Section 10: Solutions

1. Reduction

Consider the following problems:

HAM-PATH **Input**: A **directed** graph G **Output**: True if there is a Hamiltonian Path in G, that is a path that visits each vertex **exactly** once.
HAM-CYCLE **Input**: A **directed** graph G **Output**: True if there is a Hamiltonian Cycle in G, that is, a path $v_0, v_1, ..., v_n$ that visits each vertex **exactly** once along with the edge (v_n, v_0).

Suppose that HAM-PATH is NP-hard. Use that fact to show HAM-CYCLE is NP-hard.

Solution:

We will show that HAM-PATH $\leq_P$ HAM-CYCLE, that is we will use a library function designed for HAM-CYCLE problem to solve the HAM-PATH problem.

Let G be an input for the HAM-PATH problem. Create the graph H, as follows: Starting from G, add two vertices u and v, and the edge (u, v). For every vertex w that was in G, add the edges (w, u) and (v, w). Call the HAM-CYCLE library on H and return its result (unchanged) as the answer for HAM-PATH on G.

Correctness:
Suppose that G has a HAM-PATH, $w_1, w_2, ..., w_n$. Then in H, note that $w_1, w_2, ..., w_n, u, v$ is a path that visits every vertex in H (since we copied H and then added edges from all w into u and from u to v. Since we also included edges from v to every w, there is also the edge (v, w_1) in H and there is a Hamiltonian Cycle in H, so our reduction correctly returns YES.

Conversely suppose that H has a Hamiltonian cycle. Observe that since u has only one outgoing edge and v has only one incoming edge, the cycle must include the edge (u, v). Thus the cycle can be written as $u, v, w_1, ..., w_n$ (since cycles have the edge from the last vertex to the first vertex, we can rewrite the cycle with u appearing first). Since all edges in H but not in G have u or v as an endpoint, the edges (w_i, w_{i+1}) are from G for all i, and we have that $w_1, ..., w_n$ is a Hamiltonian path in G.

Running time:
Copying the graph, adding two vertices and $2n + 1$ edges can be done in polynomial time, and we make only one call to the library.

2. Max-Flow/Min-Cut

A group of traders are leaving Switzerland, and need to convert their Francs (the local currency) into various international currencies. There are n traders and m currencies. Trader i has T_i Francs to convert. The bank has B_j Francs worth of currency j. Trader i is willing to trade as much as C_{ij} of his Francs for currency j. (For example, a trader with 1000 Francs might be willing to convert up to 700 of his Francs for USD, up to 500 of his Francs for Japaneses Yen, and up to 500 of his Francs for Euros).

Assuming that all traders give their requests to the bank at the same time, describe an algorithm that the bank can use to satisfy the requests (if it can).

Solution:

This is set up as a flow network, with Francs flowing from the source s to the sink t. A row of vertices $t_1, ..., t_n$ represents the traders, and a row of vertices $b_1, ..., b_m$ represents the currency held by the bank. The edge (s, t_i) has capacity T_i which gives the amount trader i wants to change. The edge (t_i, b_j) has capacity C_{ij} giving the maximum number of Francs i wants to trade into currency j. Finally, the edge (b_j, t) with capacity B_j gives the limit on the amount of currency j that is available. If there is a flow f in the network with $|f| = \sum_i T_i$ then all traders are able to convert their currencies. (An alternate solution which reverses the source and sink, and has currency flow to from the banks to traders is also valid.)

3. DP Non-Adjacent LCS

The sequence $C = c_1, ..., c_k$ is a *non-adjacent subsequence* of $A = a_1, ..., a_n$, if C can be formed by selecting non-adjacent elements of A, i.e., if $c_1 = a_{r_1}, c_2 = a_{r_2}, ..., c_k = a_{r_k}$, where $r_j < r_{j+1} - 1$. The non-adjacent LCS problem is given sequences A and B, find a maximum length sequence C which is a non-adjacent subsequence of both A and B.

This problem can be solved with dynamic programming. Give a recurrence that is the basis for a dynamic programming algorithm. You should also give the appropriate base cases, and explain why your recurrence is correct.

Solution:

The recurrence for OPT(i, j) is giving the length of the longest common non-adjacent subsequence of $a_1, ..., a_i$ and $b_1, ..., b_j$. The key idea is that we match a_i and b_j , then we cannot match a_{i-1} or b_{j-1}. This alters the recurrence from the LCS problem by making the dependency on OPT$(i-2, j-2)$ if there is a match. The recurrence becomes:

$$\text{OPT}(i,j) = \begin{cases} \max(\text{OPT}(i-2,j-2)+1, \text{OPT}(i-1,j), \text{OPT}(i,j-1)) & \text{if } a_i = b_j \\ \max(\text{OPT}(i-1,j), \text{OPT}(i,j-1)) & \text{if } a_i \neq b_j \end{cases}$$

To handle the base cases appropriately, we need to ensure that we the don't access indices that are out of bounds. The most convenient way to do this is just to define OPT$(0, j) =$ OPT$(i, 0) =$ OPT$(-1, j) =$ OPT$(i, -1) = 0$ so we can avoid special cases in the recurrence.

4. DP Electoral College

The problem is to determine the set of states with the smallest total population that can provide the votes to win the electoral college. Formally, the problem is:

Let p_i be the population of state i, and v_i the number of electoral votes for state i. All electoral votes of a state go to a single candidate, so the winning candidate is the one who receives at least V electoral votes, where $V = \lfloor (\sum_i v_i)/2 \rfloor + 1$. Our goal is to find a set of states S that minimizes the value of $\sum_{i \in S} p_i$ subject to the constraint that $\sum_{i \in S} v_i \geq V$.

(a) The dynamic programming solution for this problem involves computing a function OPT where OPT(i, v) gives the minimum populations of a set of states from $1, 2, ..., i$ such that their votes sum to exactly v. Give a recursive definition of OPT and an explanation as to why it is correct.

Solution:

OPT$(i, v) = \min\{\text{OPT}(i-1, v), \text{OPT}(i-1, v-v_i) + p_i\}$ The first term corresponds to state i not included in the vote total, and the second term corresponds to the state i included in the vote total.

(b) What are the base cases for your function OPT.

Solution:

OPT$(0, 0) = 0$ and OPT$(0, v) = \infty$ for $v \geq 1$