CPSC 320 2021S2: Tutorial 9 Solutions

1 Hospitals and Residents

A group of residents each needs a residency in some hospital. A group of hospitals H each need some number (one or more) of residents from some set R, with some hospitals needing more and some fewer. Each group has preferences over which member of the other group they'd like to end up with. The total number of slots in hospitals is exactly equal to the total number of residents.

Our problem, which we'll call RPH, is to fill the hospitals slots with residents in such a way that no resident and hospital that weren't matched up will collude to get around our suggestion (and give the resident a position at that hospital instead).

Let $n = |R|$ (the size of the set of residents). Note that $|H| \leq |R|$, but H may be much smaller. Let $s(h)$ denote the number of slots in hospital h, where $n = \sum_{h \in H} s(h)$. That is, there are exactly enough slots for the residents.

In a *valid* solution, each resident must appear in exactly one tuple (be paired with one hospital), while each hospital h must appear in exactly $s(h)$ tuples (be paired with as many residents as it has slots). An *instability* arises when some hospital h is matched with residents $H_h = \{r'_1, r'_2, ...r'_{s(h)}\}$ and some resident r is matched with h', such that r prefers h to h' and h prefers r to its lowest-ranked member in H_h.

Describe an approach to solve RHP by reducing it to some problem B that you know already has an efficient algorithm.

1. Choose a problem B to reduce to.

   SOLUTION: Let's reduce to SMP.

2. Design an algorithm to transform any instance I of RHP into an instance I' of B.

   SOLUTION: We'll split **every** hospital h into $s(h)$ "hospital-slots". Since we know $\sum_{h \in H} s(h)$ is exactly the number of residents, this will give us a set of hospital-slots of the same size as the number of residents.

   Each of these hospital-slots needs a preference list. Let hospital-slot have the same preference list as the hospital it came from.

   Also, we'll have each resident replace a hospital h in their preference list with $h_1, h_2, ..., h_k$ for $k = s(h)$. That is, where they had hospital h, they now have one entry in order for each hospital-slot broken off of h (but all are worse than the hospital-slots coming from hospitals the resident preferred and better than those from hospitals the resident liked less).

   At that point, we have an SMP instance.

3. Design an algorithm to transform a solution S' for I' of B into a solution S for instance I of. RHP.

SOLUTION: The Gale-Shapley algorithm will give us back a stable, perfect matching M to our SMP instance I'. With the solution representation we used, the only thing different about M from a possibly-stable RHP solution would be the subscripts on the hospital-slots. If we erase those, then since each hospital-slot had one match and each hospital had $s(h)$ hospital-slots, each hospital in the RHP solution will now have $s(h)$ matches, as we expect. The residents will still each have exactly one match, since we haven't changed them.

4. Explain why your reduction produces a correct solution for any RHP instance: if S' is a good solution for the instance I' produced by part (i) of your reduction, then part (ii) of your reduction transforms S' info to a good (i.e., valid and stable) solution S for the original instance I of RHP.

   SOLUTION: First, we already showed that any good solution S' (i.e., stable matching) for instance I' of SMP follows the basic rules of RHP, i.e., each hospital is partnered with exactly the right number of residents (and each resident with exactly one hospital). So solution S must be valid.

   To show that S is stable, let's prove the contrapositive: Assuming that S is unstable, we'll show that S' must also be unstable, contradicting our assumption that S' is good.

   Since S is unstable, there must be some pair h and r that cause the instability. (Maybe multiple, but we don't care about that.) In particular: h is matched with residents $H_h = \{r'_1, r'_2, ...r'_{s(h)}\}$ and resident r is matched with h' such that r prefers h to h' and h prefers r to the member of H_h it least prefers (the 'worst' member).

   The pairing of r with h' must have come from S's pairing of r with one of h''s slots, say h'_k. Let's also look at S's pairing of h with its least-preferred partner r'. We don't know which slot of h's that is, but we'll say it's h_j. We'd like to see that just as r and h form an instability with respect to $'$, r and h_j form an instability with respect to S'.

   Do they form an instability?

   Well, r prefers h to h' in instance I of RHP. The "cloning" we did to split hospitals into hospital-slots in instance I' keeps all the slots of a hospital together. So, in instance I', r must prefer all slots of h to all slots of h', and so r prefers h_j to h'_k.

   Similarly, all of h's slots in I' have the same preferences as h in instance I. So, just as h prefers r to r' in I', h_j must prefer r to r' in I.

   So, r and h_j do indeed constitute an instability with respect to S.

   Since the SMP solution S' is unstable if the RHP solution S is unstable, we can conclude that the RHP solution is **stable** if the SMP solution is stable. We know the SMP solution S' **is** stable, which means the RHP solution S is as well!.

2 Edit distance

Let $X = x_1x_2...x_n$ and $Y = y_1y_2...y_m$ be strings over some fixed alphabet Σ.

- An *insertion* of "a" in X at position i yields the string $x_1x_2...x_iax_{i+1}...x_n$ (of length $n+1$).
- A *deletion* at position i of X yields $x_1...x_{i-1}x_{i+1}...x_n$ (of length $n-1$).

For example, if $X =$ "tree" then an insertion of "h" at positions 0 or 1 respectively yield "htree" or "three" respectively, and a deletion at position 2 of X yields "tee".

The *edit distance* between X and Y is the minimum number of insertions and deletions to get string Y from string X (or vice versa).

1. What is the edit distance between $X =$ "fence" and $Y =$ "wicked"?

   SOLUTION: The edit distance is 7. The following sequence of insertions and deletions work (we show the resulting string after each step in parenthesis):

   delete "f" at position 1 (ence),
   insert "i" at position 1 (ience),
   insert "w" at position 1 (wience),
   delete "e" at position 3 (wince),
   delete "n" at position 3 (wice),
   insert "k" at position 4 (wicke),
   insert "d" at position 6 (wicked).

   Fewer than 7 insertions and deletions won't work: the letters "f" and "n" don't appear in "wicked" and so must be removed; just one "e" appears in "wicked" and so one of the "e"s in "fence" must be removed; and the letters "w", "i", "k" and "d" appear in "wicked" but not "fence" and so must be added.

2. Let $\mathrm{ED}(n, m)$ be the edit distance between strings $X[1..n]$ and $Y[1..m]$. Express $\mathrm{ED}(n, m)$ as a recurrence. Make sure to include appropriate base cases. Hint: think about the longest common subsequence problem discussed in class.

   Solution. One base case is $\mathrm{ED}(0, m) = m$, since m letters of Y must be added if the string X is the empty string. Another is that $\mathrm{ED}(n, 0) = n$, since the n letters of X must be removed if Y is the empty string. When $n > 0$ and $m > 0$ we have:

$$\mathrm{ED}(n, m) = \begin{cases} \mathrm{ED}(n-1, m-1), & \text{if } X[n] = Y[m], \\ 1 + \min\{\mathrm{ED}(n-1, m), \mathrm{ED}(n, m-1)\}, & \text{otherwise.} \end{cases}$$

   In the first case, since the last letter of X and Y match, that letter does not need to be inserted or deleted and the remaining subproblem has size $(n-1, m-1)$. Otherwise, an optimal solution must either delete the last letter of X or insert the last letter of Y, which adds 1 to the edit distance, and the remaining subproblem has size $(n-1, m)$ or $(n, m-1)$, respectively.

3. We already have efficient algorithms to find the LLCS of two strings. Can we use that algorithm to find the edit distance between X and Y? (That is, can you find a *reduction* from the Edit Distance problem to the LLCS problem? Recall that in the worksheet on breadth first search, we showed a reduction from Shortest Paths to Breadth First Search.)

   Solution. The edit distance between X and Y is equal to $n + m - 2\mathrm{LLCS}(X, Y)$. So, given instance X, Y of the edit distance problem, we can simply compute the $\mathrm{LLCS}(X, Y)$, and then subtract this quantity from $n + m$.

4. The *generalized edit distance* from X to Y allows one additional operation:

   - A *substitution* of letter a at position i yields the string $x_1x_2...x_{i-1}ax_{i+1}...x_n$ (of length n).

   For example, if $X =$ "tree" then a substitution of "i" at position 3 yields "trie".

   Let $\mathrm{GED}(n, m)$ be the generalized edit distance between $x_1x_2...x_n$ and $y_1y_2...y_m$. Express $\mathrm{GED}(n, m)$ as a recurrence. Make sure to include appropriate base cases.

   SOLUTION: The base cases for GED are the same as for ED, since letter substitution is not useful when either X or Y is the empty string. That is, So, $\mathrm{GED}(0, m) = m$ and $\mathrm{ED}(n, 0) = n$. When $n > 0$ and $m > 0$ we have:

$$\mathrm{GED}(n, m) = \begin{cases} \mathrm{GED}(n-1, m-1), & \text{if } x_n = y_m, \\ 1 + \min\{\mathrm{GED}(n-1, m), \mathrm{GD}(n, m-1), \mathrm{GED}(n-1, m-1)\}, & \text{otherwise.} \end{cases}$$

   If the last two letters match, no insertion, deletion or substitution is needed to reduce the problem size from (n, m) to $(n-1, m-1)$. Otherwise, either the last letter of X must be deleted, or the last letter of Y must be inserted, or y_m must be substituted for x_n, and whichever happens increases the edit distance by 1. The remaining subproblem has size $(n-1, m)$, $(n, m-1)$, or $(n-1, m-1)$, respectively.