CS 38 Introduction to Algorithms **Spring 2014**

Midterm Solutions

Posted: May 6

If you have not yet turned in the Midterm, you should not consult these solutions.

1. (a) Here is a counterexample. The nodes are $1, 2, 3, 4, 5, 6, 7$ and we have edges

$$(1, 2), (1, 3), (1, 4), (2, 5), (3, 6), (4, 7), (5, 6), (6, 7), (5, 7).$$

The BFS traversal explores nodes in order $1, 2, 3, 4, 5, 6, 7$, and at the time it discovers node 6, it finds already-discovered node 5. But node 6 and 5 have predecessors 3 and 2, respectively so no triangle is declared. At the time it discovers node 7, it finds already-discovered node 6, and already-discovered node 5, but nodes 5,6 and 7 have predecessors 2,3, and 4, respectively. However, there is a triangle on nodes 5,6,7 so this is a counterexample.

(b) If A is the adjacency matrix for graph G (which has a 1 in entry i, j if edge (i, j) is present in G, and a 0 otherwise), then A^2 contains a non-zero entry iff there is a 2-edge path from node i to j. This is because $A^2[i, j] = \sum_k A[i, k]A[k, j]$, and if there exists k such that i, k and k, j are edges, then this value will be non-negative (and vice versa).

This suggests the following algorithm: first, form A from the input (which is in adjacency list format). This is easy to do in $O(n+m) = O(n^2)$ time. Then use Strassen's algorithm to compute A^2 in time $O(n^{2.81})$. Finally, for each entry (i, j) in A, we check whether edge (i, j) is present, and if so, we have identified a triangle using the 2-edge path from i to j and the single ehdge i, j.

2. (a) Clearly $\mathcal{I}'$ is closed under taking subsets, and assuming every e in the universe is mentioned (as per the clarification), it is also easy to see that the empty set is present in $\mathcal{I}'$.

Finally, if A, B are subsets of $\mathcal{I}'$ and $|A| > |B|$ then by definition there exist set $A' = A \cup \{e\}$ and $B' = B \cup \{e\}$ are subsets of $\mathcal{I}$, which is a matroid. Thus there exists an element $x \in A' \setminus B'$ (which can't be e since e is in both A' and B') such that $B \cup \{x\} \in \mathcal{I}$. But then $B \cup \{x\} \setminus \{e\}$ is in $\mathcal{I}$ by definition. So the third property holds, and the first two properties are clear, so $\mathcal{I}'$ is a matroid.

(b) Observe that the spanning trees of G (which is connected) are exactly the bases of the graphic matroid. If we apply part (a) for an edge e, we obtain the matroid whose bases are exactly the subsets of G's edges that, together with e, form a spanning tree (by how I' is defined). Applying part (a) repeatedly, then, to each of the edges of F in turn, results in the matroid whose bases are exactly those subsets of G's edges that, together with F, form a spanning tree. This is as desired.

Now, we have weights on the edges, and we replace each weight w with $L - w$, for some fixed L that is set to be equal to the largest edge weight. The result is all non-negative weights, and, as we argued for the graphic matroid in a previous problem set, maximizing the weight of a bases under these weights results in a basis that minimizes the sum of the original weights.

To actually implement the algorithm, we sort the edges with their modified edge weights in decreasing order. This takes $O(m \log m)$ time. We then process each edge, attempting to add it to the current subset of edges (this is just implementing the greedy algorithm to find a maximum-weight independent set in the matroid). To check if it can be added, we need to check if it forms a cycle, and for this we can use a union-find data structure, which we "prime" by adding the edges of F initially. The running time for the union-find operations is at most $O(m \log^* m) \leq O(m \log m)$.

3. (a) We simply compute $a_i + b_j$ for all i, j (using $O(n^2)$ operations). Then we sort the third list in $O(n \log n)$ operations. Finally, for each sum $s = a_i + b_j$ computed in the first step, we perform a binary search to see if $-s$ is present in that list. If it is, we have identified i, j, k such that $a_i + b_j + c_k = 0$; if we get to the end of the n^2 pairs without finding such an i, j, k, then none exist. The overall running time is $O(n^2 \log n)$ as required.

   (b) First, observe that by adding M to all of the integers, we transform the problem into one where we are seeking i, j, k such that $a_i + b_j + c_k = 3M$ and all of the integers are non-negative. Now, let S be the subset of $[0, 2M]$ containing those a present in the a-list, and let T be the same with respect to the b-list. As suggested, form the polynomials $A(X) = \sum_{i \in S} X^i$ and let $B(X) = \sum_{i \in T} X^i$. Note that in the product AB, the coefficient on X^k is exactly

$$|\{i \in S, j \in T : i + j = k\}|.$$

   As before we sort the third list in time $O(n \log n) \leq O(M \log M)$. We compute the product AB in time $O(M \log M)$. Now, we go through the coefficients of AB one by one, and if X^k has a non-zero coefficient, we perform a binary search for $3M - k$ in the third list. If it is, we have identified i, j, k such that $a_i + b_j + c_k = 3M$; if we get to the end of the $O(M)$ coefficients of AB without finding such an i, j, k, then none exist. The overall running time is $O(M \log M)$ as required.

4. We follow the same strategy as we used to find the closest pair of points. First, we sort the n points by x coordinate, and scan through them from left-to-right to find a dividing line that splits them into two sets of size $n/2$ each. We recursively identify the lightest triangle on the left side, with total distance d_L and on the right side, with total distance d_R. Set $d = \min\{d_L, d_R\}$. As before, we observe that for a triangle that crosses the boundary, we can limit our consideration to points within distance d of the midline, because any points farther than than can only participate in triangles that are heavier than the ones already discovered. We scan once more through the points by x coordinate to select those points within distance d of the midline. Then these points are sorted by y coordinate. As with the closest-pair problem, we argue that we only need to consider points that are 80 positions or fewer separated in this latter sorted list. The main observation is that within each $d/5 \times d/5$ box (similar to the figure from lecture), there can be no more than 2 points. Any two points in such a box have distance at most $c = \sqrt{2 \cdot (d/5)^2}$ and so any 3 points in the box would constitute a triangle of weight (less than) $3c < d$, contradicting d_L or d_R being the smallest weight triangle on the each side. Now, we observe that if two points are separated by 5 rows

of these boxes, then their distance is already at least d, so they cannot both participate in the lightest triangle. So we only consider points that are separated by 4 or fewer rows, and there are 10 boxes in each row (and only 2 points per box at most). Overall we have to look ahead $4 \cdot 10 \cdot 2$ positions in the list at most, and each time, consider all triples of points in that block of at most 80. This contribute only a large constant to the time to scan through the list of at most n points. In the end, we report the lightest triangle from among the one found on each side, and the one straddling the middle. So the recurrence ends up being the same as for closest-pair, namely, $T(n) = 2T(n/2) + O(n \log n)$, for an overall running time of $O(n \log n)$ as desired.

5. As suggested, fix a vertex x and define $T(S, v)$, where S is a subset of vertices and v is an individual vertex, to be TRUE if there exists a path starting at x, passing through all of the vertices of S, visiting each exactly once, and ending at v. In our definition neither x nor v are included in S. It is easy to see that $T(\emptyset, v)$ is TRUE iff there is an edge from x to v. In general, we have that $T(S, v)$ is TRUE iff there exists $u \in S$ such that (u, v) is an edge, and $T(S \setminus \{u\}, u)$ is TRUE (since we can take the path from x to u and then just add the edge (u, v) to its end). This suggests a dynamic programming algorithm. We initialize $T(\emptyset, v)$ for all v by checking whether (x, v) is an edge. This takes $O(n^2)$ time overall. Then, we fill in the table by increasing order of $|S|$. Filling in an individual table entry entails running through each potential u and consulting table entry $T(S \setminus \{u\}, u)$ if (1) there is an edge (u, v) and (2) u is in set S. If we store the graph as an adjacency matrix, and the set in each table cell as a 0/1 array, each of these conditions can be checked in constant time. So, it takes $O(n)$ time to fill in each table cell, and there are $O(n2^n)$ table cells overall. In the end we report that there is a Hamilton cycle iff $T(V \setminus \{x, v\}, v)$ is true for some v for which there is an edge (v, x). The overall running time is $O(n^2 2^n)$ as desired.