CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 11

For: March 27–March 31

Problem 1. DFS/BFS

Problem 1.a. Recall that when performing DFS or BFS, we may keep track of a `parent` pointer that indicates the very first time that a node was visited. Explain why these parent edges form a tree (i.e., why there are no cycles).

Solution: A total of $n - 1$ new nodes will be discovered starting from the source node. With each discovery, the `parent` pointer can be thought of as an edge. The final structure will consist of n nodes and $n - 1$ edges and will form one connected component. The resultant tree is sometimes known as a BFS/DFS-spanning tree.

Problem 1.b. Do you remember learning about the BFS and DDS algorithm for trees? Turns out, trees are just a type of graph. How does BFS or DFS on a tree relate to BFS or DFS on a graph? Are they the same algorithm? Will the BFS/DFS algorithm for trees work on a graph? Likewise, will the BFS/DFS algorithm for graphs work on a tree? What happens if you run BFS/DFS on a tree but do not start at the root?

Solution: The version for graphs is a generalization of the version for trees. For trees, we did not keep track of which nodes were already visited? Why? Becauase it was a tree! Since there were no cycles, it was impossible to visit a node twice. So the version for graphs will work on trees, but the version for trees will NOT work on graphs.

If you run BFS/DFS on a tree starting from a non-root node, it works just fine. In fact, you can use a BFS or DFS from a non-root node on a tree to "re-root" the tree, specifying a new root for the tree (and updating which are the parent and child edges). You might go through an example of this, i.e., how to switch the root of a tree.

Problem 2. Graph components

(Relevant Kattis Problem: https://open.kattis.com/problems/countingstars)

Given an undirected graph $G = (V, E)$ as an adjacency list, give an algorithm to: (i) determine if the graph is connected; (ii) return the number of connected components (CC) in the graph.

Solution: For part (i), if the graph is connected, it means that we can pick any node and run BFS/DFS from it, and every other node must be reachable from that starting node.
For part (ii), for every unvisited node, run BFS/DFS from that node and flag every visited node as visited. All of the visited nodes form one connected component. Search for the next unvisited node and repeat the process. If there is exactly one connected component, the graph is said to be connected.

Problem 3. Is it a tree?
(Relevant Kattis Problem: https://open.kattis.com/problems/flyingsafely)

Assume you are given a connected graph with n nodes and m edges as an adjacency list. (You are given n but not m; assume each adjacency list is given as a linked list, so you do not have access to its size.)

Give an algorithm to determine whether or not this graph is a *tree*. Recall that a tree is a connected graph with no cycles. Your algorithm should run in $O(n)$; particularly, it should be independent of m. Assume $O(n + m)$ is too slow.

Solution: Any connected, undirected graph containing n nodes and $> n - 1$ edges has a cycle. Thus, simply count edges in the graph, stopping when you find at least n edges. Notice, though, that since the graph is given as an adjacency list, each edge is represented twice: in the list of both the source and the destination. Thus, if you find $> 2n - 2$ edges in the adjacency list, you know there is a cycle, and the graph is not a tree. Otherwise, since we are given that the graph is connected, and have verified that it has $n - 1$ edges, it is a tree.

If we are not given that the graph is connected, the same approach can be used. After verifying that it has $n - 1$ edges, simply modify it to run DFS to check whether all nodes can be reached and that there is no cycle.

Problem 4. Graph modelling

Here are a bunch of problems. How would you model them as a graph? (Do not worry about solving the actual problem. We have not studied these algorithms. Just think about how you would model it as a graph problem.) Invent some of your own problems that can be modeled as graph problems—the stranger, the better.

Problem 4.a. Imagine you have a population in which some few people are infected with this weird virus. You also have a list of locations that each of the sick people were in during the last 14 days. Determine if any of the sick people ever met.

Solution: Model the people as nodes and the locations as nodes, and add edges between people and the locations they were in. Now look at any location node that has degree > 1.

Problem 4.b. Imagine you have a list of cities. Some pairs of cities are also connected to each other by roads, however, these roads are pay-per-use and one must pay a certain toll (different for each road) to travel on that road! Find the cheapest way to travel from city A to city B.

Solution: Model the cities as nodes, and add edges between cities if there is a road connecting the two cities. The weight of each edge is the toll for that road. Search for the path between cities A and B that incurs the lowest total toll! (This is referred to as a (weighted) shortest-path problem!)

Problem 4.c. You are given a set of jobs to schedule. Each job j starts at some time s_j an ends at some time t_j. Many of these jobs overlap. You want to efficiently find large collections of non-overlapping jobs so that you can assign each collection to a single server.

Solution: Each node is a job. Add an edge between two nodes if the respective jobs overlap. Now you need to find collections of nodes that are not neighbors. (This is referred to as an *independent set* and hence can be solved by algorithms for finding a maximal independent set. In fact, there are simple greedy solutions for the Interval Scheduling problem.)

Problem 4.d. An English professor complains that students in their class are cheating. The professor suspects that the cheating students are all copying their material from only a few different sources, but does not know where they are copying from. Students that are not cheating, on the other hand, all submit fairly different solutions. How should we catch the cheaters?

Solution: Each student's essay is a node. Add an edge between two nodes if the respective essays are similar. A cluster of nodes all connected to each other likely indicates cheating. (If they are all connected, this is about finding a clique. Otherwise, this is a clustering problem.)

Problem 4.e. There are n children and n presents, and each child has a list of presents that they want. How can we assign presents to children so that each child receives a present that they

want?

Solution: Each child is a node. Each present is a node. Add an edge connecting a child to a present if that present is acceptable to that child. Now find a set of edges that do not share any endpoints. (This is called a matching.)

Problem 5. Word games
(Relevant Kattis problem: https://open.kattis.com/problems/sendmoremoney)

Consider the following two puzzles:

- Puzzle 1:

```
   S E N D
+  M O R E
----------
 M O N E Y
```

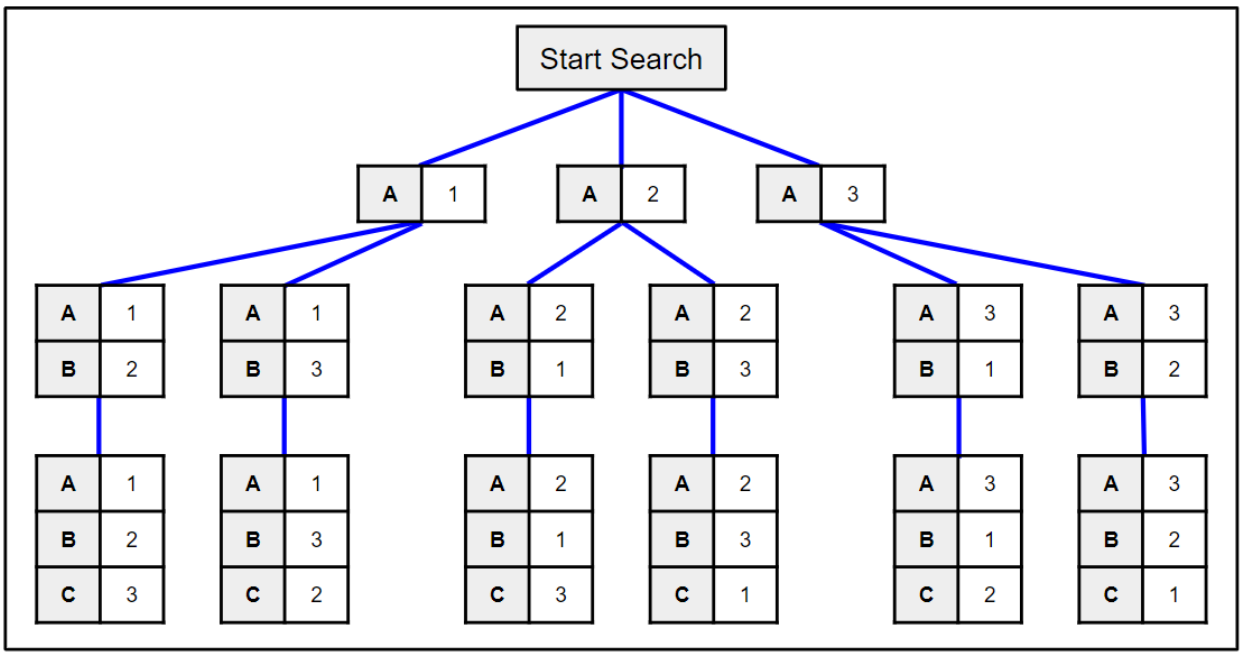
- Puzzle 2:

```
   F O R T Y
       T E N
+      T E N
------------
   S I X T Y
```

In each of these two puzzles, you can assign a digit (i.e., $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$) to each of the letters in the puzzle such that the given equation holds. (Each digit is only assigned to once letter.) The goal is to solve these puzzles. How should you model and solve these puzzles? What is the running time of your solution? Can you optimize your solution to find the answer more quickly, most of the time?

Problem 5.a. Explain how to model the problem as a graph search problem. What are the nodes? How many nodes are there? What are the edges? Where do you start? What are you looking for?

Solution: There are many solutions, but the standard idea here is that each node in the graph contains some subset of the letters and some assignment of digits to letters. We might consider nodes that have increasing subsets of letters. For example, if the letters in the equation are A, B, C and the available digits are $\{1, 2, 3\}$, we might consider the following nodes:



Notice that here I have assumed that each digit can only be used once. The edges connect nodes where you can reach one from the other by assigning one more digit. For example, the node $(A, 1)$ has two outgoing edges to $(AB, 12)$ and $(AB, 13)$. You also have a distinguished start node (that is empty) that connects to all the nodes where only one letter is fixed.

You may notice that this assignment actually yields a tree! That simplifies matters. If you instead represent each node as containing every possible subset of letters and every possible assignment, then you will get a graph (i.e., both $(A, 1)$ and $(B, 2)$ will be connected to $(AB, 12)$).

For each of the nodes where all the letters are fixed, you can determine whether it is a *valid* or an *invalid* node by determining if the equation holds. So why did we construct such a lengthy visual description instead of just assigning all colors and then checking, then making another assignment and then checking? Why are the inner nodes useful at all?

Problem 5.b. To solve this problem, should you use BFS or DFS? Why? How else can you make it run faster?

Solution: DFS is a better search algorithm, though there is no asymptotic difference. Especially for the graph representation above, BFS is equivalent to searching the entire graph. In that case, you might as well simply examine all the assignments. Using DFS, the idea will be to not search the entire graph.
In practice, for these types of exponential sized problems, you want to use various heuristics to decide which branches to explore first, and you want to rapidly trim bad directions.
For example, if you ever assign enough letters to check part of the equation, you should do so right away, and then stop exploring that path if it fails! (This is the obvious "pruning" strategy.) And you should direct your search to go visit nodes faster that you can prune. For example, if you have already assigned the letters R and T, the next letter you should assign is X so that you can verify whether or not the addition works (and prune it if not).

Problem 5.c. When does your search finish? Can you optimize the algorithm to minimize the amount of searching?

Solution: For every node, you can examine the partial assignment and decide whether it is already invalid—no possible further assignment can successfully solve the equation. For every column if the equation, if all the letters are assigned numbers, you can sum up the column (taking into account the potential carry) and determine whether it is possibly correct. If not, then abort the DFS, and do not continue exploring the sub-graph.

Problem 6. Good students, bad students
(Relevant Kattis problem: https://open.kattis.com/problems/amanda)

There are good students and bad students[1]. And at the end of every year, we have to divide the students into two piles: G, the good students who will get an A, and B, the bad students who will get an F. (We only give two grades in this class.)

To help with this process, your friendly tutors have each created a set of notecards. Each card contains the names of two students. One of the two names is a good student, and the other is a bad student. Unfortunately, they do not indicate which is which.

Since the notecards come from thirty eight different tutors, it is not immediately certain that the cards are consistent. Maybe one tutor thinks that Humperdink is a good student, while another tutor thinks that Humperdink is a bad student. (And Humperdink may appear on several different cards.) In addition, the tutors do not provide cards for every student.

Assume you can read the names on a card in $O(1)$ time and that there are more good students than bad students.
Devise an algorithm to determine the answers for the following questions:

- Are the notecards consistent, i.e., is there *any* way we can assign students to G and B that is consistent with the cards?
- Are the notecards sufficient? i.e., is there only one or are there more than one ways to assign students to the sets G and B?
- Assuming that the notecards are consistent and sufficient, determine which set each student belongs in.

[1]No, not really. This sort of binary distinction is silly.

Solution: This problem is about determining whether a graph is bipartite. The first step is to model the problem as a graph: each student is a node, and connect two students with an edge if they appear on the same notecard (on opposite sides). If the notecards are consistent, then you should be able to color the graph with two colors (G)reen (for good) and (B)lue (for bad) in a way that is bipartite, i.e., no edges connect green and blue nodes. Whichever color has more nodes will be the good students! There is one case where the graph is bipartite, but you still cannot decide: if the graph is not connected. Imagine you have two connected components. It may be possible to color the nodes properly with two colors, but not possible to figure out which color is good and which color is bad. (The only information we have to distinguish good from bad is that there are more good than bad students. Imagine we have two components, one that has four color A and seven color B; the other has four color C and six color D. How do you know whether (B, D) or (B, C) are the good students?)

One simple way of solving this problem is as follows: choose any student, and begin performing a BFS; for each node, the first time it is visited in the BFS, label it with its level (i.e., the root is level 0, its neighbors are level 1, its neighbors neighbors are level 2, etc.). If BFS completes and there are still unvisited nodes, we can run BFS again on any unvisited node if we still desire to find some pairing for the insufficient case.

If any node was not visited during the first run of BFS, then the data is not sufficient. Note that if multiple runs of BFS are necessary, then we have a BFS forest (a set of trees)

Then scan the graph: If any node with an even level has a neighbor with an even level, then the data is inconsistent. To get more intuition, think about it from a coloring perspective. Good students are, let's say, red, and bad blue. Each edge, as specified by the notecard, should be placing an edge between a red and a blue node right? Now consider the BFS Tree that is formed after the BFS. Say that all nodes of odd level is red and all nodes at even level is blue. Now if an edge exists between two nodes whose levels have same parity (ie both even or both odd), then we are essentially placing an edge between two nodes of same color, ie, the notecard representing the edge has both students of same type, hence this is not possible.

If the data is both consistent and sufficient, then all students will have been visited in the first run of BFS, and will hence be in a single BFS tree. Check whether there are more students at even levels or odd levels. If, for example, there are more students at even levels than odd levels, then assign students at even levels to G and at odd levels to B. If you find the same number of students at both even and odd levels, report an error (because we assumed that there are more good students than bad). This can all be done in one BFS, rather than multiple scans, but it is often easier to think about and explain as a multi-pass algorithm.

Problem 7. Gone viral (*more challenging*)

There are n students in the National University of Singapore. Among them, there are $n-1$ friendships. Note that friendship is a symmetric relation, but it is not necessarily transitive.

Any two people in the National University of Singapore are either directly or indirectly friends. Formally, between any two different people x and y, either x is friends with y or there exists a sequence $q_1, q_2, \ldots, q_k$ such that x is friends with q_1, q_i is friends with q_{i+1} for all $i < k$ and q_k is friends with y.

It was discovered today that **two** people were found to have the flu in the National University of Singapore.

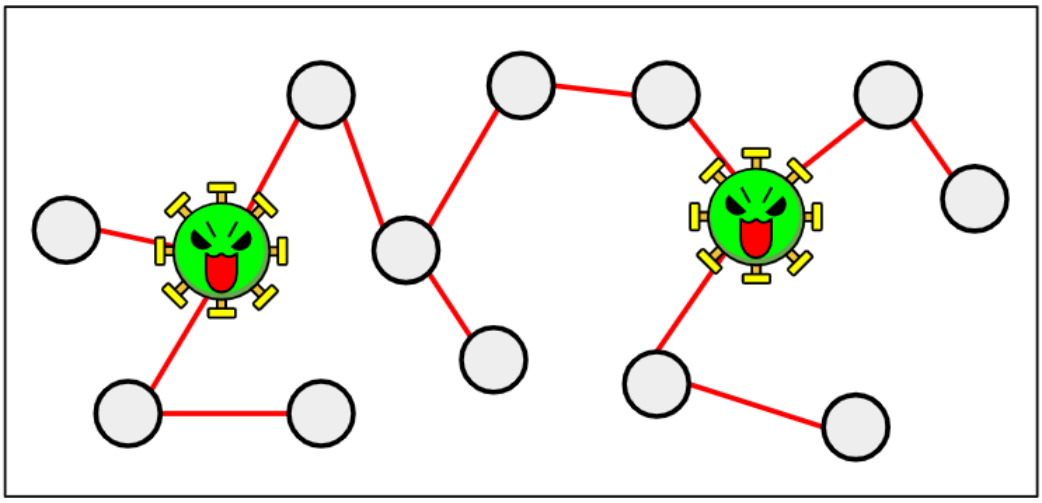


Figure 1: *Gone Viral.* (Matthew Ng Zhen Rui)

Every day, every person can meet with **at most one friend.** When these two people meet, if exactly one of them has the flu, it will be transmitted to the other.

Give an $O(n \log^2 n)$ algorithm to determine the minimum possible number of days before it is possible that *everyone* has the flu.

Hint: First, solve the case where there is only a single person was infected at the start in $O(n \log n)$

Solution: Since the graph is connected and contains n vertices and n-1 edges, it is a tree.

For the one infected student case, the solution is straightforward—consider the tree rooted at the infected student and define $f(x)$ as the minimum number of days needed for all students in the subtree of x to be infected assuming the x^{th} student is infected.

For leaf nodes $f(x) = 0$. For internal nodes, we have $f(x) = \max(f(c_1)+1, f(c_2)+2, \ldots, f(c_k)+k)$ where $c_1, c_2, \ldots, c_k$ are the children infected in this order. Since we want to minimize f, we should sort these $f(c_i)$'s so that $+1$ gets assigned to the maximum $f(c_1)$, $+2$ to the second-maximum and so on.

This can be done in $O(k \log k)$ for an internal node with k children. Overall it is $O(n \log n)$.

For the two infected students case, suppose the two infected students are x and y and consider the unique path from x to y. It is unique because we have a tree. Call this path $x, q_1, q_2, \ldots, q_t, y$. In the optimal solution, there exists some $0 \leq k \leq t$ such that such that $x, q_1, \ldots, q_k$ are infected by students originally infected by x and $q_{k+1}, \ldots q_t, y$ are infected by students originally infected by y. In other words, we can cut some edge along this path and compute the answers for the two trees (using the solution for the one infected person case). Suppose the tree containing x needs a days and the tree containing y needs b days. Then clearly if $a < b$ we should assign more students to x and otherwise y. By binary searching on this path, we have an $O(n \log^2 n)$ solution.