    end
    return $I_{\text{cur}}$

Function expectedTotalMaxInfected(components, p, p_recovery, time)
    totalSize = 0 (total number of in-person people)
    sizes = [ ]
    for compSize ∈ values of components do
        totalSize += compSize
        append compSize to sizes
    end
    expectation = 0
    for size ∈ sizes do
        expectation = $\frac{\text{size}}{\text{totalSize}}$ · SIR_maxInfected(size, p, $p_{\text{recovery}}$, time, 1)
    end
    return expectation
```

```
Function isScheduleGood(people, p, p_recovery, time, L)
    components = getComponents(people)
    expectation = expectedTotalMaxInfected(components, p, p_recovery, time)
    return expectation < L

Function findClusters(people, threshold)
    (people is a hashmap of k: name, v: Person())
    i = 0
    person_group = hashmap()
    for p ∈ keys of people do
        person_group[people] = i
    end
    Function union(person1,person2)
        if person2 not in the same group as person1 then
            people[person2]=people[person1]
        end
    for unordered pair of p1,p2 ∈ keys of people do
        (p1 and p2 are person())
        intersection = number of classes p1 and p2 share that are in-person
        union = number of distinct classes p1 and p2 take that are in-person
        similarity = $\frac{\text{intersection}}{\text{union}}$ rounded to 2 decimal places
        if similarity > threshold then
            union(p1, p2)
        end
    end
    similarity_groups = hashmap() (k: group number, v: set of people)
    for person, group_number in person_groups do
        if group_number ∉ keys of similarity_groups then
            similarity_groups[group_number] = set()
        end
        add person to similarity_groups[group_number]
    end
    return similarity_groups
```

```
Function classes_people_in_out(cluster)
    (cluster = set of people in a cluster)
    classes_in_out = hashmap()
    for person1 ∈ cluster do
        for class that person1 is taking do
            (count people in this cluster taking this class)
            if class ∉ classes_in_out[class] then
                classes_in_out[class] = [1, 0]
            end
            else
                classes_in_out[class][0] += 1
            end
            (count people out of cluster taking this class)
            (only need to do this once for each class)
            if class is in-person then
                if classes_in_out[class][1] == 0 then
                    for person2 ∈ this class do
                        if person2 not ∈ cluster then
                            classes_in_out[class][1] += 1
                        end
                    end
                end
            end
        end
    end
    return classes_in_out

Function mergeDict(dict1, dict2)
    (keys are strings) merged = dict1
    for key, value in dict2 do
        if key is a key of merged then
            oldValue = merged[key]
            merged[key+'_1']= oldValue
            merged[key+'_2']= value
        end
        else
            merged[key] = value
        end
    end
    return merged
```

```
Function findClassMakeOnline(clusters)
    classes_score = hashmap()
    class_with_max_score = None
    max_score = -1
    for cluster ∈ clusters do
        classes_in_out = classes_people_in_out(cluster)
        for class, in_out ∈ classes_in_out do
            in = in_out[0]
            out = in_out[1]
            score_within_cluster = $\frac{\text{out}}{\text{in + out}}$
            if class ∉ keys of classes_score then
                class_score[class] = score_within_cluster
            end
            else
                class_score[class] += score_within_cluster
            end
            if class_score[class] > max_score then
                class_with_max_score = class
                max_score = class_score[class]
            end
        end
    end
    return class_with_max_score

Function getSchedule(course_teacher,course_students, p, d, t, L)
    classes, students, teachers = getGraph(course_teacher,course_students)
    similarity_threshold = 0.5
    $p_{\text{recovery}} = \frac{1}{d}$
    people = mergeDict(students,teachers)
    while not isScheduleGood(people, p, p_recovery, time, L) do
        clusters = find_clusters(people,similarity_threshold)
        class = findClassMakeOnline(clusters)
        make class online
    end
return classes
```

5 Time Analysis

Our algorithm uses custom person and class data structures to store relations between people depending on the classes they share together.
The class structure has 3 attributes:

1. name, the name/id of the course
2. people, an array of person objects representing the students and professors in that class
3. status, a boolean of whether the course is online or in-person

The person structure has 4 attributes:

1. name, the name/id of the course
2. role, a boolean of wherther the person is a professor or a student
3. classes, a set of in-person classes this person is taking part in
4. neighbors, a set of people that this person has at least 1 in-person class with

The class structure also has 2 methods:

1. makeOnline, a method to set the status to be online and remove the class from each of people in the people's set of in-person classes. Then, it gets the new neighbors for each person in that class.
2. makeInPerson, a method to set the status to be in-person and add the class to each of the people in the people's set of in-person classes. Then, it gets the new neighbors for each person in that class.

The person structure also has 1 method:

1. getNeighbors, a method to get the set of people that this person has at least 1 in-person class with

For any class, there is a limit to the total number of people allowed to take that class. Denote this upperbound as l. Denote the total number of students and professors as N, the total number of classes as C, the time to recover from the virus as d, the maximum number of people allowed to be infected at any point as M and the semester length as t.

5.1 Time Analysis of getNeighbors

Each person has an upperbound of 5 in-person classes. *getNeighbors* loops through every in-person class a person is taking and gets the people in that class. The union of these people is the set of neighbors. Since there are at most 5 of these sets and each of the sets has at most l people, *getNeighbors* can be done in $O(l)$.

5.2 Time Analysis of makeOnline/makeInPerson

The methods, *makeOnline* and *makeInPerson*, have to set the the status to be online/in-person and remove/add the class from each of the people taking this classes' set of classes, respectively.

Setting the status is done in $O(1)$. The people taking this class is stored in the people attribute, so, we can get the people taking this class in $O(1)$. Then, we need to get the new neighbors for a person in this class. Getting the new neighbors for a person is done in $O(l)$. Since there are at most l people in a class, *makeOnline/makeInPerson* is done in $O(l^2)$.

Our algorithm starts after processing our inputs from which we have

1. course_teacher, a hashmap mapping course_id to the name of the professor teaching that course
2. course_students, a hashmap mapping course_id to an array of the students taking that course
3. p, the probability of infection between 2 people
4. d, the time it takes to recover from the virus
5. t, the length in days of the semester
6. L, the limit on the total number of infected people at any point during the semester

5.3 Time Analysis of getGraph

getGraph uses the course_teacher and course_students hashmaps to build the person and class objects which inherently store how people are connected.

The first 2 level nested for loops initialize the students as person objects and classes as class objects. The second 2 level nest for loops initialize the teachers as person objects and any remaining classes that were not initialized by the first nested for loops (if no haverford student is taking a class that a haverford professor is teaching). These two 2 level nest for loops are done in $O(N + C)$.

Then, g*getGraph* loops through the student and professor objects to initialize their neighbors. Since there are N people and getNeighbors takes $O(l)$. This part is done in $O(lN)$.

So, *getGraph* is done in $O(lN + N + C)$.

5.4 Time Analysis of getComponents

getComponents is similar to the depth first search algorithm. The only difference is that it counts the number of people it visits in each component. Since this is done along side the graph traversal, it adds no complexity. The vertices of G are people and the edges are in-person classes joining 2 people if they both take it. There are N people. Since all classes may be in-person and there are at most l people per class, there are at most lC edges. So, *getComponents* is done in $O(N + lC)$.

5.5 Time Analysis of SIR_maxInfected

SIR_maxInfected uses the system of differential equations behing the SIR model to calculate the amount of susceptible, infected, recovered people every day of the semester for a component. Calculating each of these populations for 1 day can be done in $O(1)$ because the calculations only depend on the populations of the previous day , which are recorded, and arguements passed into the function. Since there are t days in the semester, *SIR_maxInfected* is done in $O(t)$.

5.6 Time Analysis of expectedTotalMaxInfected

expectedTotalMaxInfected uses *SIR_maxInfected* to get the expected maximum amount of infected people for each component with 1 patient zero weighted by the probability of that component having the 1 patient zero out of the whole college. The probability is computed in $O(1)$ for a component. Assuming that people take on average of 4 classes, the maximum amount of components is $\frac{C}{4}$ where in each component, the people take the same 4 classes and no people from other components take any of those classes. Since *SIR_maxInfected* is done in $O(t)$, *expectedTotalMaxInfected* is done in $O(tC)$.

5.7 Time Analysis of isScheduleGood

isScheduleGood gets the components of G with *getComponents*. Then, it uses those components to get the expected total amount of infected people with *expectedTotalMaxInfected*. Lastly, it checks whether expected the total maximum amount of infected exceeds M. This is done in $O(1)$. Since *getComponents* is done in $O(N + C)$ and *expectedTotalMaxInfected* is done in $O(tC)$, *isScheduleGood* is done in $O(N + tC)$.

5.8 Time Analysis of findClusters

findClusters first sets every person to be in a different cluster by using a hashmap where the keys are person objects and the values are integers representing which cluster that person belongs to. Since there are N people, this is done in $O(N)$.

Then, for every combination of two people, it computes the Jaccard distance for their classes. Since each person takes on average 4 classes to check if they belong in the same cluster. This is done in $O(1)$. Since there are $\binom{N}{2} = \frac{N^2-N}{2}$ combinations of 2 people, this step is done in $O(N^2)$.

Next, people of the same cluster are grouped together into the same cluster. The cluster number for a person is found in $O(1)$ since it is stored in hashmap. Since there are N people, this step is done in (N).

So, *findClusters* is done in $O(N^2)$.

5.9 Time Analysis of classes_people_in_out

For each class a person in a cluster is taking, *classes_people_in_out* computes how many people inside/outside the cluster take that class.

It loops through every person in a cluster and goes through their classes and counts the number of people in this cluster and out (if in-person) of this cluster. There can be at most N people in a cluster. Assuming people take on average of 4 classes, *classes_people_in_out* is done in $O(N)$

5.10 Time Analysis of mergeDict

mergeDict merges 2 dictionaries with string keys while handling key conflicts. Let the one dictionary be A and the other be B. *mergeDict* loops through the keys and values of B and tries to add it to A. If there is a key conflict, the key names are changed by adding a number identifier. Since checking for a conflict is done in $O(1)$, adding a key value pair from B to A is done in $O(1)$. Since there are $|B|$, key value pairs, *mergeDict* is done in $O(1)$.

5.11 Time Analysis of findClassMakeOnline

First, *findClassMakeOnline* goes through every cluster and finds the number of people in/out of the cluster taking a class in the cluster with *classes_people_in_out.* For a cluster, *classes_people_in_out* is done in $O(N)$. The number of clusters is a linear factor of the number of in-person classes. Since there can be C in-person classes, getting every in/out for every class of every cluster is done in $O(NC)$.

findClassMakeOnline also goes through every class of every cluster and finds the sum of $\frac{\text{out}}{\text{in+out}}$ across all clusters with that class. Since there are C classes, and the number of clusters is $O(C)$, this is done in $O(C^2)$

findClassMakeOnline returns the class with the largest $\sum \frac{\text{out}}{\text{in+out}}$ score. Since it keeps track of the class with the largest such score as it sums the score of a class across all clusters, this is done in $O(1)$.

So, *findClassMakeOnline* is done in $O(NC + C^2)$.

5.12 Time Analysis of getSchedule

This is the wrapper function for getting the best schedule from our algorithm.

First, *getSchedule* gets the graph with *getGraph*. Then, while the expected maximum number of infected people is less than L, *getSchedule*

1. finds G's clusters in $O(N^2)$ with *findClusters*
2. gets the class to make online in $O(NC + C^2)$ with *findClassMakeOnline*
3. makes that class online in $O(l^2)$

The expected maximum number of infected people is less than L is check with *isSchedule-Good* in $O(N + tC)$ in every iteration.

We can expect the expected maximum total number of infected people to decrease by 1 on average every iteration. So, number of iterations of the while loop is $I_0 - M$.

where I_0 is the expected maximum total number of infected people at the start.

However, this is only true if $I_0 > M$ and p is greater than $\frac{1}{d}$ by some realistic amount.

Assuming that conditions are realistic, we can expect I_0 to be of $O(N)$. But since $M << N$ practically, there are $O(N)$ iterations.

Lastly, *getSchedule* returns the classes hashmap.

So, our algorithm, *getSchedule*, is done in

$$O\big(N(N^2 + NC + C^2 + l^2) + N(N + tC)\big)$$

$$= O\big(N(N + C)^2 + Nl^2 + N^2 + tNC)\big)$$

For Haverford College and in general, $l << N, C$. So,

$$= O\big(N(N + C)^2 + N^2 + tNC)\big)$$

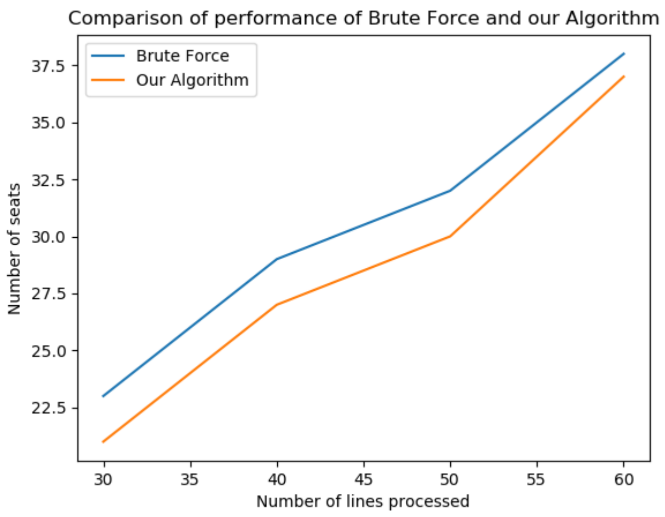
In general, $t, C << N$. So,

$$= O(N^3)$$

6 Performance Analysis

6.1 Quality Analysis

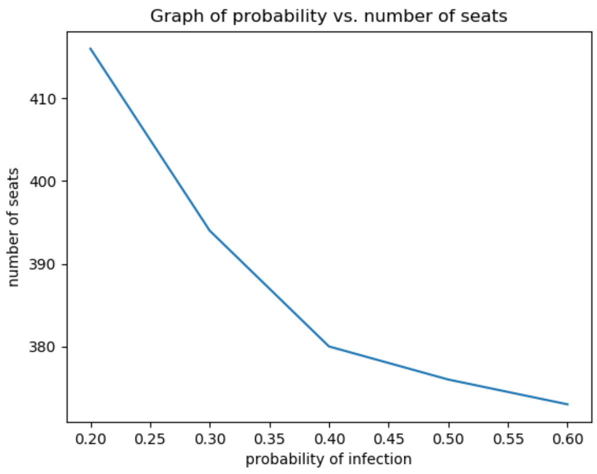
We included a brute force approach that compares every possible combination to find out the best. This is the optimal solution for our problem setting. However, it is extremely expensive in terms of time and we are only able to run very limited number of lines in the input file. Here is a graph showing performance of our algorithm compared to brute force approach.

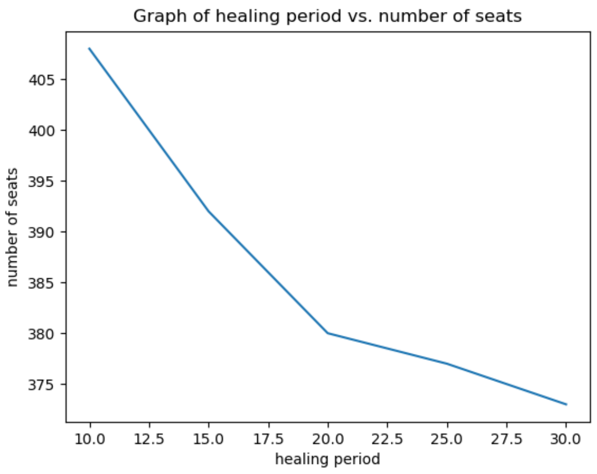


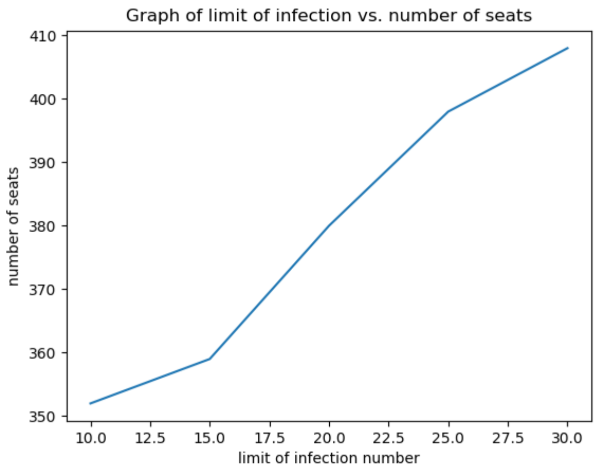
So if we consider brute force as upper bound, the fraction representing our lower bound is 0.91, because this is the lowest percentage our algorithm get in the 4 trys of different sizes. However, this is not a guarantee of performance because of out limited ability of running brute force.

6.2 Parameters

For probability of infection, the number of seats decrease as it goes up. This is reasonable since if there's higher chance for infection in person classes will be more limited. The decrease is more steep in the beginning than toward the end. We suspect it's because when the probability is already high, most people are already infected, leaving little space of more infection.
Similarly, when healing period is longer number of seats drop.
When limit of infection is larger, there are more seats. This is natural because we are tolerating more cases of infection. The speed of increase is almost linear and steady.



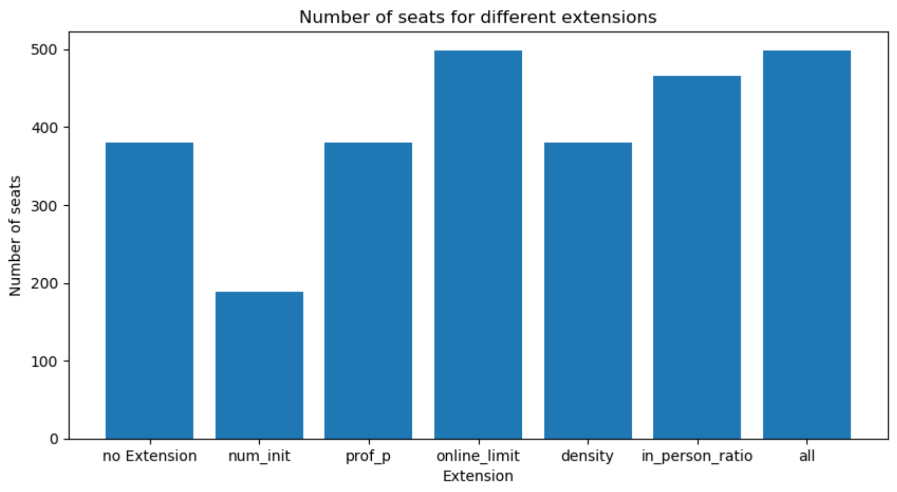




6.3 Extensions Results

There is a graph at the end of 5.3 in next page that shows number of seats result when including no extension, only each extension, and all of them. For this graph, the extensions used these input parameters:
$num\_init = 10, prof\_probability = 0.6, online\_student\_limit = 800, density = 0.3, in\_person\_ratio = 0.15$.
There are a few things worth noticing:
(1) The inclusion of a different probability of infection for professors did not make much change to our results. We think it might be due to relatively small number of professors compared to students, and our algorithm calculated the weighted average for probability.
(2)Density extension also did not have much effect on our results. We suspect it is due to

our approach of only modifying one component that is the most dense.
(3)The in person ratio of the semester only start to have effect on our result if it makes the in person part of semester significantly shorter than the original length. This is probably because after certain time the infection number stopped increasing because the components are already separated and the infected people can only infect those within their own components.



6.4 Haverford scheduling

Predictions: we constructed a graph using the hybrid in-person classes in this semester's schedule and set initial parameters: Infection rate = 0.15, length of infection = 10, length of semester=100, initial infected person = 1. Using our expected number of infection formula, we get an expected number of 70 people infected. Through inspection of the components in graph, we discovered that there are only 6 components in the graph and the largest component has more than 1000 people. Without running our cutting algorithm, it is plausible to have large number of infections due to the high connectivity within the graphs.

Migration policies: if large number of infection happens within a component, that means we can clearly identify which component are infected and the easiest policy is to make all classes online within that component. It can avoid spreading the virus to a larger portion of the component immediately. Using our cutting algorithm, we can also identify clusters within the component and partially make classes within the component online, especially those clusters with an infected person.

7 Extensions

We considered two types of extensions: first, we wanted to better reflect the situation students and professors face on the Haverford campus and model infection spread more realistically. We considered many real-life conditions that are simplified in our former problem set-up, such as having more than a single initial infection on campus, making it more likely that people who take multiple classes together get infected, and giving professors and student different raters of infection.

Second, we wanted to investigate ways to improve the class experience for students and professors. We considered situations where a certain portion of the semester goes entirely online, so more people can take in-person classes during the in-person half of the semester. We also attempted to improve the student experience by introducing more constraints to the optimization problem and limiting the total number of students whose classes are all online.

7.1 Multiple Patients Zero

In the initial problem set-up, we assume there is only one initial point of infection randomly selected among all students and professors in graph G. For this extension, we consider m initial patients zero (where $m > 1$) because it is far more likely that infections are introduced to the campus through multiple contacts and sources.

This means that it is likely that infections happen and spread through more than one component, as opposed to our initial assumptions of the initial infection occurring only in a single component. Furthermore, infection in a single component can start with multiple people for the SIR formula, if more than one initial infection is introduced into a particular component. As a result, we propose a new formulation for the expected number of the infected I in graph G:

Let X be the random vector where the values, $\vec{x}$, are vectors representing the initial infected, for components $C_1, \ldots, C_n$ where

1. $\vec{x} = (x_1, \ldots, x_n)$
2. $\sum_{i=1}^{n} x_i = M$
3. $x_i \leq |C_i|$ for $1 \leq i \leq n$

and

$$P(X = \vec{x}) = \frac{1}{\binom{N}{M}} \prod_{i=1}^{n} \binom{|C_i|}{x_i}$$

(probability that x_i patient zeroes are choosen for each C_i out of all possible M combinations of patient zeroes across all N people)

Since more than 1 component may a non-zero number of patient zeros, we run the SIR model on each component concurently.

Then, $S_t(\vec{x}), I_t(\vec{x}), R_t(\vec{x})$ are the susceptible, infected and recovered population respectively of the whole school at time t and $S_t^i(\vec{x}), I_t^i(\vec{x}), R_t^i(\vec{x})$ are the susceptible, infected and recovered population respectively of component i at time t where everyone is initially susceptible or infected and the infecion across the components is described by $\vec{x}$.

In order to run the SIR for each component concurrently, we sum the susceptible, infected and recovered populations of each component at time t to get the susceptible, infected and recovered populations for the whole population at time t. In other words,

$$\begin{aligned} S_t(\vec{x}) &= \sum_{i=1}^{n} S_{t-1}^i(\vec{x}) - \frac{p}{\mid C_i \mid} I_{t-1}^i(\vec{x}) S_{t-1}^i(\vec{x}) \\ I_t(\vec{x}) &= \sum_{i=1}^{n} I_{t-1}^i(\vec{x}) + \frac{p}{\mid C_i \mid} I_{t-1}^i(\vec{x}) S_{t-1}^i(\vec{x}) - \frac{1}{d} I_{t-1}^i(\vec{x}) \\ R_t(\vec{x}) &= \sum_{i=1}^{n} R_{t-1}^i(\vec{x}) + \frac{1}{d} I_{t-1}^i(\vec{x}) \end{aligned}$$

where

$$\begin{array}{lll} S_0(\vec{x}) = N - M & I_0(\vec{x}) = M & R_0(\vec{x}) = 0 \\ S_0^i(\vec{x}) = |C_i| - x_i & I_0^i(\vec{x}) = x_i & R_0^i(\vec{x}) = 0 \end{array}$$

Then, the expected maximum amount of people infected for an initial infected vector, $X = \vec{x}$ is given by

$$E(max(I)|X = \vec{x}) = max(I_t(\vec{x}))$$

So, the expected total maximum amount of people infected is

$$E(max(I)) = \sum_{\vec{x} \in X} P(X = \vec{x}) \cdot E(max(I)|X = \vec{x})$$

$$= \sum_{\vec{x} \in X} P(X = \vec{x}) \cdot max(I_t(\vec{x}))$$

After implementing the new formula for the expected number of infections in graph G, we can use the same optimization algorithm for making classes online and minimizing the new expectation formula.

From our experiments, we first observed that the number of in-person seats stays the same, the as number of initial nodes increase number of seats become less. We suspect that this is because the semester is long enough for Covid cases to reach a limit and it stops increasing after that point. So until the number of initial infected is large enough, all of them reached that limit before end of semester.

7.2 Density

In our initial problem set-up, we assumed that all people in a component are equally likely to get infected, and we input the total number of people in the component into the SIR formula and calculated the maximum expected infection number. However, in real-life, components can have different densities and drastically different infection rates. We can define density as the unique number of classes taken by people in a component. For example, consider two components with 10 students, with each student taking the same number of classes. If the 10 people in Component One are taking five unique classes while the 10 people in Component Two are taking 20 different classes, it means that they meet in fewer unique classes and Component is more densely connected. As a result, we can reflect the various degrees of density by varying the infection rate for clusters with high density. We run the same algorithm with and without considering of density and obtained the following results:

7.3 Different Transmission Rates for Professors and Students

In our setting, we used the same transmission rate for professors and students. This means that a student and a professor who are in the same class with someone who has Covid will have equal chance of getting infected. This is for simplicity purposes, however, in reality professors would have a higher chance of infection. This is because students often go to professors to ask questions, and professors tend to circulate around the classroom when assigning group discussions or individual exercises during lectures. This gives them higher chance of infection because they tend to be in close contact with more students.
To do this, we accept another parameter input in the constraint.txt file for probability of infection for professors. We count number of Person objects whose roles are professors, and pass on this information in addition to the probability of infection for professors into $SIR\_maxInfected()$ function. This functions take in a probability for infection between connected nodes. So we used a weighted average to compute the probability:

$$p = (p * (size - num\_prof) + p\_prof * num\_prof)/size \tag{1}$$

This probability is then passed in to the SIR calculations.

7.4 Partial Remote Semester

This is a simple extension that deals with the college's plan to only run in hybrid mode for a certain proportion of the semester. For example, Bi-Co only has in person classes until

Thanksgiving this semester. To do this, we just added a parameter *in_person_semester_ratio* that is the proportion of time where in person classes are allowed. We just updated *semester_length* to be *semester_length* * *in_person_semester_ratio*. Number of seats does not change until we significantly reduce length of semester. Our suspect is that it already reached a limit not long after start of semester.

	No. of In-Person Seats
Normal Semester Length	380
Reduced Semester Length(0.15)	466

7.5 Limited Number of Students with All Classes Online

We originally only considered maximizing number of total seats in our algorithm, but ignored specific situations for individuals. Some students might be forced to have a fully online schedule. This is problematic because:
(1) These students will be lack of social life and might feel left out.
(2) Having fully online schedule is dangerous for international students since it might affect their visa status.
(3)Students who have fully online schedule might not be able to enjoy some of the school's resources.
So we set up a parameter as limit of number of students who are fully online. Every time before we make a course online, we check all the students in this course, and then access every student's four courses taken. If three of them are already online, we increase one for the number of online only students. If the updated number of online only students exceed our limit, we skip this class and continues into the next one. The concern for this approach is that for certain inputs it might never be able to find a schedule that satisfies the requirement.

8 Human Impact

According to our experimental analysis, when we run our algorithm on a set of classes of different sizes, our algorithm tries to move larger classes online first. This is to be somewhat expected —of course, larger in-person classes mean more students and professors interacting in an enclosed space, which means a greater chance for an infection to be spread. So that the algorithm favors smaller in-person classes and moves larger classes online is not surprising. This does, however, have several consequences. First, many of the largest classes on campus are introductory classes —in other words, classes that first-years take. This means that first years are more likely to have their classes online, which is quite unfortunate for freshman expecting a traditional college experience. Indeed, this makes it harder for the first-years to interact with their professors, form study groups, and indeed bond with their classmates, which can be detrimental to many students.

We can also consider large laboratory classes, such as the Biology and Chemistry "Super-labs." Since they are larger, they are more likely to be moved online, and classes as focused on hands-on work with specialized equipment as these would undoubtedly be negatively affected by such an algorithm. So, our algorithm would be more likely to affect natural sciences majors in large laboratory classes.

Now, what students would our algorithm benefit? Well, after running the algorithm on a particular set of data, it seems to benefit very small classes, such as very high-level or obscure electives. As upperclassmen in less-popular majors are more likely to take these classes, they stand to benefit more from this algorithm

Now, we would be remiss if we didn't talk about those not considered in our algorithm, but who would undoubtedly be affected by it's real-world use. In particular, we note that our algorithm does not consider the staff at all —only the faculty and the students. Indeed, since we don't consider the staff, we implicitly admit that our algorithm doesn't care if the staff (cleaning, dining, etc.) are exposed to the virus and infected. The vast majority of the staff does not have the ability to do their work from home, so they must be present on the campus, and as such, are at risk of getting infected. It's also worth noting that because our algorithm doesn't consider the staff at all, that also means the algorithm under-estimates the risk of infection to the students and faculty, because the staff members are also at risk of getting infected and infecting others. Most importantly, because the vast majority of the custodial and dining staff are People of Color, in some senses, our algorithm, by failing to consider how the staff are put at risk, is implicitly racist. We acknowledge that a better-designed algorithm would need to take into account the staff, and the risks they face.

9 Conclusion

For our Covid scheduling algorithm, we choose to ignore certain parameters such as time and frequency of a class in order to reduce complexity. In doing so, we were able to directly model how the virus spreads with the SIR model and find the expected maximum amount of infected people throughout the semester. This allowed us to easily determine whether a schedule would meet the infected threshold and whether we needed to make a class online. Through the design of the algorithm and the analysis of the results, we realized that the way we choose how to make a class online greatly impacted the how our algorithm made the schedule. Our experimental analysis shows that our algorithm favors making larger classes online. While it doesn't take into account what subject this class is under, we can expect larger classes to include introductory course and STEM courses. Since introductory classes are mostly populated by underclassmen, this would make the early college year experience unwelcoming and disappoing. Also, many STEM classes feature on hand experiments which would be removed when shifting to online instruction.

One of the biggest issues we had in our algorithm design was decidng whether in-person seats were more important than lowering the expected maximum amount of infected people. We decided that since this algorithm can fataly impact people we opted to focus on making sure the schedule's expected maximum amount of infected people was always strictly lower than the threshold. The social and human impact of this algorithm definitely affected the way we approached Covid Scheduling and added additional, but necessary complexity to the problem.