Midterm Examination

CS 4104 (Spring 2021)

Assigned: March 15, 2021.
PDF solutions due on Canvas by 11:59pm on March 26, 2021, i.e., you have 11 days.

Instructions

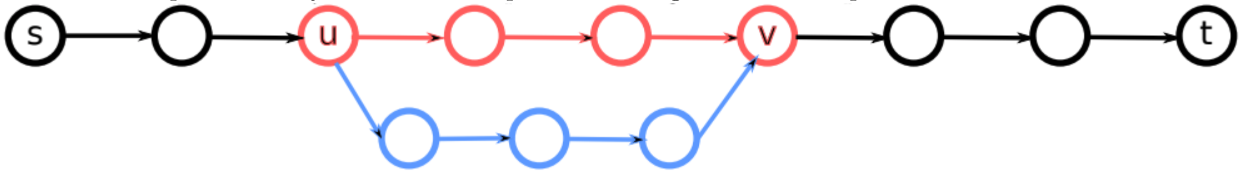
- The Graduate Honor Code applies to this examination. **Unlike in the case of homework, you must work on the examination individually.**
- You are not allowed to consult sources other than your textbook, the slides on the course web page, your own class notes, the TAs, and the instructor. In particular, do not use a search engine.
- Do not forget to typeset your solutions. *Every mathematical expression must be typeset as a mathematical expression, e.g., the square of n must appear as n^2 and not as "n^2".* You can use the LaTeX version of the homework problems to start entering your solutions.
- Do not make any assumptions not stated in the problem. If you do make any assumptions, state them clearly, and explain why the assumption does not decrease the generality of your solution.
- You must also provide a clear proof that your solution is correct (or a counter-example, where applicable). Type out all the statements you need to complete your proof. *You must convince us that you can write out the complete proof. You will lose points if you work out some details of the proof in your head but do not type them out in your solution.*
- If you are proposing an algorithm as the solution to a problem, keep the following in mind (the strategies are based on mistakes made by students over the years):
  - Describe your algorithms as clearly as possible. The style used in the book is fine, as long as your description is not ambiguous. Explain your algorithm in words. A step-wise description is fine. *However, if you submit detailed code or pseudo-code without an explanation, we will not grade your solutions.*
  - Do not describe your algorithms only for a specific example you may have worked out.
  - Make sure to state and prove the running time of your algorithm. You will only get partial credit if your analysis is not tight, i.e., if the bound you prove for your algorithm is not the best upper bound possible.
  - You will get partial credit if your algorithm is not the most efficient one that is possible to develop for the problem.
- In general for a graph problem, you may assume that the graph is stored in an adjacency list. If n is the number of nodes and m is the number of edges in the graph, then the input size is $m + n$. Therefore, a linear time graph algorithm will run in $O(m + n)$ time.

Good luck!

Problem 1 (15 points) Let us start with some quickies. For each statement below, say whether it is true or false. *You do not have to provide a proof or counter-example for your answer.*

1. Every bipartite graph is a tree.
   Solution: False
2. $\sum_{i=1}^{n} i \log i = \Theta(n^2 \log n)$.
   Solution: True
3. Dijkstra's algorithm may not terminate if the input graph contains a negative-weight cycle, i.e., a cycle such that the sum of the weights of the edges in the cycle is less than zero.
   Solution: False
4. In a directed graph, $G = (V, E)$, each edge has a weight between 0 and 1. To compute the length of the *longest* path that starts at u and ends at v, we can change the weight of each edge to be the reciprocal of its weight and then apply Dijkstra's algorithm. Here, the *length* of a path is the sum of the weights of the edges in the path.
   Solution: False
5. Suppose π is the shortest s-t path in a directed graph. Then, there can be two nodes u and v in π such that length of the portion of π connecting u to v is larger than the length of the shortest u-v path in the graph. Here π is the name I am giving to the shortest path between s and t; it is not the mathematical constant.

   In the figure below, π is the path composed of black and red edges. As an example, I have marked two nodes u and v in π and denoted the portion of π connecting u to v using red edges. The blue edges denote another path from u to v in the graph. The figure does not indicate edge costs. If you answer "yes" to this question, you are saying that there are graphs where the length of the red path can be larger than the length of the blue path (knowing that π is the shortest s-t path). If you say "no", then you are saying that for every pair of nodes u and v in π, the length of the red path is always less than or equal to the length of the blue path.



   Solution: False

Problem 2 (15 points) A connected, undirected graph G contains k cycles. Each edge in G has a distinct weight. Develop an algorithm to compute the minimum spanning tree of G in $O(mk)$ time. You may assume that $k < \log n$, so that the running time that I am seeking is faster than what you would obtain using Prim's or Kruskal's algorithm.

Solution:

The strategy is to process the edges in any order, add each edge to a set of edges representing the MST, check if adding that edge creates a cycle, and if so, remove the edge with max cost in the cycle using a modified DFS. Repeat this process k times, after which point we know that all the cycles have been handled and the remaining edges must be in the MST.

```
Algorithm 1: speedy_MST($G$)
   Input: $G$ a connected, undirected graph, $k$ the number of cycles in $G$
   Output: $T$ the set of edges representing the MST of $G$
   // Initialize an empty set of edges representing the MST
1  $T \leftarrow \{\}$
   // A counter to keep track of how many cycles have been processed
2  $c \leftarrow 0$
3  for edge $e \in G$ do
4      Add $e$ to $T$
       // Only check for a cycle if we haven't processed $k$ cycles
5      if $c < k$ then
6          $(u, v) \leftarrow e$
           // find_cycle is just a modified DFS which returns the list of edges
              comprising a cycle
7          $C \leftarrow$ find_cycle($T, u, v$)
           // find_cycle returns an empty list is no cycle is found
8          if $C \neq []$ then
9              k++
10             max_edge $\leftarrow e$
               // Remove the edge of max cost from $C \subseteq T$
11             for edge $i \in C$ do
12                 if cost(i) > cost(max_edge) then
13                     max_edge $\leftarrow i$
14                 end
15             end
16             Remove max_edge from $T$
17         end
18     end
19     return $T$
20 end
```

Proof of Correctness

Claim: T is an MST of G.

In order to prove that T is an MST, we must show that:

1. T does not contain a cycle.

   This is true by construction as the presented algorithm checks if adding and edge e to T creates a cycle, and if so, then precisely one edge in that cycle is removed from T.

2. T is spanning: it connects all vertices in G.

   This is also true by construction as all edges in G are added to T. Edges are only removed if they form a cycle in T.

3. T is Minimum: it connects G with the cheapest edges possible.

   This is also true by construction as it is only permissible to remove an edge from T if it forms a cycle, and in that case, the edge with the maximum cost $c(e)$ is removed.

As the presented algorithm conforms to these three criteria, T will be an MST of G.

Proof of Runtime Complexity

The main loop starting on line 3 of the algorithm accounts for $O(m)$ as it processes every edge in G.

We limit the number of remaining operations by the term k by only checking for a cycle in T if $c < k$. If we have not processed all the cycles in G, then the modified DFS, `find_cycle`, runs in linear time

$O(m+n)$, and the subsequent edge removal process (which can only occur k times) runs in less than m time.

Additionally, we can observe that the worst case scenario of an edge being added to T creating a cycle involving *all* the other edges in T can only occur once, so on average `find_cycle` will be faster than $O(m+n)$, since the edges and nodes involved in the modified DFS will usually be a strict subset of the E, V comprising G.

So, the inner loop(s) starting with the invocation of `find_cycle` on lines 7 and the edge removal loop starting on line 8, will take $O(m)+O(m+n)$ and will only be executed k times, for $O(k(m+n))$ time.

Therefore, the overall runtime complexity of the presented algorithm will be $O(mk)$.

Problem 3 (10 points) Consider the problem of minimising lateness that we discussed in class. We are given n jobs. For each job $i, 1 \leq i \leq n$, we are given a time $t(i)$ and a deadline $d(i)$. Let us assume that all the deadlines are distinct. We want to schedule all jobs on one resource. Our goal is to assign a starting time $s(i)$ to each job such that each job is delayed as little as possible. A job i is *delayed* if $f(i) > d(i)$; the *lateness of the job* is $\max(0, f(i) - d(i))$. Define

1. the *lateness of a schedule* as $\max_i \big(\max\big(0, f(i) - d(i)\big)\big)$ and
2. the *delay of a schedule* as $\sum_{i=1}^{n} \big(\max\big(0, f(i) - d(i)\big)\big)$.

Note that although the words "lateness" and "delay" are synonyms, for the purpose of this problem we are defining them to mean different quantities: the lateness of a schedule is the *maximum* of the latenesses of the individual jobs, while the delay of a schedule is the *sum* of the latenesses of the individual jobs.

Consider the algorithm that we discussed in class for computing a schedule with the smallest lateness: we sorted all the jobs in increasing order of deadline and scheduled them in this order. We proved that the earliest-deadline-first algorithm correctly solves the problem of minimising lateness. If we were to use the *same proof* to try to demonstrate that the algorithm correctly solves the problem of minimising delay, what is the first point where the proof breaks down? Explain why the proof is incorrect here.

Solution: The EFT proof breaks down when being applied to the problem of minimizing delay when jobs or sorted in terms of increasing deadline.

Consider the following two schedules as a proof by contradiction. The first schedule is one that is produced by the EFT algorithm:

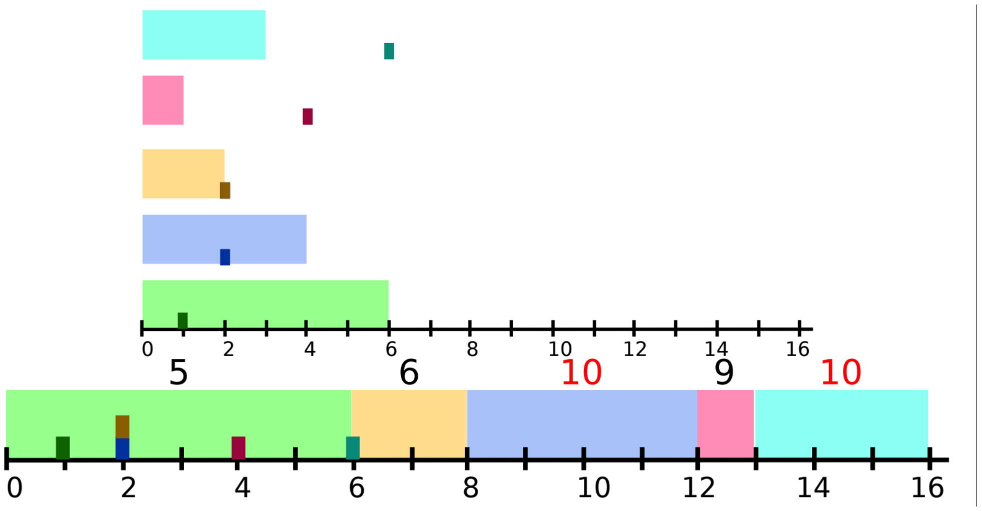


Figure 1: This EFT schedule has a lateness of 10, and a delay of 40.

And this second schedule produced by ordering jobs in increasing order of duration $t(i)$:

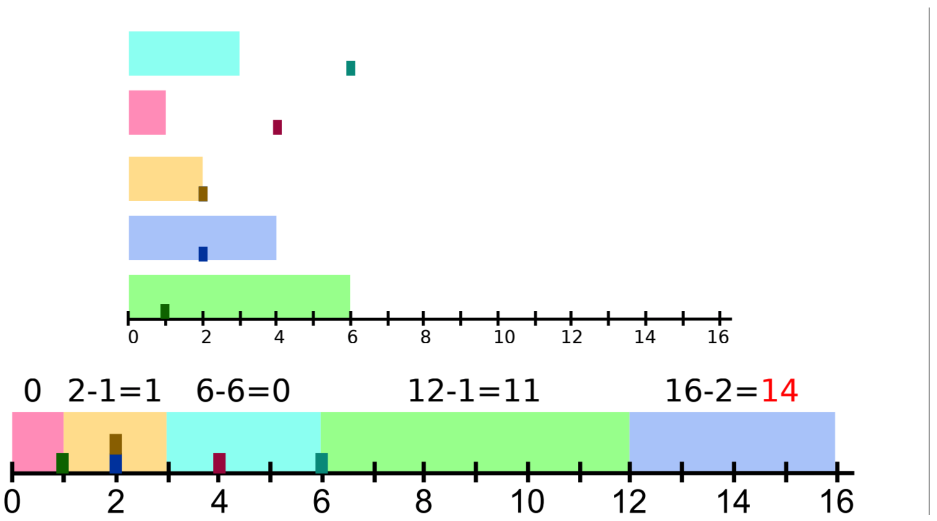


Figure 2: This schedule has a lateness of 14, but a delay of 26.

The proof breaks down due to its reliance on swapping inversions not increasing the overall lateness. This property of inversions with respect to finishing times does not hold for overall lateness.

Whereas the green job is inverted in terms of its finishing time with all of the preceding jobs (blue, tan, cyan) in the second schedule, and iteratively swapping it with those jobs does not increase the overall lateness of the schedule, the overall delay of the schedule does increase when fixing these inversions.

Because the proof of correctness for the EFT algorithm relies solely on the max lateness of all jobs and not the cumulative lateness of all jobs, the proof breaks down during the very first step of the algorithm which necessitates ordering the jobs by their deadlines.

Problem 4 (20 points) You are given a list of n real numbers (the list can contain both positive and negative numbers). Give an efficient algorithm to determine the contiguous sub-list with the largest sum. More formally, suppose the numbers are $l_1, l_2, \ldots, l_{n-1}, l_n$. Your mission, should you choose to accept it, is to compute two indices $1 \leq i \leq j \leq n$ such that

$$s(i,j) = \sum_{k=i}^{j} l_k$$

is the largest over all possible choices of i and j. Note that to compute $s(i,j)$ we sum up all the numbers between indices i and j inclusive.

Hint: I am looking for an $O(n \log n)$ time algorithm. It is possible that you have seen this problem before and are aware of a solution with an $O(n)$ running time. If you present this algorithm, it is even more essential than ever that you prove its correctness. This proof is quite complex. I will not award any points for an $O(n)$ algorithm without a complete proof of correctness.

Solution:

We employ a divide and conquer strategy similar to the solution for counting inversions in an array, or intersections between two sets of connected points forming line segments.

We recursively divide the array into two halves until we reach a base case with just a single element. Then we recursively "merge" sub lists, taking the indices and maximum value selected from either the left or right half of the sub list, or the sub list that spans both halves.

The tricky conquer step is the "merge" step where we find the largest contiguous sub-list which spans over two halves. This is accomplished by working out from the middle of the two merged halves.

Algorithm 2: `indices_max_sub_list`(S, lo, hi)

```
Input: $S$ the sub list, $lo, hi$ the indices in the sub list to be searched
Output: $sum, i, j$ the value and indices defining the range of the maximum sub list in $S$
if $lo = hi$ then
    return $S[lo], lo, lo$
end
mid $\leftarrow (lo + hi)/2$
// calculate values, indices for left half
l_sum, l_lo, l_hi $\leftarrow$ indices_max_sub_list(S, lo, mid)
// calculate values, indices for right half
r_sum, r_lo, r_hi $\leftarrow$ indices_max_sub_list(S, mid+1, hi)
// calculate values, indices for any sub lists spanning halves
s_sum, s_lo, s_hi $\leftarrow$ indices_max_spanning_sub_list(S, lo, mid, hi)
// Return the sum and indices corresponding to the maximum value of a contiguous
   sub list in $S$
return $\max_{sum}$ ((l_sum, l_lo, l_hi), (r_sum, r_lo, r_hi), (s_sum, s_lo, s_hi))
```

Algorithm 3: `indices_max_spanning_sub_list`(S, lo, mid, hi)

```
Input: $S$ the sub list, $lo, mid, hi$ the indices in the sub list to be searched
Output: The largest sum of contiguous elements in S, and the indices corresponding to this range
l_sum, r_sum $\leftarrow -\infty, -\infty$
sum $\leftarrow 0$
l_ind, r_ind $\leftarrow 0, 0$
for $i = mid; i \geq low; i--$ do
    sum += S[i]
    if $sum > l\_sum$ then
        l_sum = sum
        l_ind = i
    end
end
// reset the sum value
sum $\leftarrow 0$
for $j = mid + 1; j \leq hi; j++$ do
    sum += S[j]
    if $sum > r\_sum$ then
        r_sum = sum
        r_ind = j
    end
end
// Return the largest sum that spans outward from the midpoint
return (l_sum + r_sum), l_ind, r_ind
```

Proof of Correctness Claim: The indices corresponding to the largest contiguous sub array in the input list is found.

We begin by proof by induction:

Base case: the sub-list has a size of 1, the largest sub array is the only element, and the indices returned are the index of that element in the sub-list

Inductive Hypothesis: the presented algorithms find the largest indices and value of the contiguous sub-array in the list for all sub lists of size n.

Inductive Step: There are three cases to consider:

1. The list contains all non-negative values: the indices span the entire range of the sub list since the `indices_max_spanning_sub_list` finds the largest contiguous sum from the middle out, all values will be included/summed, and the resultant indices will be $0, n$.
2. The list contains all negative values: only one value, and its corresponding index will be returned as the `indices_max_sub_list` function will return the max of the three options between the max left, max right, and max spanning list, which will necessarily be the single largest number in the left or right half after the initial split.
3. The list contains a mixture of positive and negative values: The largest contiguous halves will be found by the `indices_max_sub_list` function, and the `indices_max_spanning_sub_list` function will search from the middle out, stopping once a negative value is found. This procedure will chain through the "merging" steps after each recursive call and once again, the indices corresponding to the maximum value will be found by searching from the middle, outwards.

As these two algorithms will recursively take the maximum value and the corresponding indices of the three available options, the resultant values are guaranteed to be the largest contiguous sublist and the indices marking that range.

Proof of Runtime Complexity

The lists will be split $\log n$ times in the `indices_max_sub_list` function, and in the worst case all values in a sub list will be traversed in the `indices_max_spanning_sub_list` function accounting for n operations to determine if two halves of a sub list contain a contiguous segment with a larger sum than either half individually.

Therefore, the running time complexity of this algorithm is $O(n \log n)$.

Problem 5 (40 points) This summer, you will be working at a synthetic biology company. The company specialises in creating cocktails of microbes (e.g., bacteria and fungi) to synthesize new antibiotics in an effort to combat infectious diseases. Their idea is to start with a compound that is cheap to make and to convert this compound into the desired antibiotic (which is another compound) using a series of chemical reactions that occur inside microbes. If you wonder why microbes are involved, know that several naturally occurring antibiotics are produced by microbes.

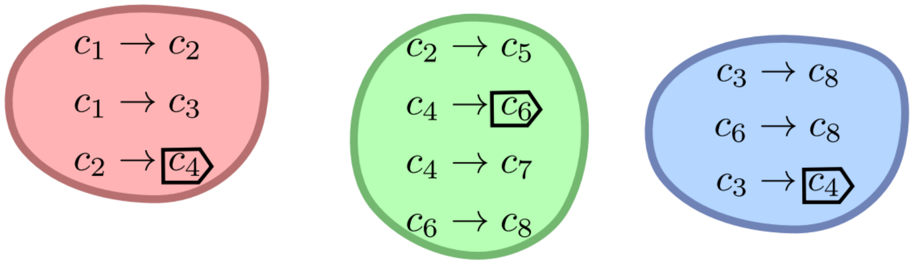


Figure 3: An illustration of microbes and reactions. The red microbe can perform three reactions, the green microbe is capable of four reactions, and the blue microbe has three reactions. A black shape surrounds a compound that can be secreted if it is made. Starting from c_1, the red and green microbes can together make c_8 since c_4 can be secreted by the red microbe and absorbed by the green microbe. Instead, if we started with c_3, then the blue and green microbes could synthesise c_8. (You may notice another way to make c_8 starting from c_1.)

Here is how the science works. A given species of microbe has the ability to synthesize specific compounds. The microbe does so using reactions that convert one compound into another. For example, in Figure 3, where the arrow sign means conversion, the red microbe can convert the compound c_1 into the compounds c_2 or c_3 (the first two reactions). It can also convert the compound c_2 to c_4. A microbe can also chain together reactions. Therefore, if the red microbe is provided the compound c_1, then it can make c_2, and thereafter use c_2 to make c_4. These chains of reactions can be arbitrarily long, as long as all the reactions can take place in that microbe.

Different microbial species can perform different sets of reactions and hence produce different sets of compounds. For instance, the only reaction that can use c_4 to form another compound exists in the green microbe. If we started with c_1 in the red microbe, produced c_4 in this microbe, and then somehow managed to "ship" c_4 to the green microbe, then we could obtain the compounds c_6, c_7, and c_8 (via c_6). And c_8 may be the antibiotic we desire!

How can we give the green microbe some c_4? Fortunately, biology comes to our rescue once again. Microbes can secrete some compounds to the environment and can absorb some compounds from the environment. For example, if c_4 is such a compound, then (starting from c_1), the red microbe can make and secrete c_4. Subsequently, the green microbe can absorb c_4 and use it to make c_6, c_7, and c_8.

Only a subset of all compounds can be secreted and absorbed. The reason is that each microbe acts like a closed bag (the solid boundaries in the figure denote the cell wall of the microbe). Therefore, in the figure, if the red microbe has c_1, then it can make c_2 and then c_4. Since the red microbe's bag is closed, c_4 should stay inside the red microbe. However, if c_4 has the right chemical properties and it can be secreted and absorbed, then c_4 will diffuse out of the red microbe and enter the green microbe. Inside the green microbe, c_4 can start new reactions that result in the production of c_8. As another example, consider c_3. Starting with c_1, the red microbe can make c_3. But c_3 cannot be secreted. If it were, the blue microbe could absorb it, giving an alternate path to c_8.

There are only two more rules. Each reaction takes some time to complete; different reactions can have different times. Secretion and ingestion of a compound also take time; these times vary from one compound to another.

What your company gives you as input are the following:

(i) the names of the microbes $M_1, M_2, \dots M_k$,

(ii) the set C of compounds,

(iii) for each microbe M_i, where $1 \leq i \leq k$, a set S_i of reactions and their times,

(iv) the subset $S \subseteq C$ of compounds that can be secreted and absorbed, and for each compound $c \in S$, a secretion time s_c and an absorption time a_c. Note that s_c may not be equal to a_c and different compounds may have different absorption/secretion times.

(v) a source compound $\sigma \in C$ and a target compound $\tau \in C$. You can assume that the starting compound σ can be absorbed by any microbe in zero units of time.

To be clear about item (iii), for each microbe M_i, a reaction in S_i is of the form (c, d, t). This triple represents a reaction where compound $c \in C$ is converted to compound $d \in C$ in time t units in the microbe M_i. This triple says that if microbe M_i contains the compound c, then it can convert it into compound d in t units of time. Therefore, a reaction indicates a possible transformation that a microbe can execute. In this case, M_i must somehow obtain c before it can convert it to d. If the microbe has no access to c, then the reaction from c to d cannot take place inside M_i .

The sizes of the sets S_i can vary from microbe to microbe.

Finally, we can state the problems the company asks you to solve:

1. Starting from compound σ, can the microbes produce the target compound τ? If the answer is yes, which set of reactions accomplishes this task in the least amount of time?
2. Solve the first problem in the special case when every reaction in every microbe takes one unit of time and for every compound in S, the total secretion and absorption time is two units.

Note: The problem is not difficult to solve using techniques you have learnt so far in this course. I am looking for clear descriptions of the approach and careful attention to the running time. There are many elements in the input. You have to decide what factors will govern the size of the input and, therefore, the running time of your algorithm. For part 2, I am expecting a faster algorithm than for part 1. And I request you again: please do not write code!

Solution:

1. We can solve this problem by modeling the components of the biological setting as a weighted, directed graph according to the following relationships:
   - nodes: compounds $c \in C$
   - edges: reactions within a microbe S_i, secretions and absorptions between microbes $M_i \to M_j$
   - edge weights: reaction times within a microbe also from S_i, the sum of secretion and absorption times $s_c + a_c$

   If we model the problem in this way, we can run Dijkstra's on the resultant graph to determine if there is a path (or chain of reactions) from $\sigma \to \tau$ by checking if $\tau \in S$ produced by Dijkstra's.

   If the compound can be produced, then we can find the shortest path from one compound σ to our target compound τ, where the edges along this path are the set of reactions (and necessary microbial secretions and absoprtions) needed to synthesize the target compound.

   Using this technique, the size of the input is governed by the the size of the compounds $C \equiv V$, as well as the edges comprised of reactions, secretions and absorptions: $\{S, S_1, S_2, ..., S_k\} \equiv E$. Therefore, the running time of the solution is the same Dijkstra's: $O(m \log n)$ with the number of possible reactions, secretions, and absoprtions being the dominating factor.

2. For this special case of nearly uniform edge weights, we can achieve linear runtime by splitting the edges modeled by secretions and absorptions into two edges of weight 1. Splitting the edges would take $O(m)$ time as we have to iterate over each edge and check if it connects two distinct microbes. If so, then we split the edge and add a dummy node in between the microbes that maintains the same connection as before.

   Once this step has been completed, we can simply perform a BFS on the modified graph since all of edges have the same weight, and therefore weights can functionally be ignored.

   The runtimes of our edge-splitting and subsequent BFS are $O(m) + O(m + n) = O(m + n)$, linear over the size of the graph, and better than the solution to problem 1.

The correctness of both of these approaches follows from the correctness of of Dijkstra's and BFS, respectively.

As we have modeled the first problem as a weighted graph with nodes and weights corresponding to the compounds and reaction times prioritized by the synthetic biology company, we can proceed by proof of contradiction.

There are two cases to consider:

Case 1: It is not possible produce compound τ starting from compound σ. Then the τ will not be in the set S returned by Dijkstra's.

Case 2: Let us assume that the path p representing the set of reactions, secretions, and absorptions (edges) to reach τ from σ is not the cheapest.

That is the distance $d(\sigma, \tau)$ stored in the Priority Queue is greater than $d^*(\sigma, \tau)$ the length of the optimal path. This is not possible since Dijkstra's is guaranteed to return the shortest distance from σ to all other reachable nodes in the graph, contradicting the proof of correctness for Dijkstra's algorithm.

Similarly, for the second problem, the same proof strategy can be employed, and even simplified when relying on BFS:

There are two cases to consider:

Case 1: It is not possible produce compound τ starting from compound σ. Then the τ will not be in the set S returned by BFS.

Case 2: Let us assume that the length of the path p representing the set of reactions, secretions, and absorptions (edges) to reach τ from σ is not as short as possible.

This is not possible either since BFS explores the connected component of the graph that σ is a part of, exploring outwards in layers. Therefore, if τ is in the same connected component of the graph G as our starting compound σ, then it exists in some layer L_i that will be explored on some iteration i of BFS. All of the compounds on layer L_{i-1} that can produce τ are equidistant from σ, and therefore the optimal path to τ from the starting compound will be found before BFS advances to another layer L_{i+1} (representing higher cost) that might contain a compound that can also produce τ.