CS161 Summer 2022 Problem Session 4

Problem 1: Sketchy Amortization

Indy has yet another idea for sorting n comparable items. He was told that linked lists are not great for sorting, and he proposes to disrupt that paradigm with his new IndyList structure:

- Put the first 10 items into a linked list L_1 and then sort L_1 using InsertionSort. This takes constant time since 10 is a constant.
- Repeat the following:
  - Put each of the next items into a new linked list (call it L_2) until L_2 has 10 items.
  - Sort L_2 using InsertionSort, which takes constant time since the list has 10 items.
  - Merge (as in the MergeSort operation) L_1 and L_2.

(a) Indy argues that insertion into this structure takes $O(1)$ amortized time. What is the actual amortized cost of insertion?

(b) Indy tries to fix this issue by not processing L_2 until it has 10, 20, 40, 80, ... items. He argues that an insertion that causes L_2 to be merged (when it has k items) takes $O(k)$ work, and so the total work after inserting n items is $O(n) + O(\frac{n}{2}) + O(\frac{n}{4}) + ...$ plus no more than $O(1) \cdot n$, and this still adds up to $O(n)$. Therefore insertion is still $O(1)$ amortized. Is Indy's amortized analysis reasonable?

Problem 2: Connected Components

Suppose you have an unknown graph G (with n nodes and m undirected edges), represented as an adjacency list. The only thing you know is that G is not connected; it has c connected components, with $2 \leq c < n$.

(a) Describe an algorithm that *adds* as few edges as possible to make G connected, and give its running time.

(b) Describe an algorithm that determines the minimum number of edges that you would need to *remove* in order to increase the number of connected components in G by 1. (No need to give the running time.)

(c) Repeat part (a), but for a G with *directed* edges, and replacing "connected" with "strongly connected" everywhere.

Solutions to Problems 1 and 2

(1) (a) Unfortunately for Indy, the costs of the Merges really add up. Suppose we have just inserted our n-th item, where n is a multiple of 10. Then that Merge took $O(n+10)$ time, and the previous one took $O((n-10)+10) = O(n)$ time, and the one before that took $O(n-10)$ time, and so on. That is, we have a sequence like $10 \cdot O(1) \cdot (2 + 3 + ... + (n+1))$, which we know to be $O(n^2)$. Now, all the other insertions (almost n of them) into the IndyList took constant time, which adds an $n \cdot O(1)$ factor, and the $\frac{n}{10}$ InsertionSorts so far (on lists of size 10) each also took constant time. But $O(n^2) + O(n) \cdot O(1)$ is still $O(n^2)$. Therefore the actual amortized cost per insertion is $\frac{O(n^2)}{n} = O(n)$ – asymptotically no better than just inserting each element into the linked list individually.

(b) Indy's amortization argument would actually be correct, except that he fails to consider one thing: now the InsertionSorts run on variable numbers of elements and can no longer be claimed to take constant time.

(2) (a) We can first use BFS or DFS to find all the connected components; this takes $O(n+m)$ time. Then, to connect the graph, we only need to form a tree out of the connected components. We can order the connected components arbitrarily and then connect any vertex in the first component to any vertex in the second component, etc.; this takes $O(c)$ time. Therefore the overall running time is $O(m+n+c)$.

(b) If we just want to add one more connected component, we do this by finding an existing connected component and splitting it into two. So, as in (a), we first find all the connected components using BFS or DFS. Then we can, e.g., run Karger's Algorithm on each connected component individually, identify a smallest min cut out of any component, and choose that one.

(c) The answer here is similar to (a), except that here we use Kosaraju's Algorithm to find the strongly connected components (which also takes $O(n+m)$ time), then add c edges to connect them in a *directed cycle* (not a tree), which takes $O(c)$ time. So the running time is still $O(n+m+c)$.

Problem 3: Terry's Hyperefficient Vacation

Terry is currently visiting a region with n towns ($n \geq 2$), labeled 1 through n. The only way to travel between towns is via roads, which you can think of as *undirected* edges. Terry has access to an adjacency list A that shows which towns are directly connected to each other. There is at most one road directly connecting each pair of towns, and there are m roads in total. The towns are guaranteed to form a single connected component.

The adjacency list A also stores whether each road is "scenic" or "non-scenic". That is, if the only two roads from town 1 were a non-scenic road to town 7 and a scenic road to town 5, the adjacency list entry for 1 could look like `[(7, False), (5, True)]`.

Terry is currently in town 1, but they want to get back to town n (Palo Alto) to resume studying algorithms. Naturally, they want to do so using as few moves as possible. (Whenever Terry goes from one town to another, it counts as one move, regardless of the direction of travel or whether they have used that road before.)

In each of the following questions, **assume that at least one sequence of moves exists that satisfies the conditions**. Your algorithms do not need to handle cases in which no such answer exists.

For parts (a) and (b), you may provide pseudocode and/or a clear description. You may use algorithms from class as subroutines without writing out all their details, but if you modify them, you should describe the modifications clearly. You do not need to justify correctness or running time.

(a) Give an $O(m+n)$ algorithm for finding the smallest number of moves needed to get from town 1 to town n, using *only* scenic roads.

(b) Give an $O(m+n)$ algorithm for finding the smallest number of moves needed to get from town 1 to town n, using *at least one* scenic road. (*Hint for one possible approach: Add another copy of the graph, and think about how to connect it to the original graph so that the copy has a useful meaning.*)

(c) Now suppose that Terry no longer cares whether roads are scenic or non-scenic, but still wants to use as few moves as possible to get from town 1 to town n. Waverly proposes the following algorithm:

- Check whether town 1 is directly connected to town n. If so, return 1. Otherwise:
- Run Dijkstra's algorithm (treating all edge weights as 1), starting from town 1, and find all towns that are at most $\lceil \frac{n}{2} \rceil$ moves away from town 1. For any town found in this way by this run of Dijkstra's, label the town (using a red pen) with its number of moves from town 1, according to this run of Dijkstra's.
- Do the same thing, but starting from town n, and find all towns that are at most $\lceil \frac{n}{2} \rceil$ moves away from town n. For any town found in this way by this run of Dijkstra's, label the town (using a blue pen) with its number of moves from town n, according to this run of Dijkstra's.
- Find a town that has both a red label and a blue label, such that the sum of the labels is minimized. Return that sum.

Does Waverly's algorithm always return the correct value? Briefly explain why or why not.

Solutions to Problem 3

(a) We can preprocess the adjacency list to remove all non-scenic roads (which takes $O(m+n)$ time), then perform BFS starting from town 1 (which also takes $O(m+n)$ time). Alternatively, we can perform a slightly modified version of BFS that checks whether edges are scenic before exploring them.

DFS would not be well-suited to this problem – we have no guarantee that it would find a shortest path.

(b) Let G be the original graph. We can create a copy C of G and then, for each scenic edge in G between towns i and j, replace it with one edge from town i in G to town j in C, and another edge from town j in G to town i in C.

Notice that C corresponds to the universe in which we have used at least one scenic road, and G corresponds to the universe in which we have not. Then all we need to do is run a BFS from town 1 in G to town n in C.

Creating C adds another n vertices and m edges, and we add up to m more edges beyond that, but the BFS still runs in $O(m+n)$ time.

We do not need to try to prevent the search from going back from C to G, since any such exploration cannot find a better answer than we could get by staying in C. Also notice that the copy has all its scenic edges in the places they were originally in G.

(c) Yes – Waverly gets it right sometimes! Consider any optimally short path from town 1 to town n. Suppose its length (in number of edges) is l. A key insight is that the two Dijkstra's searches can use parts of this path as well. Another is that Dijkstra's behaves like BFS because the edge weights are all 1.

- If the optimal l is even, let c be the town in the center of the path. The first Dijkstra's will reach c in exactly $\frac{l}{2}$ moves, and the second Dijkstra's will also reach c in exactly $\frac{l}{2}$ moves, so Waverly's algorithm will correctly return n.
- If the optimal l is odd, let c_1 and c_2 be the two towns in the center of the path, with c_1 being closer to town 1 and c_2 being closer to town n. The first Dijkstra's will reach c_2 in exactly $\frac{l+1}{2}$ moves, and the second Dijkstra's will also reach c_2 in exactly $\frac{l-1}{2}$ moves, so Waverly's algorithm will correctly return l.

Problem 4: BFS/DFS orders

Draw any undirected graph with vertices A, B, C, D, E such that a BFS starting at A finds the vertices in the order A, B, D, E, C, but a DFS starting at A finds the vertices in the order A, B, C, E, D. Assume that both searches break ties alphabetically.

Optionally: How many valid solutions are there?

Problem 5: Decrease-Key

In lecture, we learned that we shouldn't use our standard binary-tree-based heap for Dijkstra's Algorithm because it is not optimal at handling `decrease-key` operations. These occur in Dijkstra's when an overestimate for a node is replaced with a smaller estimate.

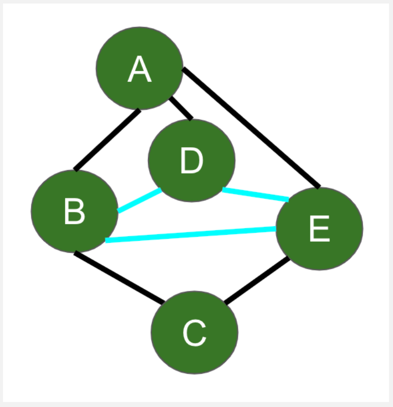
(a) Can you come up with a weighted undirected graph in which Dijkstra's has to do $O(n^2)$ `decrease-key` operations? You don't need to be completely precise about the details – just convince yourself that it really can happen.

(b) Why is it, again, that the standard binary-tree-based heaps are "bad at" `decrease-key`, i.e., they take $O(\log n)$ time to do it (as opposed to the Fibonacci heap's amortized $O(1)$?) (Note: you do not need to know how Fibonacci heaps work, just that they can handle this operation in $O(1)$ time.)

Solution to Problem 4

This takes some experimentation and/or logic:

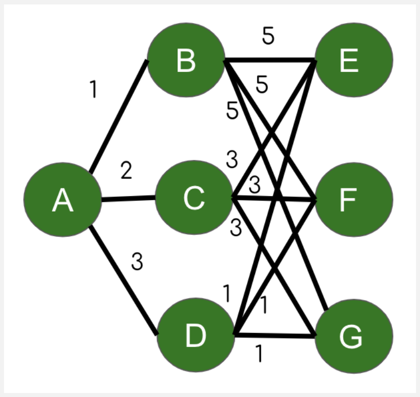
- We know that B must be connected to A, since it is found first from A by either search.
- From the DFS order, we see that either C must be connected to B, or it is connected to A and found via backtracking. But if C were connected to A, then BFS would have found it right after B. So C is connected to B, but not A.
- Now observe that BFS finds D right after B. So either D is connected to A, or B is the only vertex connected to A and D is connected to B. However, in the latter case, because we already know C is connected to B, then BFS would have found C next. Therefore we must be in the former case: D is connected to A.
- D cannot be connected to C, because then the DFS would have found D after C, instead of E.
- E must be connected to A. If not, there is no way the BFS could have found it before C, since even if E were connected to B, the BFS would have found C before E.
- At this point, we can check that any of the remaining three edges (shown here in light blue) can either be present or absent without causing a contradiction, so there are a total of 8 valid solutions.



Notice that the unconstrained edges are precisely those between different nodes that are on the same "level" away from our starting point; this is not a coincidence.

Solution to Problem 5

(a) Here's one such construction:



- We start off by adding B, C, and D to the heap with estimates 1, 2, and 3, respectively.
- Next, we process B (since 1 is the smallest estimate in the heap). We add E, F, and G to the heap with estimates 6, 6, and 6.
- Then we process C (since 2 is the smallest estimate in the heap). Oh no – now we have updated estimates for all of E, F, and G! They all go down to 5, so we have to call decrease-key on each one.
- Next we process D (since 3 is the smallest estimate in the heap). Oh no, again – we have updated estimates of 4 for each of E, F, G! Again, we have to call decrease-key on each one.

You can imagine extending this to an extreme situation, with arbitrarily large n, in which just under half of our n nodes each have to be updated just under $\frac{n}{2}$ times each. This is $O(n^2)$ calls to decrease-key.

(b) If we decrease a value somewhere in the heap, it may now be smaller than its parent, so we have to swap it up the tree, perhaps even all the way to the root, which can take $O(\log n)$ time.