Assignment 2

CSCI 270

Paul Kim | pbkim@usc.edu | 1723717002

Question 1

In this problem, we will look at how to manipulate the outcome of an election. In many elections, voters can only vote for one candidate — if so, there is really only one reasonable voting rule, which is to choose the candidate with the largest total. But as you probably know, this leads to a lot of speculating to avoid "wasting" one's vote on a candidate who won't win anyway. For that reason, some elections ask voters to rank all candidates from most favorite to least favorite; with this extra information, one can then do a better job picking a "good" candidate. There are a lot of different voting rules for choosing a candidate — you may have heard about Single Transferable Vote (STV), also known as Ranked Choice voting. Another well-known method, and the one we will consider here, is the Borda Count, named after Jean-Charles de Borda, who invented it in the late 18th century.

Assume that there are $m \geq 2$ candidates and n voters. Each voter ranks all m candidates from most to least favorite. For voter $v = 1, \ldots, n$, their order is a permutation π_v, and we write $\pi_v(i)$ for the candidate that voter v has in position i. Voter v's most favorite candidate is $\pi_v(1)$, and their least favorite is $\pi_v(m)$. Under the Borda count rule, candidate c gets $m-1$ points for each first-place vote they get, $m-2$ for each second-place vote they get, and so on; with 0 points for each last-place vote they get. The points are added up, and the candidate with the largest total number of points wins.[1] As an example, suppose that there are three voters and four candidates, and the rankings of the voters are $\langle A, B, C, D\rangle, \langle B, C, A, D\rangle, \langle C, D, A, B\rangle$. Because $m = 4$, candidate A gets $3 + 1 + 1 = 5$ points, B gets $2 + 3 + 0 = 5$ points, C gets $1 + 2 + 3 = 6$ points, and D gets $0 + 0 + 2 = 2$ points. Here, C would win.

Now assume that you are voter n, and your favorite candidate would be A. You would like A to win. You can see all the ballots of all other voters. The question is: is there some ranking you can put on your ballot such that A is the unique winner, i.e., A gets strictly more points than every other candidate (no ties)? Clearly, sometimes this will be impossible, and other times, it is possible. For instance, in the example above, if you were the fourth voter, you could write $\langle A, B, D, C\rangle$ on the ballot, and now, A would have 8 points, B would have 7 points, C would have 6 points, and D would have 3 points, so A would win. On the other hand, if everyone else had ranked A last and B first, there would have been nothing you could do.

Give a polynomial-time algorithm to compute a ballot you can submit to make A win, or correctly conclude that no such ballot exists. Prove the correctness of your algorithm, and analyze its running time.

Answer:

**Note: We will not worry about ties for this question. A must strictly be the unique winner; if there is a draw between two candidates, the algorithm must conclude that there is no valid ballot, and thus, we treat drawing as losing.*

Borda Count Method: Assume that there are M >= 2 candidates and N voters. Each voter ranks all M candidates from most favorite to least favorite. We will denote the voter's most favorite candidate as X(1) and their least favorite candidate as X(m). A candidate gets M-1 points for each first place vote, M-2 points for each second place vote, and so forth: 0 points for last place. The points are summed and the candidate with the largest total of points wins.

**M (Candidates), N (Voters), X(1,2,3,...m) (Ranking of Candidates).*

As an example, assume three voters and four candidates. The rankings are as followed:

<A,B,C,D>, <B,C,A,D>, <C,D,A,B>.

Because M = 4 candidates:

Candidate A gets 5 points. (3 from first, 1 from second, 1 from third)

Candidate B gets 5 points. (2 from first, 3 from second, 0 from third)

Candidate C gets 6 points. (1 from first, 2 from second, 3 from third)

Candidate D gets 2 points. (0 from first, 0 from second, 2 from third)

*Since Candidate C has the highest total points out of the other candidates, C is the winner.

Assume that you are a voter N and you prefer candidate A, and you want this candidate to win. You have access to all the information of all other voters. **Is there some ranking you can put on your ballot such that A is the unique winner? (A has more points than every other candidate. Give an algorithm to compute a ballot you can submit to make A win, or correctly conclude that no such ballot exists. Prove the correctness of your algorithm, and analyze its running time.**

Such an algorithm that can compute if there is a possible ballot you can submit to make "A" win, involves a procedure that first computes the current rankings of every candidate, given that you have access to all the information of all other voters. Then you will hypothetically want to formulate a "cheating ranking" where you place your preferred candidate A as the first choice, and set the current winning candidate as the last choice. You would also want to formulate the rankings of your choice, to be listed in descending order of total points, such that the second highest ranked candidate (given that it is not A), is ranked second to last, and the third ranked candidate (given that it is not A), is ranked 3rd to last, and so forth. What this basically does, is that the algorithm computes the current ranking of where A stands in comparison to the rest of the competition/candidates, and gives A a certain "boost" of max points possible, and proceeds to distribute low points via your vote, to the other candidates that are currently in the lead. *This is totally unfair, but gives A the best chance to succeed.

However, before this algorithm even begins, it is true that for some inputs there is a ballot that makes A the unique winner, and other inputs where it is just not possible for A to win given the state of everyone's vote. To determine this even before the start of constructing a possible ballot,

we can compute the difference of ”points” between the current point leader and candidate A (the candidate you prefer). If this difference is greater than or equal to ”M-1”, M being the number of candidates, then it is not possible to form such a ranking of candidate that yields A as the highest ranking candidate.

Here we can form a proof:

If the difference between the current point leader and candidate A is greater than or equal to M-1, where M is the number of candidates, then it is not possible for you as a voter to form such a ranking of candidates to make A win the election. This is because M-1 points is the highest point value you can possibly assign to a candidate's pool of votes, and you only have 1 vote to give. Therefore, even if you were to give this candidate M-1 points to add, and hypothetically give the current leader 0 points by ranking him/her last in your preference ranking, the rankings will conclude in a draw, in which concludes that A **DOES NOT** strictly have more points than every other candidate, thus, there is nothing you can do. The algorithm will conclude that no such ballot exists.

To Formally State the Algorithm:

Initially, let R be the set of preference rankings excluding yourself, and let A be empty. Also let us declare an unordered Map called Map1. ”O(1)”

WHILE R is not yet empty ”O(n*m)” (M is the number of candidates, N is the number of votes)

We choose any arbitrary set from R, and sum the total points from the ranking given by voters for each candidate.

For each traversal through the arbitrary set, we take the position of the candidates in the set, and sum the respective point values corresponding to each Candidate and store this value in the unordered Map, Map1. If the Candidate doesn't exist, we insert the candidate and assign an initial total of 0, and sum accordingly. If exists, we sum the points. ”O(1)”

**By position in the set, I mean the first position (candidate) will get M-1 points, the 2nd position (candidate) will get M-2 points and there-forth.*

**We make the assumption that every voter votes for all candidates*

We add the set to A indicating that we have computed and added the points to each candidate and remove this arbitrary set from R. **This does nothing*

END WHILE

**At this point of the algorithm we have access to the current rankings of each candidate stored in an unordered map, and we can determine who is in the lead.*

IF A is in the lead, we construct any ranking/ballot that places A as the first preference to make A the winner. ”O(m)” (m being the number of candidates)

RETURN this preference ranking

**Algorithm is complete, A is a unique winner*

ELSE we iterate through the unordered map and search for the candidate with the highest total point count, and continue to iterate the same candidates and push into a priority queue (Min-Heap), except for Candidate A. ”*O(m) to iterate through map, and O(log m) to push into priority queue.

”This process of initializing Min-Heap takes O(m*log m)” *M being number of candidates.

We search for the number of points of candidate A easily through the property of hashing/unordered maps. We compute the difference between the highest candidate points and the point of candidate A. ”O(1)”

IF the difference is greater than or equal to M-1, where M is the number of candidates ”O(1)”

RETURN no such ballot exists

**We proved this above*

ELSE

We formulate a new construction of ranking that places candidate A as the first preference, and continues to add into this ranking, all the candidates from the lowest points to the highest from the Min-Heap. ”O(n)”. **Use the pop() function of heaps to get the lowest point candidates in order. O(1)*

**We want to formulate this ranking such that A gets the maximum amount of points, and the current winner gets 0 points. Also it is imperative that the second place current winner gets the second least amount of points, so that there will not be a case, where the second place winner can somehow be the first. O(n) to traverse through the new ballot.*

Recalculate the sums of points for all Candidates **Now including the new preference rank of yourself. ”O(m)”*

*This will serve as our implicit check/proof that A is the winner.

IF A is the unique winner of the calculation

RETURN BALLOT

**We successfully make A win.*

ELSE *This should never hit (but just in case)

RETURN no such ballot exists

**A is not the unique winner*

*End of Algorithm

Running Time: O(n*m) WORST CASE *M is the number of candidates, and N being the number of votes.

Proving Correctness:

This Greedy Algorithm Finds the Optimal Solution:

Here can use an ”exchange argument” and start with an optimum solution OPT, and gradually modify it until it is the same as the greedy solution. We want to make sure that no modification makes the solution worse. So what we end up with a solution that is at least as good as what we started with. So the greedy is at least as good as the OPT. So our greedy solution is optimal.

However, we can conclude that our algorithm is already correct since the algorithm checks it by itself already: it adds/recalculates the sum of points for all Candidates and adds up the existing tally

and only returns a ballot if it makes A be in the lead. Therefore, our algorithm generates and checks for a correct optimal solution. *Note that the question only asks to compute a ballot to make A win. That's all we are looking for and our algorithm coherently checks if the solution we are offering satisfies this case.

Here we must prove that if our algorithm returns no ballot, then there really is no possible ballot. Our algorithm only generates a "no ballot" solution if the difference between the highest total candidate and candidate A is equal to or greater than "M-1", M being the number of candidates.

We apply an exchange argument here.

Let there be 3 candidates, (A, B, C) and let the total points of sum be such that A = 7 points, B = 5 points, and C = 9 points. We can see that the difference between candidate C and candidate A is greater than or equal to M-1, therefore, the algorithm will produce a "No Ballot" solution.

Here let us assume this case of a "No Ballot" solution with an OPT solution and the solution that our algorithm produces.

Let us assume that the most optimal solution be: (A,B,C) that gives A a boost of 2 points, B a boost of 1 point, and C 0 points. More generally, we can extend the "most optimal solution" as a set of preferences that list A as the first candidate, and candidates there after.

Let our Greedy Algorithm produce "No Ballot Solution".

Here we find that we can make as much adjacent swaps as we want to the most optimal solution set to try every permutation/combination of A,B,C, or even with an extension with more candidates, but we will come to the conclusion that A,B,C is the best thing we can do for the sake of the candidacy of A. If we are to add the points to the previous total tallies, we get that A = 9 points, B = 6 points, and C = 9 points. However, even still by using this optimal solution, we get that A is not the unique winner; it is tied with C. This should be the case, and is a proof of contradiction that A,B,C (which is the most optimal solution) cannot be such a solution to this algorithm. Therefore, the greedy solution is at least as good as the optimal. The most optimal solution we can make is the greedy solution in the case that we produce "No Ballot Solution."

*We can use this same exchange argument idea for when it does produce a "correct" solution, however, our algorithm pretty much checks itself already (We only return the solution if A ends up being in the lead).

***To sum, the algorithm maximizes the increase in ranking of Candidate A, while minimizing the increase in ranking of everyone else.**

Question 2

Given how Covid is disrupting all of our lives, including this class, let's spend some time analyzing a computational problem related to the asymptomatic spread of Covid-19 and contact tracing. You are given a set V of n students (nodes), and a set E of m undirected edges between pairs of students. For each edge e, you are also given an open interval I_e, which denotes that for at least one moment in that interval, these two students were hanging out close to each other. For example, if you have an edge $e = (v_2, v_4)$ with $I_e = (7, 9)$, this means that students v_2 and v_4 were hanging out with each other at some time t with $7 < t < 9$. To keep things simple[2], we assume that there are never two edges/intervals for the same pair of

***NOTE* CLOSED INTERVAL INSTEAD OF OPEN INTERVAL (CHANGE)**

students — every pair met at most once.

We assume that if v_i has Covid by time t, and v_i and v_j hang out together[3] at time t, then v_j will immediately catch Covid, and be able to instantaneously spread it to other people they meet, and stay infected for the full duration under consideration.[4]

Suppose that at some point in the future, you find out that v_1 had Covid at time 0 (but are sure that no one else had Covid at time 0). We assume that no student had any meetings not recorded in the list[5], so the only way that new people can/will get infected is by meetings that were recorded with other students in V. You are student v_n, and you want to find out two things:

(a) Is it possible that you got infected, for some meeting times consistent with the given intervals?

(b) Is it possible that you did not get infected, for some meetings times consistent with the given intervals?

Notice that the answer to both questions could be simultaneously "Yes". As an example, if v_1 is the only one infected at time 0, and met v_2 at some time in the interval $(1, 3)$, and you met v_2 at some time in the interval $(0, 2)$, there are the following two scenarios:

(a) v_1 and v_2 met at time 1.414, and you met v_2 at time 1.618. You are now infected.

(b) v_1 and v_2 met at time 2.718, and you met v_2 at time 0.577. While v_2 did become infected, this was after you met them, so you escaped.

[Hint: To solve this problem (both parts), it may be very helpful to remind yourself of Dijkstra's Algrithm and its correctness proof.]

(a) Give a polynomial-time algorithm to correctly decide whether you could conceivably have *caught* Covid-19. If the answer is "Yes", your algorithm should also output the meeting times for each pair of individuals that make it possible.

(b) Give a polynomial-time algorithm to correctly decide whether you could conceivably have *escaped* Covid-19. If the answer is "Yes", your algorithm should also output the meeting times for each pair of individuals that make it possible.

As always[6], your algorithms should be accompanied by correctness proofs and at least a rough runtime analysis.

Answer:

**Note: We assume that there are never two edges/intervals for the same pair for the same pair of students – every pair met at most once. A student catches COVID instantly, if in close contact with another COVID positive student – there is no COVID incubation period. Students stay infected for*

the full duration – we ignore that not all meetings lead transmissions, ignore the incubation period of infection, and ignore the fact that one could recover and become non-infectious. We assume that the contact tracing list is 100 percent accurate, except for the uncertainty about the exact meeting time.

**Note, again we note that a student catches COVID instantly. Say a student X gets COVID at time T=2, and meets with a student Y at time T = 2. Student X will infect Student Y at this instance. (Definition of instantaneous asymptomatic spread.*

1. **1A.)** Given that an arbitrary student tests positive for COVID at some time T, and that you have a meeting with this said student consistent with the given intervals, **then it is possible that you got infected with COVID as well.** This is only true in the case that you meet with this arbitrary student at a time T **AFTER** the other student has COVID. For example, let student X test positive for COVID at time T = 1, and you have a meeting with this student from an interval T = [0,2]. After this meeting, you will only get infected with COVID, if the exact time of this meeting was after T >= 1, within the interval.
2. **1B.)** Given that an arbitrary student tests positive for COVID at some time T, and that you have a meeting with this said student consistent with the given intervals, **then it is also possible that you will not get infected with COVID as well**. This is only true in the case that you meet with this arbitrary student at a time T **BEFORE** the other student has COVID. For example, let student X test positive for COVID at time T = 1, and you have a meeting with this student from an interval T = [0,2]. After this meeting, you will not get infected with COVID, if the exact time of this meeting was before T < 1, within the interval.

Dijkstra's Algorithm

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
       d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d'(v)
EndWhile
```

**The algorithm maintains a set S of vertices u for which we have determined a shortest-path distance d(u) from s; this is the "explored" part of the graph. We want to find the shortest path from the infected node at T = 0 to yourself. It will be true that every adjacent vertex starting from the patient at T = 0, and therefore, it's adjacent nodes, and its adjacent nodes... will be infected. As long as it is a tree from the first patient node.*

**Through the textbook, it is proved that: For each edge produced by the algorithms execution, the path is the shortest path (proves correctness of Dijkstra's Algorithm).*

Give a polynomial-time algorithm to correctly decide whether YOU could conceivably have caught COVID. If the answer is "Yes," your algorithm should also output the meeting times for each pair of individuals that make it possible. (Include Correctness Proofs and Runtime Analysis)

1. Basically, the question is asking to give an algorithm to decide whether you are exposed to someone with COVID (students stay infected for the full duration under consideration), following a traversal of edges from patient infected at T=0, and if yes, output the exact time T where you are exposed to the asymptomatic student. We can achieve the answer by using Dijkstra's Algorithm, to find the shortest path from the student infected at T = 0, and traverse through all the nodes from the exposed students. However, instead of using distance as the criteria for weights, we use the earliest infection time (T) as the parameter.

   The reason why we use Dijkstra's, is because evidently, it must be true that every node that is infected, must somehow be related to the infection of the student at T=0, therefore, we start at this node. By simply following Dijkstra's Algorithm we follow the edges of earliest infection time of students (adjacent nodes). By using this intuition, we will follow the students in order in a consistent, chronological manner. However, it is not always the case, that there is one path that could conceivably give you COVID because there can be multiple paths from adjacent nodes to the target (yourself). The prompt asks to output the meeting times for **each pair** of individual that could. We make a few modifications to the general Dijkstra's to make sure to capture the earliest time of infection the edge we are traversing and the student node that we are hypothetically infecting. If the time intervals overlap, we mark the edge as an valid edge (infection could've occurred with that student). What we would do here in our modification of Dijkstra's is to update the node with the earliest possible time of infection. Basically this is a modified Dijkstra's with a change in parameters from distance to earliest time.

   To formally state the algorithm:

   *CSCI 104 Dijkstra's and Textbook

   Initialize an array called T[n] where N is the number of nodes in the search space. * *This array will store the earliest meeting time between the initially infected node and its neighbor.*

   Initialize P[n] where N is the number of nodes. **This array will store the predecessor nodes.*

   Initialize a 2D array C[n][n] where N is the number of nodes. **This 2D array will store the earliest time of infection between two nodes.*

   Initialize a Priority Queue named PQ to follow edges in respect to the earliest time of infection at which the current node can infect its adjacent neighbors.

   Initialize an unordered map, Map1. **To store the solutions.*

   Set T[u] where u is the start node to 0. **We make the starting node the initial infected person.*

   Add node u and T[u] to the priority queue.

   WHILE PQ is not empty

      We check the first element in PQ and remove from PQ. We will now process this element.

      IF the element is yourself, then this instance shows a possibility that you could've gotten COVID through a traversal from the intial node.

We add the key and value of this node to Map1, the key being the node that connects to yourself, and the value being the earliest possible time of infection T assigned. Then we skip this iteration, since we don't care about any other connections and do not need to add to the PQ.

For all neighboring nodes from this element.

IF the neighboring node has not been infected, then infect this nodes and store the earliest meeting time T between these two nodes in the 2D array. (We will have to compute this in our algorithm: simple arithmetic operations). We push this newly infected student in our PQ to explore later. *We know that this node has possibly been infected, so we must check its adjacent nodes next to find more possible infections.

**We achieve that Dijkstra's traverses through edges in a chronological order by using the PQ. We follow by the earliest infection time, which makes sense.*

We know from previous classes that the Runtime of Dijkstra's is O(n log n + M) *Where N is the number of students, and M is the number of edges.

Proving Correctness:

Proof By Induction (Very Similar to Proof of Dijkstra's)

***Proof is stated in the Algorithms Textbook Page 139-140. (Referenced and followed through with a few modifications) (Piazza @73)**

*Anything used in class can be used without re-proving. I am just re-using this proof because it can be applied to our instantiation of Dijkstra's algorithm.

*Idea of Dijkstra's is you have a subset of vertices where any nodes shortest path to the starting vertex is contained in the subset. And in this case, the subset contains all the shortest paths to the starting vertex. We want to prove/argue that adding a new vertex doesn't change the property at each step for the previous vertices. This should make sense because if the new vertex was part of the shortest path for one of the old vertices, it would have already been in your subset of earliest infections.

"Consider the set S at any point in the algorithm's execution. For each u in S, the path Pu is the shortest s-u path that sets the path to yourself as the earliest infection time of COVID."

Base Case:

If you are the initial starting node (Patient infected at T = 0), then it must be true that you infected yourself as early as possible. Proved.

We suppose the claim holds when true when there are more than 1 student (N nodes) for some value N>=1. We grow S to size N+1 by adding another student node.

By the inductive hypothesis, path Pu is the shortest path of infection to yourself for each u in set S. Now consider any other path P; we want to show that the earliest infection time is at least as short as Pv. In order to reach v, this path P must leave the set S somewhere; let Y be the first node on P that is not in S, and let x in set S be the node just before y.

We can conclude that P cannot have a shorter infection time than Pv because it is already at least as short as Pv by the time it has left the set S. In iteration N+1, the algorithm considered adding node y to the set via its edge and rejected this option because it found that another node that offered an earlier time of infection. This means that there is no path from s to y through x that has an earlier shortest infection than Pv. The textbook also says that the sub paths of P up to y is such a path, and so this sub path is at least as short as Pv. The full path P has the shortest infection time at least as Pv does as well.

Thus, we prove by using the proof by Dijkstra's algorithm that everything in set S produces a path that indicates the earliest possible infection time T. But instead of traversing through the shortest path, we are analyzing the "earliest infection time" as the weight, and saying that no such shortest traversal must come first before the shortest. Therefore, we show that all the paths in our set (output) are shortest paths.

Give a polynomial-time algorithm to correctly decide whether you could conceivably have escaped COVID 19. If the answer is "Yes," your algorithm should also output the meeting times for each pair of individuals that make it possible. (Include Correctness Proofs and Runtime Analysis)

1. Basically, the question is very similar to part A, asking to give an algorithm to decide whether you are exposed to someone with COVID, following a traversal of edges from patient infected at T=0, and if yes, output the exact time T where you are exposed to the asymptomatic student. However, instead of outputting the earliest time, we now want to output the intervals where there is no overlap. We can achieve the answer by again using our modification of Dijkstra's Algorithm, to find the path from the student infected at T = 0, and traverse through all the possible edges.

   To formally state the algorithm:

   Initialize an array called T[n] where N is the number of nodes in the search space. **This array will store the earliest meeting time between the initially infected node and its neighbor.*

   Initialize P[n] where N is the number of nodes. **This array will store the predecessor nodes.*

   Initialize a 2D array C[n][n] where N is the number of nodes. **This 2D array will store the earliest time of infection between two nodes.*

   Initialize a Priority Queue named PQ to follow the earliest time at which the current node can infect its adjacent neighbors.

   Initialize an unordered map, Map1. **To store the solutions.*

   Set T[u] where u is the start node to 0.

   Add node u and T[u] to the priority queue.

   WHILE PQ is not empty

We check the first element in PQ and remove from PQ. We will now process this element.

IF the element is yourself, then this instance shows a possibility that you could've gotten COVID. We add the data from this node to Map1, the key being the node that connects to yourself, and the value being the latest possible time T. Then we skip this iteration, since we don't care about any other connections and do not need to add to the PQ.

For all neighboring nodes from this element.

This is the change

IF the neighboring node has not been infected, and the current node is infected, then infect this node and store the **latest** meeting time T between these two nodes in the 2D array that these two could be infected. (We will have to compute this in our algorithm: again basic arithmetic). However, if the neighboring node is not infected and the current node can meet them at a time when they are both not infected, we would want these two students to meet at the **earliest** time possible when both nodes are not infected.

We push this node and the new corresponding earliest/latest time in our PQ to explore later.

**We achieve that Dijkstra's traverses through edges in a chronological order through using the PQ.*

We know from previous classes that the Runtime of Dijkstra's is O(n log n + M) *Where N is the number of students, and M is the number of edges.

Proving Correctness:

Proof By Induction (Very Similar to Proof of Dijkstra's and 2.a)

***Proof is stated in the Algorithms Textbook Page 139-140. (Referenced and follow through with a few modifications) (Piazza @73)**

*Idea of Dijkstra's is you have a subset of vertices where any nodes shortest path to the starting vertex is contained in the subset. And in this case, the subset contains all the shortest paths to the starting vertex. We want to prove/argue that adding a new vertex doesn't change the property at each step for the previous vertices. This should make sense because if the new vertex was part of the shortest path for one of the old vertices, it would have already been in your subset of earliest infections.

"Consider the set S at any point in the algorithm's execution. For each u in S, the path Pu is the shortest s-u path that sets the path to yourself as the **latest** infection time that does not give you COVID."

*This is very similar to the proof above, but now instead of traversing through he shortest infection time, we want to follow the "longest" infection time that a student could have possibly escaped COVID.

Base Case:

If you are the initial starting node (Patient infected at T = 0), then it must be true that you infected yourself as early/long as possible. Proved.

We suppose the claim holds when true when there are more than 1 student (N nodes) for some value N>=1. We grow S to size N+1 by adding another student node.

By the inductive hypothesis, path Pu is the shortest path of infection to yourself for each u in set S that **does not give you COVID**. We consider any other path P and want to show that the latest infection time is at least as short as Pv. In order to reach v, this path P must leave the set S somewhere; let Y be the first node on P that is not in S, and let x in set S be the node just before y.

We can conclude that P cannot have a shorter infection time than Pv because it is already at least as short as Pv by the time it has left the set S. In iteration N+1, the algorithm considered adding node y to the set via its edge and rejected this option because it found that another node that offered an earlier time of infection. This means that there is no path from s to y through x that has an earlier shortest infection than Pv. The textbook also says that the sub paths of P up to y is such a path, and so this sub path is at least as short as Pv. The full path P has the shortest infection time at least as Pv does as well.

Thus, we prove by using the proof by Dijkstra's algorithm that everything in set S produces a path that indicates the latest possible infection time T. But instead of traversing through the shortest path, we are analyzing the "latest infection time" as the weight that does not give a student COVID, and say that no such traversal must come first before the one that the algorithm traverses. Therefore, we show that all the paths in our set (output) are shortest paths.

*The difference between this algorithm and the previous algorithm is that we store different values as the "metric of time". For the first Algorithm we want to keep track of the earliest time of infection that a node could've possible gotten COVID. For the second algorithm we want to keep track of the latest time of an interaction of two students that could've possibly spread COVID, but also want to follow this in the "shortest path" finding algorithm.

*Again, I reference and use the Textbook for this proof as Piazza @73 suggests I can.