College of Computer & Information Science
Northeastern University
Fall 2017

CS5800
Algorithms
November 16, 2017

Solution Sketch for Some Practice Problems for Midterm 2

1. Depth-First Search

Let $G = (V, E)$ be a directed graph, and let $u, v \in V$ be two vertices such that there exists a path P_1 from u to v and a path P_2 from v to u. Show that in any execution of the DFS algorithm the vertices u and v are in the same component of the DFS forest.

Answer: We give two proofs.

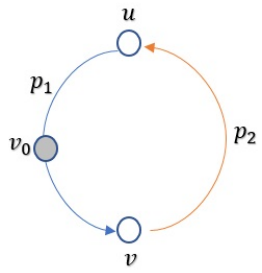
Proof 1: The key idea is based on the following property of DFS. A call to DFS from a source vertex s results in a tree rooted at s consisting of all hitherto undiscovered or white vertices reachable from s. To prove this, consider any undiscovered vertex v reachable from s at the start of DFS(s). Let P be a path from s to v. If v does not end up in the DFS tree under s, then there must be some vertex on the path that is not included in the DFS tree under s. Let x be the vertex in P closest to s that is not included, and let y the vertex preceding x along this path. The vertex y is discovered and finished yet x is not discovered, clearly leading to a contradiction.

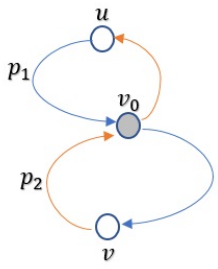
Once we have established the above property, let us consider the first instant one of u or v is visited in the DFS, and let s denote the source vertex from where DFS was initiated. Clearly, u and v are both reachable from s since one of them is and there is a path between u and v in each direction. By the above property, it follows that u and v must both end up in the DFS tree of s.

Proof 2: We use the coloring terminology here. Consider the cycle C that is defined by P_1 and P_2:

$$C: \; u \overset{P_1}{\rightsquigarrow} v \overset{P_2}{\rightsquigarrow} u$$

Let v_0 be the first vertex in the cycle that turns gray (first one to be discovered). It follows that there exists a white path (a path of undiscovered vertices) from v_0 to u and from v_0 to v. Therefore it follows from the white path theorem that u and v are descendants of v_0 in the DFS forest and therefore they are in the same DFS component.





2. MSTs and Shortest Paths

Let G be an undirected connected graph with positive weights on edges. For each of the following claims, indicate whether it is true or false. Briefly justify your answers.

(a) If T is a minimum spanning tree of a weighted connected undirected graph G, then a minimum-weight edge of T is also a minimum-weight edge of G.

Answer: True. Let e be a min-weight edge of T and e' be a min-weight edge of G. If $w(e) = w(e')$, then the statement is true. Otherwise, we have $w(e) > w(e')$; we will now derive a contradiction. The edge e' is not in T since otherwise, e' would have been minimum weight in T. So if we add e' to T, we get a cycle. Every other edge in this cycle has larger weight than e' since the min-weight of any edge in T is higher than $w(e')$. Hence, dropping any of these edges would yield a new spanning tree of weight smaller than T, a contradiction.

(b) Let G be a weighted directed graph and s be a vertex in G. Let p denote a shortest path from s to t. If we increase the weight of every edge in the graph by 1, then p remains a shortest path from s to t.

Answer: False. Consider two vertex-disjoint paths from s to t. One consisting of three edges, each of length 1, and the other consisting of one edge of length 4. Clearly, the former path is shortest. If we increase every edge weight by 1, the latter path becomes shortest.

3. Subsequences

Recall that we say that S is a *subsequence* of T if S can be obtained from T by deleting some characters from T. For instance, the sequence A, T, C, T, G is a subsequence of $G, \underline{A}, A, A, \underline{T}, \underline{C}, G, G, \underline{T}, T, \underline{G}$. (Positions indicating the subsequence property are underlined.)

Give an algorithm that takes as input two sequences S and T and determines whether S is a subsequence of T. State the worst-case running time of your algorithm. The more efficient your algorithm is in terms of its worst-case running time, the more credit you will get.

Answer: We give a greedy algorithm that finds the first element in T that matches $S[1]$, then the first element after this in T that matches $S[2]$, and so on. In the following algorithm, we use the array M to denote the matches, i to denote the current position in T and j the current position in T.

$i = j = 1$
while $i \leq n$ and $j \leq m$:
    If $T[i] = S[j]$:
        $M[j] = i$
        $i = i + 1$; $j = j + 1$
    else $i = i + 1$
endwhile
if $j = m + 1$ return S
else output "S is not a subsequence of T"

The running time is $O(n)$ since i increases by 1 in each iteration, and each iteration takes $O(1)$ time. For completeness and for exhibiting a straightforward formal correctness proof, we establish the correctness of the algorithm using two key arguments. First, if the algorithm returns a match, then the match is correct. This is straightforward since the algorithm matches each element of S one by one, and returns a match only if each element matches a distinct element of T in order.

Next, we want to show that if there is a match, then the algorithm will find it. Suppose S is the same as $T[k_1], T[k_2], \ldots, T[k_m]$. We prove by induction that $M[j] \leq k_j$ for all $j = 1, \ldots, m$.

Claim 1. *For each $j = 1, \ldots, m$, the algorithm finds a match $M[j]$ and $M[j] \leq k_j$.*

Proof: The proof is by induction on j. For the induction base, consider $j = 1$. In our algorithm, we have $M[1]$ to be the first index in T such that $S[1] = T[M[1]]$, so $M[1] \leq k_1$.

For the induction step, we consider $1 < j \leq m$, and assume that our algorithm found the match $M[j-1]$ such that $S[j-1] = T[M[j-1]]$ and $M[j-1] \leq k_{j-1}$. The algorithm sets $M[j]$ to be the first index after $M[j-1]$ such that $S[j] = T[M[j]$. Since $S[j] = T[k_j]$ and $k_j \geq k_{j-1} \geq M[j-1]$, it follows that $M[j] \geq k_j$, completing the induction step, and the proof of the claim. □

The above claim shows that if there is a match, the algorithm always find it, completing the proof of correctness of the algorithm.

4. Directed cycle of minimum weight

Given a directed graph $G = (V, E)$ with positive weights on each edge, design an efficient algorithm to determine a directed cycle in G of minimum total weight. If no directed cycle exists in G, then your algorithm must indicate so.

Analyze the worst-case running time of your algorithm. The more efficient your algorithm is in terms of its worst-case running time, the more credit you will get.

Answer: Fix a vertex v. We will give an algorithm for finding the a directed cycle containing v of minimum total weight, if it exists. Repeating this for all vertices will then yield the final algorithm.

To find a directed cycle containing v of minimum total weight, we replace vertex v with two vertices v_i and v_o and edges of the form (u, v) by (u, v_i) and of the form (v, u) by (v_o, u) with their original weights. Now a directed cycle containing v in G of weight W corresponds to a path from v_o to v_i of weight W in the revised graph. So we solve the shortest path problem in the revised graph from v_0 to v_i.

The running time is n times the running time of any single-source shortest path algorithm (say Dijkstra): $O(nm \log n)$.

Another algorithm is to run Floyd-Warshall's all-pairs shortest paths algorithm. And then, for each edge (u, v), determine the shorest cycle containing (u, v), if any, by adding $w(u, v)$ to the length of the shortest path from v to u. Taking the minimum of the preceding quantity, over all edges (u, v), then yields the shortest directed cycle. This takes running time $O(n^3)$. Depending on the value of m, this may be better or worse than the previous algorithm.

5. Maximum weight independent set for a path

A path graph G is simply an undirected graph consisting of the vertex set $V = \{v_1, v_2, \ldots, v_n\}$ and the edge set

$$E = \{(v_i, v_{i+1}) : 1 \leq i \leq n-1\}.$$

An *independent set* of G is a subset I of V such that there is no edge in E between any pair of vertices of I. We also associate a positive weight w_i with vertex v_i. The weight of any subset S of vertices is simply the sum of the weights of the vertices in S.

For example, consider a path graph G with five vertices v_1, v_2, v_3, v_4, v_5, with weights 1, 9, 6, 3, and 7, respectively and four edges $(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5)$. The set $\{v_2, v_5\}$ is an independent

set with weight 16, while the set $\{v_1, v_3, v_4\}$ is not an independent set.

Give an efficient algorithm to determine a maximum weight independent set of a given weighted path graph G.

Answer: We denote $m(k)$ as the subproblem of the original problem, which represents finding the maximum-weight independent set from the first k vertices and the subgraph consisted by them.

Initially we have:

$$\begin{aligned} m(0) &= 0 \\ m(1) &= \max\{w_1, 0\} \end{aligned}$$

Note, we need to consider if w_1 is positive.

Recursively, when we consider $m(k)$, we have two possibilities:

- If we choose v_k, then we have $m(k) = m(k-2) + w_k$(if w_k is negative, we have $m(k) = m(k-2)$)
- If we don't choose v_k, then we have $m(k) = m(k-1)$

So we have the following equations:

$$m(k) = \max\{m(k-1), m(k-2) + w_k\}$$

Thus we need only one-pass to traverse all the $m(k)$, and got the solution in $O(n)$ time.

6. Planning a road trip

You are planning a road trip that starts at location 0 and ends at location n, going through locations 1, 2, ..., $n-1$ in that order. Due to health and logistical reasons, you are able to visit at most k locations a day. If at the end of a day, you are at a location i, then you need to shell out d_i dollars to stay at a hotel in location i. (If you reach a location i on a particular day and stay overnight, then this location is counted toward the limit of k for the day you reach i, not for the following day. Also, location 0 does not count toward the limit for the first day.)

Given the significant differences among the hotel costs, you would like to find a schedule that minimizes the total hotel cost, while maintaining the constraint that you visit at most k locations in a day.

Give an $O(nk)$ time algorithm to determine the minimum total hotel cost you will incur for your road trip. It is sufficient to give a recurrence, and describe your algorithm briefly in words. For partial credit, you may give an algorithm that is less efficient.

Answer: Let $M(i)$ denote the minimum cost for visiting the locations 0, 1, ..., i, ending at location i. For the base case, we set $M(0) = 0$.

$$M(i) = \min_{1 \le j \le \min\{i,k\}} (M(i-j) + d_{i-j}).$$

We convert this recurrence to an algorithm by looping for i from 1 to n, and adding an inner loop for j from 1 to $\min\{i, k\}$. For computing the ith entry, we perform at most k lookups and additions. And we have n entries in M. So the running time is $\Theta(nk)$.

7. Alternating red-blue paths

Let $G = (V, E)$ be a directed graph in which each vertex has been assigned a color, either red or blue. A directed path in G is called an *alternating red-blue path* if and only if no two consecutive vertices on the path have the same color. Give an efficient algorithm that determines for *all* pairs of vertices u, v in V whether v is reachable from u via an alternating red-blue path.

Analyze the worst-case running time of your algorithm. The more efficient your algorithm is in terms of its worst-case running time, the more credit you will get.

Answer: The algorithm is simple. Remove all red-red and blue-blue edges. For each vertex u, run depth-first search from u to determine all vertices reachable from u.

Running time is $O(n(n+m))$.

One could do better in practice by first computing strongly connected components and then traversing the dag obtained. But that does not yield a better running time in the worst case.

8. Longest Non-Increasing Subsequence

Design an algorithm that gets a sequence of distinct integers $a_1, a_2, \ldots, a_n$ and finds a longest non-increasing subsequence.

Answer: We provide two solutions to the problem.

1. Given that we have already considered the longest common subsequence problem (LCS) in class, there is an easy reduction from the longest monotically increasing subsequence problem (LMIS) to LCS. Let X be the given sequence of n numbers. Let Y be the set of numbers in X sorted in increasing order, in which we remove any duplicates. Thus, the length of Y is at most n. Now we claim that the LMIS of X is the same as the LCS of X and Y. If the claim holds, then the LMIS of X can be obtained by running the LCS algorithm which will take $\Theta(n^2)$ time.

   The above claim follows from the following two observations: (i) any monotonically increasing subsequence of X is a common subsequence of X and Y, and (ii) any common subsequence of X and Y is a monotonically increasing subsequence of X. Therefore, a *longest* monotonically increasing subsequence of X is the same as a *longest* common subsequence of X and Y.

2. An alternative approach is by first principles. Let LMIS[i] denote the LMIS of X that ends at $X[i]$. We now give a recurrence relation for calculating LMIS[i] as follows:

$$\text{LMIS}[i] = \begin{cases} X[i] & \text{if } i = 1. \\ \text{append}\left(\max_{1 \le j < i}\{\text{LMIS}[j] : X[j] < X[i]\}, X[i]\right) & \textit{otherwise.} \end{cases}$$

   In the above equation, the "max" function returns the set largest in size, if one exists; otherwise, it returns the empty set. We justify the above equation as follows. If $i = 1$, then clearly the LMIS ending at $X[i]$ trivially consists of $X[i]$ only. Otherwise, the LMIS ending at $X[i]$ is equal to $X[i]$ appended to the longest of the LMISs ending at any $X[j]$ that satisfies $X[j] < X[i]$.

   The algorithm consists of simply computing LMIS[i] iteratively, i going from 1 to n. Since the computation of any LMIS[i] requires the computation of the maximum of at most n sets, the running time of the algorithm is $n \cdot O(n) = O(n^2)$.

Remark: By using a data structure like a balanced binary search tree, one can reduce the running time of the algorithm to $O(n \log n)$.