CS 170 Homework 3

Due **9/21/2020, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

In addition, we would like to share correct student solutions that are well-written with the class after each homework. Are you okay with your correct solutions being used for this purpose? Answer "Yes", "Yes but anonymously", or "No"

2 Exam Policy

Please read the exam policy document that was distributed to all students and then do the following:

(a) Submit a Zoom meeting link that you will use during the exam here:

https://forms.gle/t9dZhc5b5veeWBgR8

(b) Follow the steps in the setup checklist in the exam policy document. As your answer to this question, please describe your setup for taking the exam, where you plan to sit, and any other relevant details in a few sentences.

(c) Please record a 5min sample recording following the Set-Up Checklist and After the Exam sections. Submit the link to this recording here:

https://forms.gle/ovsWX1SSxR93rCyv9

3 Triple sum

We are given an array $A[0..n-1]$ with n elements, where each element of A is an integer in the range $0 \leq A[i] \leq n$ (the elements are not necessarily distinct). We would like to know if there exist indices i, j, k (not necessarily distinct) such that

$$A[i] + A[j] + A[k] = n$$

Design an $\mathcal{O}(n \log n)$ time algorithm for this problem. Note that you do not need to actually return the indices; just yes or no is enough. *Hint: Use FFT/fast polynomial multiplication.*

Please give a 3-part solution to this problem.

Solution:

Main idea Exponentiation converts multiplication to addition. For example, observe $x^3 * x^2 = x^{2+3} = x^5$. This gives us the idea to represent the lists as polynomials, with the elements in the lists in the exponents (This is a very common trick that you should remember!). For example, we can represent the array $[1, 3]$ as $x^1 + x^3$. If we multiply this with the polynomial for $[2,4]$, $x^2 + x^4$, we get the polynomial $x^3 + 2x^5 + x^7$. Notice that $3, 5, 7$ correspond to the possible sums of an element in $[1,3]$ and an element in $[2,4]$.

More formally, define

$$p(x) = x^{A[0]} + x^{A[1]} + \cdots + x^{A[n-1]}.$$

Notice that $p(x)^3$ contains a sum of terms, where each term has the form $x^{A[i]} \cdot x^{A[j]} \cdot x^{A[k]} = x^{A[i]+A[j]+A[k]}$. Therefore, we just need to check whether $p(x)^3$ contains x^n as a term.

Proof of Correctness Observe that

$$q(x) = p(x)^3 = \left(\sum_{0\le i<n} x^{A[i]}\right)^3 = \left(\sum_{0\le i<n} x^{A[i]}\right) * \left(\sum_{0\le j<n} x^{A[j]}\right) * \left(\sum_{0\le k<n} x^{A[k]}\right)$$

$$= \sum_{0\le i,j,k<n} x^{A[i]}x^{A[j]}x^{A[k]} = \sum_{0\le i,j,k<n} x^{A[i]+A[j]+A[k]}.$$

Therefore, the coefficient of x^n in q is nonzero if and only if there exist indices i, j, k such that $A[i] + A[j] + A[k] = n$. So the algorithm is correct. (In fact, it does more: the coefficient of x^n tells us *how many* such triples (i, j, k) there are.)

Runtime Analysis Constructing $p(x)$ clearly takes $O(n)$ time. $p(x)$ is a polynomial of degree at most $n = O(n)$. Therefore doing the two multiplications to compute $q(x)$ takes $O(n \log n)$ time with the FFT. Finally, looking up the coefficient of x^t takes constant time, so overall the algorithm takes $O(n \log n)$ time.

Comment: This problem promised you that each element of the array is in the range $0 \ldots n$. What if we didn't have any such promise? Then the FFT-based method above becomes inefficient (because the degree of the polynomial is as large as the largest element of A). It is easy to find a $O(n^2)$ time algorithm, but no faster algorithm is known. In particular, it is a famous open problem (called the 3SUM problem) whether this problem can be solved more efficiently than $O(n^2)$ time. This problem has been studied extensively, because it is closely connected to a number of problems in computational geometry.

4 Protein Matching

Often times in biology, we would like to locate the existence of a gene in a species' DNA. Of course, due to genetic mutations, there can be many similar but not identical genes that serve the same function, and genes often appear multiple times in one DNA sequence. So a more practical problem is to find all genes in a DNA sequence that are similar to a known gene.

To model this problem, let g be a length-n string corresponding to the known gene, and let s be a length-m string corresponding to the full DNA sequence, where $m \ge n$. We would like to solve the following problem: find the (starting) location of all length n-substrings of s which match g in at least $n - k$ positions. For example, using 0-indexing, if $g = ACT$, $s = ACTCTA$, and $k = 1$ your algorithm should output 0 and 2.

(a) Give a $O(nm)$ time algorithm for this problem.

Solution:

For each of $i \in \{0, 1, \ldots, m-n\}$ starting points in s, check if the substring $s[i : i+n-1]$ differs from g in at most k positions. The check takes $O(n)$ time at each of $O(m)$ starting points, so the time complexity is $O(mn)$.

(b) Assume g and s are given as bitstrings, i.e. every character is either 0 or 1. Give a $O(m \log m)$ time algorithm that works for any k.

Hint: Represent the strings as vectors, and use FFT/fast polynomial multiplication.

Solution: Let g' be g with 0's replaced by -1's. Let $p_1(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ where $a_d = g'(n-d-1)$ for all $d \in \{0, 1, \ldots, n-1\}$. Similarly, let $p_2(x) = b_0 + b_1 x + \cdots + b_{m-1}x^{m-1}$ where $b_d = s'(d)$ for all $d \in \{0, 1, \ldots, m-1\}$, where again s' is just s with 0's replaced by -1's. Notice $p_1(x)$ is reversed in the sense that the coefficients are in opposite order of the bits in g.

Now consider $p_3(x) = p_1(x) \times p_2(x) = c_0 + c_1 x + \cdots$, the coefficient of x^{n-1+j} in $p_3(x)$ is $c_{n-1+j} = \sum_{i=0}^{n-1} a_{n-1-i} b_{j+i} = \sum_{i=0}^{n-1} g'(i)s'(j+i)$ for any $j \in \{0, 1, \ldots, m-n\}$, which is exactly the dot product of the substring in s' starting at index j and the string g'. If these strings differ in at most k positions, then this dot product will be at least $n-2k$. Thus all we need is to compute $p_3(x)$, and output all the j's between 0 and $m-n$ such that $c_{n-1+j} \geq n-2k$.

The computation takes a FFT, a point-wise product, an inverse FFT, and a linear scan of the coefficients. The running time of this algorithm, $O(m \log m)$, is dominated by the FFT and inverse FFT steps, which each take $O(m \log m)$ time. The point-wise product and search for $c(i) \geq n-2k$ each take $O(n)$ time.

You do not need a 3-part solution for each part. Instead, describe the algorithms clearly and give an analysis of the running time.

5 Practice with SCCs

This question is a "solo question". It is meant as a way for you to check how well you would do on a question of similar difficulty if it appeared on an exam. Do not collaborate with other students on this problem, and staff may give less help on this problem than they do on other homework problems.

(a) Design an efficient algorithm that given a directed graph G, outputs the set of all vertices v such that there is a cycle containing v. In other words, your algorithm should output all v such that there is a non-empty path from v to itself in G.

Solution:

Main idea: We compute the SCCs of G, and output all vertices in an SCC that is of size 2 or greater.

Runtime analysis: Computing the SCCs can be done in $O(|V|+|E|)$ time, and we can check their size/report all vertices in each SCCs in less than this much time.

Correctness: If a vertex v is in a cycle, then there must be some other vertex u in this cycle. This cycle contains a path from u to v and v to u, so they must be in the same SCC, which is thus a SCC of size at least 2. On the other hand, if v is in a SCC of size at least 2 containing some other vertex u, then there is a path from v to u and u to v, and thus a path from v to itself.

(b) Design an efficient algorithm that given a directed graph G determines whether there is a vertex v from which every other vertex can be reached. (Hint: first solve this for directed acyclic graphs. Note that running DFS from every single vertex is not efficient.)

Solution:

We provide two solutions below that both run in linear time $O(|V| + |E|)$ (there may be many more). Note that for full credit for the runtime portion of the solution, you must specify what the variables of your runtime are.

Solution 1:

Main idea: In directed acyclic graphs, we just check if the number of source nodes (zero indegree) is 1 or more than 1, and output yes/no respectively. For general graphs, we first form the SCCs and the metagraph, and then just run this algorithm on the metagraph.

Runtime: Finding the SCCs/metagraph can be done in $O(|V| + |E|)$ time via DFS as seen in the textbook, and counting the number of sources in the metagraph can also be done in $O(|V| + |E|)$ time by just computing the in-degrees of all vertices using a single scan over the edges.

Correctness: In a DAG, certainly if more than 1 source node exists, there is no true source, because one cannot reach either source from the other. If there is only 1 source node u, let v be any other vertex. If we repeatedly follow incoming edges, starting at v, each edge we follow moves us backwards in the topological ordering in v, which means we must eventually reach a source node, which can only be u. This means that we can reach v from u.

Similarly, if there is only one source SCC, any vertex in it can reach any other vertex in the graph, but if there are more than one source SCCs, there is no single vertex that can reach all vertices.

Solution 2:

Main idea: Run DFS once on G to form a DFS forest. Now, let v be the root of the last tree that this run of DFS visited. Run DFS starting from v to determine if every vertex can be reached from v. If so, output v, if not, output that no true source exists.

Runtime: Since the algorithm just involves running DFS twice, it runs in $O(|V| + |E|)$ time.

Correctness: If we find v is a true source, outputting it of course is correct.

Suppose we outputted that there was no true source. v clearly can't be a true source, and nothing visited after v can be a true source, since v is the last root and thus all vertices visited after v are reachable from v. But every vertex visited before v must not be able to reach v, because otherwise the DFS would have taken a path from one of those vertices to v and thus v would not be a root in the DFS forest. So nothing visited before v can be a true source, since nothing visited before v can reach v, so there is no true source.

Please give a 3-part solution to both parts of this problem. For both parts, try to keep each part of the 3-part solution no longer than a few sentences.

6 Splitting Edges

Let $T = (V, E)$ be an undirected connected tree. That is, T has no cycles and there is a path from every vertex to every other vertex in T. For each edge $e \in E$, $T \setminus e$ has exactly two connected components. Let us call the sizes of these two components the *split numbers* of e.

(a) Describe a linear-time algorithm to compute the split numbers of all edges $e \in E$, and prove its runtime (proof of correctness not required).

Solution:

Algorithm: Fix an arbitrary root node $r \in V$. For any $v \in V$, define $N(v)$ as the number of nodes in the sub-tree rooted at v (that is, the vertices u whose path from r to u contains v). Run a DFS traversal of T from r, and on the post-order visit of any vertex $v \in V$, compute

$$N(v) = 1 + \sum_{c \,\in\, \text{children of } v} N(c)$$

Now any edge $e \in E$ must be of the form $e = \{p, c\}$ where p is the parent of c is the DFS tree rooted at r. We compute its split numbers as $N(c)$ and $|V| - N(c)$.

Another algorithm is to observe that for any edge $\{p, c\}$ where p is the parent and c is the child, $N(c) = (post(c) - pre(c) + 1)/2$ (where $pre(c), post(c)$ are the pre-visit and post-visit numbers when we DFS from r), and we can then just compute the split numbers as before. This is because when we DFS from r, the $N(c) - 1$ vertices that are descendants of c in the DFS tree each contribute 2 to the difference $post(c) - pre(c)$, since we need to pre-visit and post-visit each of them between pre-visiting and post-visiting c.

Runtime: When computing $N(v)$, we do $O(\deg(v))$ work at each node, so the runtime for that part is $O(\sum_{v \in V} \deg(v)) = O(|E|)$. Then, for each edge $e \in E$, we compute its split numbers in constant time, so computing all of them takes $O(|E|)$ time. Therefore the overall runtime is $O(|E|) = O(|V|)$. The same runtime holds for the pre/post-order based algorithm, since the pre/post-order can be computed in the same time as DFS.

(b) For any $u, v \in V$, define $d(u, v)$ to be the number of edges on the unique path from u to v in T. Suppose we've already computed the split numbers for each edge $e \in E$. Now, given only the split numbers and not the original graph, describe how to compute the total pairwise distance between all vertices

$$R := \sum_{u,v \in V} d(u, v)$$

More specifically, if the edge e has split numbers x_e, y_e, then you're only given all (x_e, y_e) values as input. Justify your answer.

Hint: for a given edge $e \in E$, how many times does it contribute to the above sum?

Solution: For each edge $e \in E$, the edge contributes 1 to R for each pair u, v such that e lies on the unique path from u to v. Notice that e lies on the path from u to v if and only if u and v are from the two opposite connected components of $T \setminus e$, which have size x_e and y_e. The number of ways we can choose one vertex from each component is $x_e y_e$, so this is how much each edge contributes to the sum R. Therefore $R = \sum_{e \in E} x_e y_e$.

(c) A perfect matching is a set of edges $M \subseteq E$ such that every vertex in V is incident to *exactly* one edge in M. Again define the split numbers of e to be x_e, y_e.

Let T be a tree that has a perfect matching M. Prove that

$$M = \{e \in E : x_e \text{ is odd and } y_e \text{ is odd}\}$$

Note that not all trees have perfect matchings - you only want to prove this property holds for T that does have a perfect matching.

Solution: Suppose e is not in M. Then each component of $T \setminus e$ still has a perfect matching by matching vertices as they're matched in M. Therefore, the size of each component must be even, i.e x_e and y_e are even.

Now suppose e is in M. Denote the endpoints of e as u, v, and let T' be the connected component of u in $T \setminus e$. Notice that $T' \setminus u$ has a perfect matching by matching vertices as they're matched in M. Therefore $|T' \setminus u|$ is even, and hence $|T'|$ is odd. By the same argument, the connected component of v in $T \setminus e$ must be odd, so the split numbers of e are both odd.

Therefore, $e \in M$ if and only if its split numbers x_e and y_e are odd, so

$$M = \{e \in E : x_e \text{ is odd and } y_e \text{ is odd}\}$$

7 Vertex Separators

Let $G = (V, E)$ be an undirected, unweighted graph with $n = |V|$ vertices. We call a set of vertices S a $u - v$ *separator* if (1) S does not contain u or v and (2) deleting all vertices in S and all edges adjacent to these vertices from G disconnects u and v. In other words, S is a $u - v$ separator if every path from u to v goes through some vertex in S.

Give an efficient algorithm that takes G, u, v as input and finds a $u - v$ separator of size at most $\frac{n-2}{d-1}$, where d is the length of the shortest path from u to v. Assume that u and v are connected in G and $d > 1$. **Give a three-part solution.**

Hint: Is there a natural way to partition (some of) the vertices that aren't u, v into $d - 1$ sets?

Solution:

Main idea: We run BFS starting from u to compute the distance from u to all vertices in the graph. Let S_i be the set of all vertices at distance i from u. We output whichever of $S_1, S_2, \dots S_{d-1}$ is the smallest (we know the value of d from BFS).

Correctness: For all i from 1 to $d-1$, we know from the analysis of BFS that any path from u to v must pass through some vertex distance i from u. So deleting any S_i disconnects u and v. Each vertex that is not u or v appears in at most one of $S_1, S_2, \dots S_{d-1}$, and u and v don't appear in any of these sets. So the total size of the sets is at most $n-2$. The average size of these sets is thus at most $\frac{n-2}{d-1}$, and the smallest set's size can only be smaller than the average size.

Runtime analysis: BFS takes $O(|V|+|E|)$ time. We can write down the S_i and figure out which set is the smallest in $O(|V|)$ time. So the overall runtime is $O(|V|+|E|)$.

8 The Greatest Roads in America

Arguably, one of the best things to do in America is to take a great American road trip. And in America there are some amazing roads to drive on (think Pacific Crest Highway, Route 66 etc). An intrepid traveler has chosen to set course across America in search of some amazing driving. What is the length of the shortest path that hits at least k of these amazing roads?

Assume that the roads in America can be expressed as a directed weighted graph $G = (V, E, d)$, and that our traveler wishes to drive across at least k roads from the subset $R \subseteq E$ of "amazing" roads. Furthermore, assume that the traveler starts and ends at her home $h \in V$. You may also assume that the traveler is fine with repeating roads from R, i.e. the k roads chosen from R need not be unique.

Design an efficient algorithm to solve this problem. Provide a 3-part solution with runtime in terms of $n = |V|$, $m = |E|$, k.

Hint: Create a new graph G' based on G such that for some s', t' in G', each path from s' to t' in G' corresponds to a path of the same length from h to itself in G containing at least k roads in R. It may be easier to start by trying to solve the problem for $k = 1$.

Solution: Main idea:
We want to build a new graph G' such that we can apply Dijkstra's algorithm on G' to solve the problem.

We'll start by creating $k+1$ copies of G. Call these $G_0, G_1, \dots G_k$. These copies include all the edges and vertices in G, as well as the same weights on edges. Let the copy of v in G_i be denoted by v_i. For each road $(u, v) \in R$, we also add the edges $(u_0, v_1), (u_1, v_2), \dots (u_{k-1}, v_k)$, with the same weight as (u, v). The intuition behind creating these copies is that each time we use an edge in G' corresponding to an edge in R, we can advance from one copy of G to the next, and this is the only way to advance to the next copy. So if we've reached v_i from h_0, we know we must have used (at least) i edges in R so far.

Now, consider any path p' from h_0 to h_k in G'. If we take each edge (u_i, v_i) or (u_i, v_{i+1}) and replace it with the corresponding edge (u, v) in G, we get a path p in G from h to itself. Furthermore, since the path goes from h_0 to h_k, it contains k edges of the form (u_i, v_{i+1}), where (u, v) is an edge in R. So, p will contain at least k edges in R.

Our algorithm is now just to create G' as described above, and find the shortest path p' from h_0 to h_k using Dijkstra's, and then output the corresponding path p in G.

Correctness:
Assume there is a valid path p in G that is shorter than the one produced by this algorithm. Consider the equivalent path p' in G' formed by modifying the path to go to the next copy of G whenever an edge of R is crossed. Since p is valid, p' must go from h in G_0 to h in G_k. But then p' would be a shorter path in G' than the one produced by Dijkstra's, which is a contradiction.

Runtime:
Since G' includes $k+1$ copies of G, Dijkstra's algorithm will run in time $O((km+kn)\log(kn))$. Since $k \leq m$ and $\log m = O(\log n)$, the runtime can be simplified to $O(k(m+n)\log n)$.