

CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 10

For: March 23–March 27

Below is a proposed plan for tutorial for Week 10. In Week 10, we will continue talking about graph modelling, and continuing to look at shortest path problems. (This week we will mostly focus on problems that can be solved via Bellman-Ford, while next week we will talk about Dijkstra's algorithm.)

Problem 1. (Exploring all Paths)

Relevant Kattis Problem: <https://open.kattis.com/problems/beepers>

Sometimes, it seems like a good solution to a graph problem is to explore all possible paths in the graph. Starting from some vertex s , we explore all possible paths to some destination t , and examine the resulting cost. For example, imagine you have a highway road map, with tolls specified along various roads. You want to find the route with the smallest cumulative toll.

Perhaps we can explore all possible routes using a cleverly modified DFS or BFS? Why is this a bad idea? Draw a graph in which exploring all possible paths from s to t will not work well.

Solution: There are graphs in which there are an exponential number of possible paths! Draw one, and show that BFS (or DFS) works very poorly. It may seem like we have repeated this observation many times, but it remains a common mistake.

Problem 2. Overcooked

Relevant Kattis Problems:

- <https://open.kattis.com/problems/pickupsticks>
- <https://open.kattis.com/problems/reactivity>
- <https://open.kattis.com/problems/easyascab>



Figure 1: Overcooked. (Matthew Ng Zhen Rui)

After many years of playing Overcooked, you finally have fulfilled your dream of opening your own restaurant. On the opening day you are given a list of reviewers that will be coming. You

want to ensure that they are served in a timely manner so that they leave good reviews for your restaurant.

Problem 2.a. Firstly, you want to determine in what order you want to serve your dishes. There are a lot of dishes to serve out but you can't serve them in any order. Given a set of n dishes you know certain foods must be served after another (for example, you will serve appetizers like mushroom soup before the main course like filet mignon). Given a list of n dishes as well as k constraints on relative orderings of the dishes, output a valid sequence in which the dishes can be served. You may assume that such an ordering always exists.

In addition, you want to ensure that if there are multiple possible ways to order the dishes, you want output the one which is *lexicographically* the smallest (So that it looks presentable on a menu).

As an example, let's say there were 4 dishes: Garlic Bread, Mushroom Soup, Filet Mignon Burger and Banana Split. Now, if we are given the constraints as follows:

- Garlic Bread must be served before Filet Mignon Burger
- Mushroom Soup must be served before Filet Mignon Burger
- Banana Split must be served after everything else

These give the following valid orders:

- Garlic Bread, Mushroom Soup, Filet Mignon Burger, Banana Split
- Mushroom Soup, Garlic Bread, Filet Mignon Burger, Banana Split

However, we will want to output the former, since it is lexicographically smaller than the latter (since Garlic Bread is alphabetically before Mushroom Soup).

Optional : In order to protect against potential modifications of the order of dishes being served, you also want to check whether any valid ordering of the dishes is unique. Note that if this is the case then the valid order will instantly be the smallest lexicographically as well. How would you do this?

Solution: First we note that given the ordering constraints in the question, we can convert this into a directed graph where the nodes are the dishes and the edges are the ordering constraints. That is to say if we are given that we must serve dish a before dish b , then we shall draw a directed edge from the node representing dish a to the node representing dish b . Now we simply want to output a topological ordering of this graph. However, since we want to specifically output the lexicographically smallest ordering, we cannot use the DFS algorithm covered in the lecture. Hence we consider an alternative method, known as Kahn's algorithm.

The rough idea is as follows. We shall first calculate the in degree of every node by using the adjacency list. Now, since there is a valid topological ordering, there must exist a node that has zero in degree. (and there must also exist a node that has out-degree of 0) as well. What we shall start out with an empty list, which will eventually hold our topological ordering. First, we shall calculate the in degree of all the nodes. Then for every iteration, we shall add the node with zero in-degree into our topological order. If there are multiple such nodes, we shall add the one which is lexicographically the smallest. Once we add the node to our topological order, we shall look at the neighbours of this node to decrement the in-degree of that node. Then we again repeat this process until we process every node. For a normal implementation of Kahn's where we do not care about lexicographic ordering, we can do this in $O(V + E)$ time just by using a queue to hold all the zero degree nodes. However, since we need the lexicographically smallest zero degree node at each point, this will take $O(V \log V + E)$ time instead, where we use a priority queue instead of a normal queue.

If we want to check that the topological order is unique, we only need to check that there is a directed edge between every adjacent pair in the outputted topological order.

Problem 2.b. You quickly realise that based on your chef's schedules, that you actually need to precisely time when each dish will be served to the guests! The situation is the same as before, but you now also need to assign each of n the dishes to the n time slots there exists where each dish can be served. Based on how long a dish remains fresh after being prepared, you have determined intervals where each dish can be served, where each interval is a contiguous series of slots within 1 to n . (Example: Mushroom Soup must be served between slots 3 to 10, otherwise it will be too cold). All the intervals are inclusive of their endpoints. Note that you are also NOT allowed to serve two dishes at the same time. Hence, each timeslot will only allow to serve 1 dish. Determine whether you can give a schedule where each dish will be served, and if so, output that schedule.

Solution: This is slightly more tricky to handle since now we need to handle slots where the dishes can be placed in addition to ensuring that none of the relative ordering constraints are violated either. We also need to ensure that each dish has a slot that it can go into.

We can still solve this via a toposort however, with some extra book-keeping. We do however require a slightly different graph to model. Now, in addition to the n dishes, we will have $n + 1$ numbered nodes representing the slots that we are serving the dishes (the last slot is a dummy node). We shall represent that dish k is served during the interval $[i, j]$ by drawing a directed edge from the numbered node i to the node representing dish k , and another directed edge from dish k to the numbered node j . Of course, we want the numbered slots themselves to be outputted in the proper order, so we shall draw directed edges from node i to node $i + 1$ for $i \in [1, n]$.

Now, while performing the topological sort (using Kahn's again), we want to output the schedule by alternating between a numbered node and then a dish. Hence, we shall have the topological order in terms of numbered node 1, dish to be served in that slot, numbered node 2, dish to be served in that slot, etc. until the entire schedule is outputted. If at any point, we cannot add a numbered node to the topological order (if all the remaining numbered nodes do not have a zero in-degree), then we shall say such a schedule is impossible. Also, since we need to have one dish for every slot, if at any point, we need to add a dish and there are no dish nodes with zero in-degree, we again output that a schedule is impossible.

There is one last thing to consider here. If there are multiple dishes that have a zero degree node currently, which one do we output? In this case, we shall order priority based on the dish which has the earlier end time. (since every dish has a earliest to latest slot, we can use this as a tiebreaker). We can again use a priority queue here to hold the zero degree nodes for the dishes, where the priority is based on which dish is more constrained by the end time. This will give the required solution.

Problem 3. (Weighted Diameter of a Graph)

Relevant Kattis Problems: <https://open.kattis.com/problems/allpairspath>

The diameter of an unweighted graph is the maximum length “shortest path” between any two nodes. In a weighted graph, the weighted diameter is also the maximum length “shortest path” between any two nodes (but now the weights of the edges matter). Give an algorithm for finding the weighted diameter of a weighted graph.

Solution: One easy answer: run Bellman-Ford once for every node. That takes $O(V^2E)$ time. Now just look at every pair of source and destination and take the maximum distance.

Problem 4. (Internet Routing)

<https://open.kattis.com/problems/shortestpath3>

Imagine a hypothetical world where there are a set of computers connected by wires called *the internet*. Each of these computers has a unique address that is used for routing. (We might call this an IP address.) Each computer needs to maintain a routing table, i.e., a table that maps destination addresses to the next hop of the route. For example, if computer X has links to computers

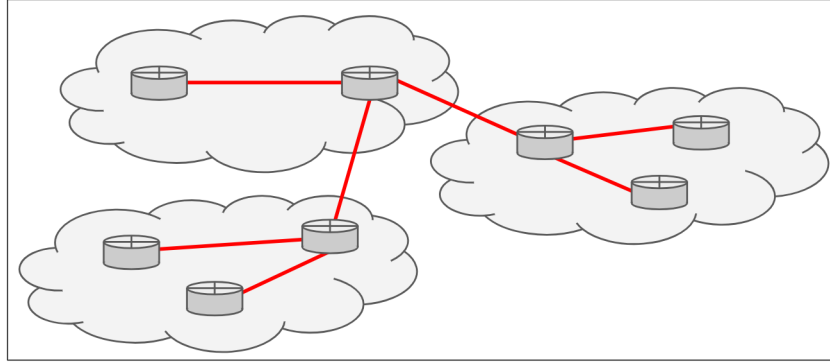


Figure 2: A Network Diagram

A , B , and C , and if it receives a message for destination D , the routing table might specify that that message should be forwarded to B . The goal is to forward the message to the destination using the minimum number of hops possible.

Problem 4.a. Collectively, the routing tables specify shortest path trees. How might you use **Bellman-Ford** to construct the routing tables?

Solution: The key observation is that we can construct the routing information for destination d by running Bellman-Ford starting at d . Each node in the network then learns its distance from d . At the same time, its parent in the BFS tree should be the next hop route for d . So we need to run n different Bellman-Ford algorithms.

Problem 4.b. Now imagine we want to construct the routing tables in parallel. That is, imagine that all the computers can send information to their neighbors at the same time. How might you implement a relax step so that all the computers can run it at once? Will that work? (Why or why not?) How long will the entire Bellman-Ford execution take?

Solution: When running n parallel Bellman-Fords, each node has a vector of distance estimate containing n estimates. In the relax step, it simply sends these estimates to all of its neighbors. If a computer has an estimate of t for some destination d , if it ever receives an estimate $t' < t - 1$ for destination d from some node p , then it updates its estimate for d to $t' + 1$, and updates its routing table entry for d to forward next to p .

If you think for a little bit about how and why Bellman-Ford work (and do some examples), you will see that all the computers can run the relax step at the same time. It may speed things up a little if you are lucky if the relax steps are run sequentially (but it may not). However, even if you run them all at the same time, then Bellman-Ford will terminate in D relax rounds where D is the diameter of the graph.

Problem 4.c. Imagine that the computers run Bellman-Ford relax rounds forever. E.g., every one minute they send their vector of estimates to their neighbors and update their routing tables

as necessary. Assume that a new edge is added to the network, e.g., A and B were not neighbors and now they are. Will the ongoing algorithm fix the estimates? If so, how long will it take? If not, what needs to be done to make it work?

Solution: In this case, it works just fine! We have just made some paths shorter, so within D more rounds of relaxation (where D is the diameter), everything will be correct again.

Problem 4.d. Imagine that the computers run Bellman-Ford relax rounds forever. E.g., every one minute they send their vector of estimates to their neighbors and update their routing tables as necessary. Assume that one edge fails in the network, e.g., A and B were neighbors and now they are not anymore. Will the ongoing algorithm fix the estimates? If so, how long will it take? If not, what needs to be done to make it work?

Solution: In this case, it is not immediately good enough. Relaxation can only make paths shorter, not longer! However, a simple modification will make it work: in each relax step, collect estimates from each of your neighbors and set your current estimate to 1 plus the minimum estimate. (Do this even if all the estimates are larger than your estimate.) Now, show that this works! Within D rounds (where D is the diameter), all the routing tables are correct again.

NOTE : Now some might ask since we are using an unweighted graph why can't we just run BFS from every node. There are few reasons for this. Firstly, Bellman-Ford is much more easier to implement because in the actual setting, what is done is that each router will calculate estimates of where the other routers are and send those estimates to other routers for them to update, while they receive updated estimates from each router. This is much closer to what Bellman-Ford is doing as compared to BFS, which requires a lot of backpropagation and is not convenient to maintain dynamic estimates. (They will learn more about this in CS2105, which is still compulsory for them!)

Problem 5. (Tourism)

Relevant Kattis Problem: <https://open.kattis.com/problems/maximizingwinnings>

You are off to travel the world in your trusty old beat-up Chevrolet. You have a map, a full tank of gas, and you are off. Just as soon as you figure out where you want to go. Looking at the map, you notice that some roads look very appealing: they will make you happy with their winding curves and beautiful scenery. Other roads look boring and depressing: they will make you unhappy with their long straight unwavering vistas.

Being methodical, you assign each road a value (some positive, some negative) as to how much happiness driving that road will bring. You may assume that the happiness is spread out uniformly over the entire road, for every road on your map. Pondering a moment, you realize that you can only travel a fixed number k kilometers. Starting from here on the map, find the destination that is exactly k kilometers away that will bring you the most net happiness. (Write out your algorithm carefully. There is a subtle issue here!)

Solution: First, let's negate all the happiness values so we are trying to minimize the net value of your route. Now this is just a shortest path problem: for each destination, find the minimum cost k -hop path to that destination.

The obvious thing to try is to run Bellman-Ford for k iterations. That almost works. But the problem is that depending on the order in which you relax the edges, you may find too short a path, i.e., your estimate for a given node may depend on a path of length $> k$. (Think, for example, what happens if you get lucky and relax the edges in the perfect order; then even in one iteration of Bellman-Ford you may find the real shortest path to a node, even if it is longer than k hops.)

Thus it is important to modify Bellman-Ford: in each iteration, first relax all the edges (using the estimates from the previous iteration, storing the result in a temporary array), and then update the estimates. In this way, each update depends only on the previous iteration, and you will find exactly the result of k -hop paths after k iterations.

Problem 6. (Spaceship Troubles)

The wonderful, fantastic product made by the Whoozit Company is a newfangled spaceship with a fancy warp drive. Now, warp drives are not quite as useful as you might think: they only let you travel between a set of fixed locations, along fixed (directed) paths. One of the possible jobs at Whoozit Company is *Guinea Pig*, and you have just been promoted. They hand you a map of the universe (complete with locations and directed paths), put you in the new spaceship, and off you go.

At first, everything works fine. You start off at the little dot marked “Earth”, chose one of the neighboring dots (“Alpha Centauri”, at only 4.24 light years away), and hit the big green *GO* button. Whoosh. You made it in one piece! Wow, each hop takes exactly 1 minute, regardless of how far you go!

But then a little red light starts blinking, and the screen reads out the following error message: **Error: your warp drive is now broken. Abort, Retry, Fail?** After some fussing with the computer, you discover that the warp drive still works, but it has the following limitation: it can only jump five hops at a time. You cannot just jump to a neighbor of Alpha Centauri; you have to jump directly to some location that is exactly five hops away.

Describe an algorithm that will get you from Alpha Centauri back to Earth in minimum number of hops, explain why it works, and give its efficiency.

Solution: There are two good solutions here, both of which run in $O(V^3)$ time. Assume your map is a graph $G = (V, E)$.

The simpler option (which follows something they would have learnt in CS1231) is to store the graph as an adjacency matrix and calculate G^5 via matrix exponentiation. This takes $O(V^3)$ time (for matrix multiplication), and the resulting graph tells you exactly where you can go in 5 hops. Now run BFS on this new graph.

An alternative solution that might appeal more to them is the following: Create a new graph $G' = (V, E')$ containing the same nodes as G , but a different set of edges: there is an edge between v and w only if there is a 5-hop path between v and w . How do we decide if there is an edge between v and w ? A simple way is as follows: first calculate $N(v, 1)$, all the 1-hop neighbors of v ; this takes at most time V (as v has at most V outgoing edges). Now calculate $N(v, 2)$, all the 2-hop neighbors of v . This takes at most time $O(V^2)$: each of the V nodes in $N(v, 1)$ has at most V outgoing edges, so we explore each and store (in a hash table) the set $N(v, 2)$. Similarly, since there are no repeated nodes in $N(v, 2)$, it has at most V nodes and we can calculate $N(v, 3)$ in $O(V^2)$ time. Continuing, we can calculate $N(v, 5)$ in $O(V^2)$ time. Now add an edge from v to every node in $N(v, 5)$. Now repeat this process for every node V . In total, it takes $O(V^3)$ time. Then, run BFS on the new graph, and find your way home. This takes slightly more space as compared to the previous solution, but the runtime would roughly be the same.

Problem 7. (Course Correction)

Remember when you first entered university and you were overwhelmed with all the different modules that were offered to you, hence leaving you confused as to how to do your module planning? Well now with the knowledge you have gained, you are better equipped to tackle this problem. After researching all the courses offered closely, you have extracted a set of courses that you wish to take, and have noted their required prerequisites. You are not allowed to take a course with one of its prerequisites in the same semester, and courses may have multiple prerequisites such that you are required to clear all the previous prerequisites before you can take that particular course. You may assume that there are no requirements for modules to be taken together, and there are no workload limits (i.e. you can take as many modules as you want in a single semester). Given this information, what's the minimum time you will take to clear all these courses?

Solution: Again, all the modules can be represented as a DAG (and we are assuming that this graph has no cyclic dependencies, which would be disastrous). Now, in order to find the minimum time required to clear all the modules, since we are not constrained by how many modules we can take in parallel, what we are effectively looking for is the longest path in this DAG.

The longest path in the DAG can be solved in two ways, both of which require us to perform toposort on the graph. Now, at this point, we look at the "sinks" of this graph (that have 0 out-degree). The longest path starting from these nodes will be 0. Then we can process the graph in reverse topological order, and for every node v , $d(v) = \max_{u \in N(v)} d(u) + 1$, where $d(v)$ is the longest path in the graph starting from this node, and $N(v)$ is the list of neighbours of node v .

This again will run in $O(V + E)$.