CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 12

For: April 3–April 7

Below is a proposed plan for tutorial for Week 12. In Week 12, 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 and Dijkstra's algorithm.)

1 Review Questions

Problem 1. (Kahn's Algorithm)

Note to tutors: Recommended time: 15—20 mins

Reference to Kahn's algorithm:
https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/

You have seen in lecture how Kahn's Algorithm work. The algorithm aims to find a topological ordering of a directed graph by removing nodes without any incoming edges, removing all edges originating from that node, and finally add that node to the ordering. This is repeated until there is no more nodes left without any incoming edges. What kind of data structure can we use to help us decide which node to remove next? What would the time complexity be?

Solution: First, a naive solution using AVL tree would take $O(E \log V)$

Realise that in Kahn's Algorithm, all we really need is to maintain a collection of nodes that have no incoming edges. Therefore, instead of storing all the nodes in an avl-tree/priority queue keyed by the number of incoming edges, we can simply use a queue to store the nodes that have no incoming edges. We know that each of the V nodes are added to the queue at most once, when there is no more incoming edges for that node. For each node, we are iterating through its edges, thus in total we are iterating through E edges. This then gives rise to $O(V + E)$ run time.

Problem 2. (Shortest Path in a Tree)
Suppose you have an arbitrary undirected tree with weighted edges, you want to find the shortest distance between a pair of nodes. Clearly you can use dijkstra, is there a simpler (and faster) way to do it?

Optional: What if you are given Q pairs of nodes that you want to find the shortest distance between. Clearly there is an $O(QN)$ solution. Can you do better? (There are multiple $O(NlogN + Q)$ solutions.

Solution: Since it is a tree, there is only one unique path between any two nodes. Using this we can root your tree from one node in the pair and just run BFS or DFS (both works), to get to your destination.
Answers to the optional extension (Addtional stuff for your interest). There are 2 different approaches to this solution. One uses centriod decomposition, the other requires us to find ancestors on a tree.
Approach 1: Centriod Decomposition. Here, the idea is that, suppose we arbitrarily root the tree. Then, we are able to answer queries with nodes that passes through the root. This refers to all pairs of nodes that are on different subtrees. To do this, we run one BFS from the root and the answer is just the 2 distances from the root added together. This allows us to answer for all queries that passes through the root. Then, we can now cut the tree up into the subtrees and recurse (ie solve for pairs that are strictly contained within one subtree). In the worst case scenario, this is still $O(N^2+Q)$ time. However, we can choose our root in a clever way. If we choose the root such that no subtree is more than half of the size of the entire tree then the recursion will just be $O(logN)$ deep. To do this, we run a BFS from an arbitrary root, keeping track of subtree sizes, and then keep going down the children until there exist no children with more than half of the total size. Thus, we get a time complexity of $O(NlogN+Q)$. For more information: https://codeforces.com/blog/entry/81661
Approach 2: Finding Lowest Common Ancestor
Similar to before, we root the tree arbitrary at r. Then suppose we want to find distance between u and v. Then, we can run DFS from r to everyone else to get all distances from r. Thus, we have $d(r,u)$ and $d(r,v)$. Notice that, u and v will share the same path until a divergence, which will be their lowest common ancestor say a. Then $d(u,v) = d(r,u) + d(r,v) - 2d(r,a)$.
Now, how do we find the lowest common ancestor? There are 2 ways. One is to use a technique called binary lifting. In essence, you store every power of 2 parents of every node. (ie, your parent, your grandparent, your 4th parent, 8th parent etc). Then, now we have the tool to find an arbitrary k-th parent in $O(logk)$. We can then use binary search to find the LCA for $O(log^2 n)$. There exists a cleaner way to find LCA in $O(logn)$ though by realising you can do the binary search while trasversing up at the same time. More details: https://cp-algorithms.com/graph/lca_binary_lifting.html
Another way is to use euler tour(which is similar to pre and post order traversal combined) as well as a Range Min Query Data structure (AVL Tree works but there exists other data structures like a Sparse Table or Segment Tree). If you use a sparse table you can get a time complexity of: $O(nlogn+Q)$You can read the details here: https://www.geeksforgeeks.org/find-lca-in-binary-tree-using-rmq/

2 Problems

Problem 3. Overcooked

Note to tutors: Recommended time: 10—15 mins

Relevant Kattis Problems:

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

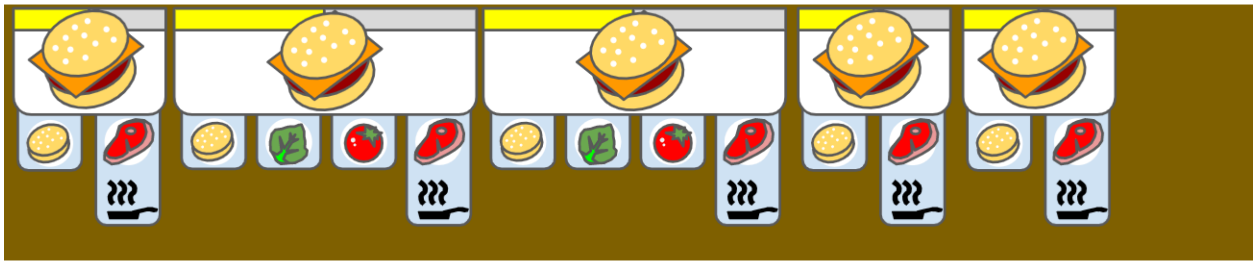


Figure 1: *Overcooked.* (Matthew Ng Zhen Rui)

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.

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 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. This can be done in $O(V)$ using an adjacency matrix or $O(V + E)$ time using an adjacency list.

Problem 4. (Tourism)

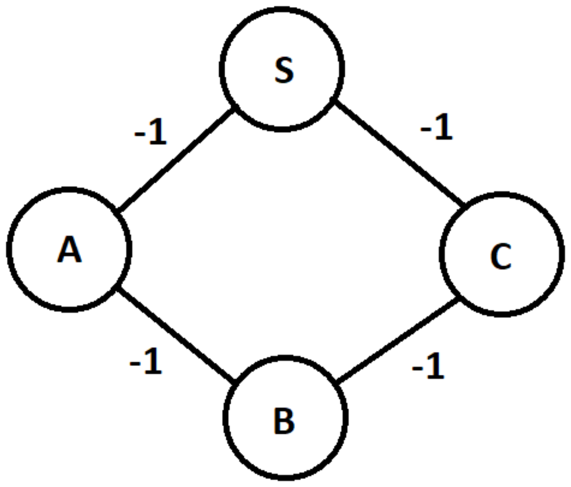
Note to tutors: Recommended time: 10—15 mins

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. You may also assume that every road spans a distance of 1 kilometer. 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 at most 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.)



Consider the graph above, we will start at S and set $k = 1$, so we will perform 1 iteration of Bellman-Ford. Let's relax the edges in the order of $(S, C), (S, A), (A, B), (B, C)$. Once we are done, realise that node C will have the cost of -3, which in turn gives our highest happiness. But that is from a 3 hop journey. Worse still, node B's cost has also been updated, but it can only be reached in 2 hops by looking at the graph. So how do we avoid this?

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 5. (Strange Car)

Note to tutors: Recommended time: 15—20 mins
This problem aims to introduce to the students the concept of augmenting the graph, focusing on the technique of duplicating the nodes.

At this point, you are probably well acquianted with the shortest path problem. Now, let's shake things up a little.

You have finished work and want to go home. Unfortunately, some guy decided to trash your car and it no longer works. But fret not as your friend however decides to lend you his car. So your problems are solved right?

Well, not quite. As you start the engines, there is a strange note written on the door. It states that the car can only be parked after moving every 5 kilometers! Indeed the odemeter is currently a multiple of 5! If you have not travelled a distance exactly a multiple of 5, the doors will not unlock and you will be stuck until the odemeter is a multiple of 5!

Nonetheless there might still be hope. The town you are in is strange. Buildings are connected by unidirectional roads exactly 1 kilometer long. How long then will you take to reach your home AND be able to stop? (You can visit your home multiple time before being able to stop).

Optional Extension 1: Now, what if the care can only be parked after moving arbitrary k kilometer, here k can be large.

Optional Extension 2: Now, the roads are bidirectional BUT are of arbitrary length (but are of integer kilometer length). However, you are sure that the total length of the all roads in kilometer is small. What if k is still large? (Adapted from: `https://open.kattis.com/problems/kwalks`*)*

Solution: The first two solutions both run in $O(V^3)$ time, and the third solution runs in $O(V + E)$. 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 5km. 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.

However, there exist a solution that runs in $O(V + E)$ time and this can be achieved by creating 5 copies of the original graph, G_0, G_1, G_2, G_3, G_4, each with the same set of vertices as G, but a different set of edges. For every edge (a, b) in the original G, draw an edge from node a in G_i to node b in $G_{(i+1)\%5}$ for all $i = 0, \ldots, 4$. Now, we run BFS starting from the node representing your workplace in G_0 and return the shortest distance to your home in G_0.

Solution to Extensions: For extension 1, you must use the first solution presented. Notice that you can do matrix exponentation in $O(V^3 logK)$. (Recall how you power numbers in CS1101S).
For extension 2, if k is small, you can just use the previous methods provided where you create copies of nodes. However, if k is large, you must use the first solution with some modification. First, you break your bidrectional road into multiple directed small roads. For example, there is a road of 7km between A to B. You will first create 12 dummy nodes. Then you will connect from A to B with 6 dummy nodes in between with roads of 1km long and do the same from B to A. Notice, you cannot just use 6 nodes as that will create a graph and allow you to go back and forth in the old road, which is not allowed.

Problem 6. (Roads with discount)

Note to tutors: Recommended time: 15—20 mins
Another augment the graph problem, this time introducing new nodes.

Adapted from: NOI 2022 Finals Task 1 (Voting cities)
https://codebreaker.xyz/problem/votingcities

You have a country with multiple cities connected by unidirectional roads, and travelling along each road will cost a corresponding amount of money in bus fare. Some of these cities are good cities. You want to get from a city A to any good city in the cheapest way possible. At the start of your journey, you are given the choice to buy 5 different vouchers. They can be used on one road per voucher, reducing its costs by $10\%, 20\%, 30\%, 40\%, 50\%$ respectively. You cannot use multiple vouchers on a single road to stack effects. You can choose to buy all 5 vouchers, or none at all.

Alas, you do not know your starting city A, nor the prices of the vouchers. As such, you thought of Q different scenarios. For each scenario, you start at a different city and are given different voucher prices, find the cheapest possible way to get to a good city. (Note that the cost of bus fares doesn't change in each of the scenarios, e.g. it will always take 3 dollars to travel from city A to city B regardless of which city you start from)

Problem 6.a. For this part, let us consider the subproblem where there are no vouchers. How would you then answer the Q scenarios effectively?

Solution: Of course, there is a naive solution of just running SSSP Q times for each scenario and taking the cheapest good city each time. This will give us $O(QElogV)$ time complexity. However, we can do better. Instead of starting from the starting point, why don't we start from the good cities instead. Let us consider just one good city, if we can find the preprocess cheapest route from every other city to that city, we can answer each query in $O(1)$ time. To do this, we just reverse the edges, and do SSSP from that good city. The reason we reverse the edges is because, we are now travelling in the opposite direction. Now, because we have multiple good cities. We can do the standard multisource trick. We can just create a super-good city and connect it with each of the good city with road of cost 0 and run SSSP from that super good city. Then, once we do that, we can answer each query in $O(1)$ time. Time complexity is thus $O(ElogV + Q)$

Problem 6.b. Now, we will consider the full problem with vouchers involved. How would you answer this question? Hint: Transform the graph, perhaps we can keep track of which vouchers we have used already somehow?

Solution: We can transform the graph as follows. For each city, we can create 32 copies of it. Each of them is associated with a array of 5 booleans which indicates which vouchers has already been used. For example, we could have the first copy being associated with array [F, F, F, F, F], where none of the vouchers have been used, and the third copy with [F, F, F, T, F], where only the second voucher have been used. We create 32 copies as there are 32 possible combinations of voucher usage. Now, then we add the edges between them. When we use a voucher, we reduce the cost of the road, and it will go from one copy of the map to another copy with the array indicating that the voucher has been used. We then do the similar edge reversal and super city trick in the previous part. Then, to answer each scenario, we go through 2 copies of the map, specifically the one with all vouchers used and the one with no vouchers used, and find the cheapest way from there.

For time complexity analysis, let us be generic and consider the number of vouchers as T. Then, we would have $O(2^T V)$ vertices. The original edges are duplicated 2^T times as well, but now there are $O(T)$ additional extra edges which denotes a used ticket. Thus, we would have $O(2^T TE)$ edges. Each scenario would take $O(1)$ time to answer. This gives us the final time complexity of $O(2^T TElog(2^T V) + Q)$.

Extra: This is a possible WRONG solution that most people may come up with. Once, we found the shortest path from the starting city to the good city, we greedily use the best voucher on the most expensive road if it is optimal. This however will not give us the best solution. Consider, you have found a route that goes take 5 roads of cost 9,9,9,9,9. For simplicity, assume the vouchers are free. If we use all 5 vouchers, we will get 31.5 total cost. However, perhaps there is another route that take 1 road of cost 50 which is more than the original 45. However, we can use the 50% voucher on it to get a cost of 25 which is better. This gives us a counterexample.

Fun fact: This is the easiest question in NOI finals of 2022. Quite a number of high schoolers are able to solve and implement a working solution without bugs from sratch in a span of 5 hours.

Problem 7. (How many arrays, hard)

Note to tutors: Recommended time: 15—20 mins

Adapted from: https://open.kattis.com/problems/permutationarrays

Problem 7.a. You are given an unknown array A of length n and a few facts about it. You know that all the elements are a number from 0 to $2^N - 1$. You also are given m constraints of the following tuple: (l, r, k). This means that $A_l \oplus A_{l+1} \ldots \oplus A_r = k$ ($\oplus$ is the XOR operator, e.g. $5 \oplus 4 = 1$. If you are unsure how this operator works, try converting the numbers to binary forms first and XOR bit by bit). You want to find out how many different arrays (or none) satisfies your constraints.

First Hint: Instead of considering the number of arrays, would it be easier to consider the number of prefix XOR arrays? How would you convert the constraint from the original array to this new array and what does the number of prefix xor arrays tell you about the number of original arrays?

Second Hint: Try to model this as a graph where the goal is to find how many ways you can assign values to nodes. What would be the nodes? How about the edges?

Note that we define the prefix XOR array B to contains $n+1$ entries, where entry $B_0 = 0$ and $B_i = A_1 \oplus A_2 ... \oplus A_i$.

Solution: We define an Array B to be the XOR prefix of array A. A key observation is that there exists a bijection between sets of A and B, which we define as the XOR prefix of A. For example, if $A = [3, 4, 5]$, then $B = [0, 3, 7, 2]$. The rigorous mathematical proof for the Bijection is left to the reader. As such, if we know how many different B there is, that is our desired answer. Now, we notice that the constraints will then be converted to $B_{l-1} \oplus B_r = k$ or $B_{l-1} \oplus k = B_r$. This suggests that for every constraint (l, r, k), knowing the value of B_{l-1} will give us one unique answer for B_r that satisfies the constraint. We can then model this problem as a graph.

Each B_i is a node and each constraint is an edge with weight k, connecting B_{l-1} with B_r. Notice that for each connected component, if one of the node is determined, the other nodes can be uniquely determined. As such, apart from the 1st component that consists of B_0 which is 0, we have 2^N possible ways to assign values to the nodes of each connected component. After finding the number of connected components, the answer could be $2^{N*(CC-1)}$. However, we are not done. The constraints may not be consistent. For example if we are given constraints (1,2,3), (3,4,4) and (1,4,0), they are inconsistent (consider $A_1 \oplus A_2 \oplus A_3 \oplus A_4$, which gives us 0 based on constraint (1,4,0), but gives us $3 \oplus 4 = 7$ based on the constraints (1,2,3) and (3,4,4)). Thus, we need to check consistency. For each connected component, we assign the first node as 0 and then run BFS or DFS to determine the other nodes value. Note that we do not visit the node twice. Then, we compare the array we get with the constraints. If any of the constraints does not match the array, the answer is 0. This idea of checking is actually similar to how we check if the graph is indeed a directed acyclic graph after doing toposort by cross referencing the order the nodes with the edges. Time complexity is $O(m + n)$.

Problem 7.b. (Optional)

Note to tutors: This is an optional extension to the problem above and may be rather challenging conceptually. Tutors can feel free to skip it if they are not confident that the class can understand it or save it for future tutorials. However, I believe it features a clever use of small to large merging or binary search.

What if now, you are given the constraints one by one? At the start, you have no constraints. Everytime you receive a constraint, you want to know at present moment, how many arrays satisfies all previous constraints you have seen. Of course, we could naively run the above algorithm from scratch everytime we receive a constraint, but can we do better?

Solution: At the start, we can assign each node a value of 0. Then, everytime we receive a new constraint, it can either be in 2 cases. First, the new constraint is an edge within the same connected component. In this case, we just need to check for consistency. If consistency is preserved, the answer remains unchanged. If not, the answer will become 0, as consistency has been broken. Notice that once a graph is inconsistent, it will forever remain inconsistent. Second, the new constraint can be connecting 2 different components. In which case, we need to update the values in the nodes of one of the component. If you pick the node at random, this will potentially take $O(n)$ time all the time. However, we can be clever. We will just modify the values of the smaller set of nodes. Notice that, in total, over the whole algorithm,this will take at most $O(n)$ time, as the size will at least double of the smaller set from the last merging. This trick is similar to that from a prevous tutorial. Of course there are other alternatives in which you can modify the usual Union Find algorithm and augment values which you can explore. Time complexity: $O(m+n)$

One solution that I would also like to point out is something to do with binary search. This is slightly less efficient and would only work if the problem can be processed offline and just the m numbers need to be reproduced after receiving all the output. This is commonly known as offline query processing. In this solution, we notice that the moment we recieve a constraint that breaks graph consistency, the answer is always 0. This motivates a binary search approach. Notice that, we can thus binary search for the exact constraint in which after that, the graph becomes inconsistent. After that, we can just use Union Find to keep track of how many connected components there are, and once we receive that constraint, we just output 0. While slightly less efficient, this solution is much easier to implement. Time complexity: $O((n + m)logm)$.

Extra Notes: Notice, that this question would work for any group, instead of just integers under XOR. The group need not be abelian/commutative as with the case of integers under XOR. The group properties of inverses and associativity is enough to make the question work. In the original question, the question uses permutation groups which makes the question slightly harder, especially since permutations are not commutative, so the implementation is abit tricky.