Midterm 1

Rules and Guidelines

- **The exam is out of 100 points and will last 120 minutes.**
- Answer all questions. Read them carefully first. Not all parts of a problem are weighted equally.
- Begin each problem on a new page
- Be precise and concise.
- The problems may **not** necessarily follow the order of increasing difficulty.
- Good luck!

1 Is There A Cycle?

Design a linear time algorithm that given a directed graph G, outputs a cycle in the graph if there is one, or else a source vertex and a sink vertex. Just an algorithm and a clear justification of why it always gives a valid output are needed.

Solution 1:

Main idea: Run DFS. If there's a back edge, (u, v), then follow parent pointers from u to v to find the path from v to u in the DFS tree. This path + the edge (u, v) is a cycle. Otherwise, if there is no back edge, then there's no cycle in the graph, and it is a dag. So we find a topological ordering of the graph and output the first and last vertex in this ordering – these must be a source and sink by the definition of a topological ordering.

Solution 2:

Main idea: Compute the metagraph (the graph on the SCCs) of G. If any SCC has size at least 2, there must be a cycle in the SCC. To find it just start from any vertex and follow edges within the SCC (marking vertices as you go) until you hit a vertex you have already visited, yielding a cycle.

Otherwise all SCCs are size 1, i.e. there is no cycle in the graph, since any cycle creates an SCC of size at least 2. In this case, any source and sink SCC are actually just a source and sink vertex.

2 FFT simplified

(a) Write the 2-by-2 Fourier transform matrix. What root of unity did you use? Write it in the form $a + bi$.

(b) Denote by H_1 your 2-by-2 solution to the previous part. We recursively define the 2^n-by-2^n matrix H_n as:

$$H_n = \begin{bmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{bmatrix}$$

Explicitly write down the 4-by-4 matrix H_2.

(c) Let $N = 2^n$. Given an N-dimensional vector v, give an $O(N \log N)$-time algorithm to compute $H_n v$. Just the algorithm and runtime analysis is needed.

(a)

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

We used the 2nd root of unity, which is -1.

(b)

$$H_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

(c) Let's split v into two $N/2$-dimensional vectors v_1, v_2 and observe:

$$H_n v = \begin{bmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} H_{n-1}v_1 + H_{n-1}v_2 \\ H_{n-1}v_1 - H_{n-1}v_2 \end{bmatrix}$$

So the algorithm is to recursively compute $H_{n-1}v_1, H_{n-1}v_2$, and then use the above formula to compute $H_n v$ in linear time from the answers. This takes time $T(N) = 2T(N/2) + O(N) = O(N \log N)$ by the master theorem.

3 Booking Flights

We want to book a cheap flight route to travel from city s to city t. There are n cities with airports including s and t. Airlines offer m flights, where the ith flight goes non-stop from city u_i to v_i and costs c_i dollars (all c_i are positive integers). We wish to find the cheapest route from s to t, but if there are multiple cheapest routes, we wish to find among them one with the fewest flights (to minimize the number of airports we have to transit through). Give an algorithm that outputs such a flight itinerary, and a clear justification of correctness.

Note: If you modify Dijkstra's itself, the correctness of Dijkstra's probably does not imply the correctness of your algorithm. You will have to give a modified version of the proof of correctness as well for full credit. If, on the other hand, you use Dijkstra's as a black box in your algorithm, using Dijkstra's correctness as a black box in your proof will probably suffice.

One way you might approach this question is by modifying Dijkstra's algorithm. In this case you should give a clear description and proof of correctness of your algorithm. For partial credit you may give the key idea in the correctness proof of Dijkstra's algorithm. Another way you might approach this problem is by using Dijkstra's algorithm as a subroutine. In this case you may assume the correctness of Dijkstra's algorithm while proving the correctness of your algorithm.

We can represent the flights as a weighted graph, where the cities are vertices and the flights are directed edges with weights c_i.

Solution 1:

Main idea: Change each edge's weight from c_i to $n \cdot c_i + 1$, and then run Dijkstra's to find the shortest path from s to t. (alternately change each edge weight from c_i to $c_i + 1/n$)

Correctness: A path with total cost C and using $\ell < n$ edges will have weight $n \cdot C + \ell$ using the new weights. Notice that $n \cdot C \leq n \cdot C + \ell < n \cdot (C + 1)$, so that any path of cost $C + 1$ is heavier than any path of cost C under the new weights. This means that the lightest path under the new weights must necessarily be the lightest path under the old weights, but with the additional property that it has the fewest edges (smallest ℓ). The correctness now follows from the correctness of Dijkstra's.

Solution 2:

Main idea: In addition to $dist(v)$, we track $\ell(v)$, the least edges used on any path of length $dist(v)$. We initialize $\ell(s) = 0$. Normally Dijkstra's uses the update "if $dist(u) + c(u,v) < dist(v)$ set $dist(v) = dist(u) + c(u,v)$". Instead we use "if $dist(u) + c(u,v) < dist(v)$, set $dist(v) = dist(u) + c(u,v), \ell(v) = \ell(u) + 1$" and "if $dist(u) + c(u,v) = dist(v)$, set $\ell(v) = \min(\ell(v), \ell(u) + 1)$. We sort the heap Dijkstra's uses by $dist(v)$ and then $\ell(v)$.

Correctness: Observe that our extra tracking of ℓ does not affect the $dist$ function at all for each vertex — the heap only uses ℓ to break ties, and we only do the extra work when the distances are equal, and during the extra work, we do not reassign $dist$ values. Therefore, we can safely inherit from Djikstra's algorithm that the outputted distance is the minimum possible. To show that the path must additionally have the fewest number of edges, we can argue by induction on the path length. Assume that the algorithm correctly finds the shortest lightest path for all vertices where the length of the path is at most k. If the shortest lightest path from s to v has length $k + 1$, and u is the previous vertex on that path, then u must already be assigned its correct distance before v (since $d(s,u) < d(s,v)$), and at that point the $dist(v)$ is set to $dist(u) + c(u,v)$ and $\ell(v) = \ell(u) + 1$, which is the correct distance and length, thus completing the induction step.

Solution 3:

Main idea: We modify Dijkstra to work on tuples rather than integers. The edges now have weight tuples $(c_i, 1)$ and we initialize this tuple to $(0,0)$ for s. Notice that addition of tuples is well defined, and the "minimum" between two tuples defined as whichever comes first in lexicographic order (e.g. $(c_i, a) + (c_j, b) = (c_i + c_j, a + b)$ and $(c - 1, 100) < (c, 1) < (c, 2)$).

Correctness: The correctness follows similarly to that of Solution 2.

Solution 4:

Main idea: We run Dijkstra's on this graph. Given $d(v)$ for all vertices, we construct a new graph G' with the same vertices, but only the edges (u,v) for which $d(u) + c(u,v) = d(v)$. We use BFS to find the path from s to t using the least edges in G' and output this path.

Correctness: It suffices to show that the set of paths from s to t in G' is equivalent to the set of shortest paths from s to t in the original graph. Given this, since BFS will find the path in G' from s to t using the

least edges, it also finds the shortest path in the original graph using the least edges.

Any shortest path p from s to t in the original graph satisfies $d(u) + c(u, v) = d(v)$ for all edges (u, v) on the path, as otherwise since we can swap in a shortest path from s to v to improve it. So all such p are also a path in G. On the other hand, any path of the form $s, v_1, v_2, \ldots v_k, t$ in G' has cost $d(s) + c(s, v_1) + c(v_1, v_2) \ldots c(v_k, t)$. Since edges in G' satisfy $d(u) + c(u, v) = d(v)$, this cost equals $d(t)$, i.e. this path is a shortest path to t in the original graph.

4 Netflix Similarity

There are n movies on Netflix (identified by the numbers $1, \ldots, n$) and Alice and Bob have watched all of them! Alice lists all n movies according to her ranking starting with her favorite $(a_1, \ldots, a_n)$ and similarly, Bob lists all movies according to his ranking $(b_1, \ldots, b_n)$. One measure of difference between Alice and Bob's tastes is the number of inversions between their orderings, i.e. the number of pairs of movies u, v such that u is rated higher than v by Alice but lower by Bob. Design an $O(n \log n)$ algorithm to compute the number of inversions. Justify the correctness of your algorithm.

Example: if Alice's ordering is $(4, 3, 1, 2)$, with 4 her favorite movie and 2 her least favorite, and Bob's ordering is $(3, 4, 2, 1)$, then the number of inversions is 2, corresponding to the pairs 1, 2 and 3, 4.

Note: if you are unable to solve this problem, for half the points you may solve the problem where you are given just one sequence of n numbers $(x_1, \cdots, x_n)$, and you wish to compute the number of inversions in that list, i.e. pairs (i, j) such that $i < j$ but $x_i > x_j$ (this was covered in lecture!)

If you are solving the problem for full points, you may call the algorithm from lecture as a black box if needed.

We first provide an answer to the full credit problem, which invokes the algorithm from lecture as a black box.

Full credit solution:

Solution 1: *Main idea:* In the special case where Alice's preference list is $(1, 2, \ldots, n)$, the quantity we are interested in is simply the number of inversions in Bob's list. So we simply rename the movies: a_i is renamed i. So now Alice's preference list reads $(1, 2, \ldots, n)$! Let $(b'_1, \ldots, b'_n)$ be Bob's preference list after this renaming. We can compute this in $O(n)$ steps. We now invoke the algorithm from lecture to compute the number of inversions in $(b'_1, \ldots, b'_n)$.

Correctness: Correctness under renaming is immediate, since the definition of an inversion does not depend upon the names of the movies – just their order in the two preference lists.

Solution 2:

Rewrite the input as a list of n tuples (m, a', b'), where m is the integer identifying a movie, a' is the rank of that movie in Alice's preferences, and b' is the rank in Bob's preferences. Now we sort this list by a' in increasing order. All this can be done in $O(n \log n)$ steps. Now we run the counting inversions algorithm from lecture on the b' values in the newly sorted list.

Correctness: For the tuples solution, notice that every inversion in the b' values in the final list corresponds to a pair of movies m_1, m_2, with Alice's ranking a_1, a_2 and Bob's rankings b_1, b_2, such that $a_1 < a_2$, but $b_1 > b_2$. But these are exactly the pairs of movies m_1, m_2 that are inversions between Alice and Bob's preference lists.

Half credit solution:

Main idea: Split the list into two lists L and R of size $n/2$ and recurse on L, R. It remains to find the number of inversions x, y with $x \in L$ and $y \in R$. Note that sorting L and R does not change whether x, y is an inversion, but does make it easier to determine the number of such inversions involving x. So we design an algorithm that not only counts inversions, but returns a sorted copy of the list in ascending order.

Let I_1, I_2 be the number of inversions the recursive calls find in L, R respectively. We will now count I_3, the number of inversions of one element in L and one in R.

After the recursive call L, R are sorted, and we apply the merge procedure: we peel off the smaller of the first elements of the two lists, and add it to our sorted list. Every time we peel an element off of R, we increase I_3 by the number of elements remaining in L.

After the merge procedure finishes, we return the sorted list and $I_1 + I_2 + I_3$ as the number of inversions.

Correctness: Our algorithm sorts correctly because it is carrying out Mergesort (merging sorted lists produces a sorted list + induction).

For counting inversions, by the induction hypothesis our algorithm correctly finds the number of inversions in L, R. Anytime the merge procedure peels an element e off R, we know that e is larger than all elements already peeled off L and smaller than the elements not peeled off L since L is sorted.

So the number of inversions consisting of an e and an element in L is the number of elements remaining in L, which is exactly what we count. Therefore I_3 is correct, and so our algorithm finds the correct total number of inversions, thus proving the induction step.

Clearly the running time satisfies $T(N) = 2T(N/2) + O(N) = O(N \log N)$ by the master theorem.