CPSC 320 Sample Solution, Reductions and Resident Matching: A Residentectomy

September 16, 2016

A group of residents each needs a residency in some hospital. A group of hospitals each need some number (one or more) of residents, 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.

We want 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).

Definitions: An *instance* of a problem is a particular input drawn from the space of possible inputs the problem allows. For example, the 4-element array $[5, 1, 4, 3]$ is an instance of the problem of sorting arrays of integers.

A *reduction* from problem P_1 to problem P_2 is an algorithm that solves P_1 using (but without needing to define) an algorithm A_{P_2} that solves P_2. Often, we put some restriction on the number of calls to A_{P_2} (e.g., a polynomial number).

We'll call a *simple reduction* a reduction in which only one call is made to A_{P_2}. This is the most common case we'll run into, and it's often easiest to think about this case by proceeding through these steps:

1. Change an instance of P_1 into an instance of P_2 by hand.
2. Change the corresponding solution to P_2 into a solution to P_1 by hand.
3. Design an algorithm to change instances of P_1 into instances of P_2.
4. Design an algorithm to change corresponding solutions to P_2 into solutions to P_1.
5. Prove that if the solution to the P_2 instance is correct, so too is the solution your algorithm creates for the original P_1 instance. (**Or equivalently**, if the P_1 solution generated is incorrect, then the P_2 solution must have been incorrect as well.)

1 Trivial and Small Instances

1. Write down all the **trivial** instances of RHP. We think of an instance as "trivial" roughly if its solution requires no real reasoning about the problem.

   SOLUTION: Certainly instances with 0 hospitals and 0 residents are trivial (solution: no matchings).

   Additionally, any time we have one hospital, no matter how big it is (and therefore how many residents there are), the solution will be trivial: place all residents with that one hospital.

2. Write down two **small** instances of RHP. Here's your first:

   SOLUTION: Here's one, but it could as well be an SMP instance.

```
r1: h1 h2     h1: r2 r1
r2: h2 h1     h2: r1 r2
```

The other can be even smaller, but not trivial:

SOLUTION: So, it's a good idea to make an instance that actually illustrates what's unique to our problem. (Otherwise, how will we know what to specify??) Here, the number in parentheses after a hospital indicates how many slots it has.

```
r1: h1 h2     h1 (1): r2 r1 r3
r2: h2 h1     h2 (2): r1 r2 r3
r3: h1 h2
```

3. Hold this space for another instance, in case we need more.

2 Represent the Problem

1. What are the quantities that matter in this problem? Give them short, usable names.

   SOLUTION: Generally speaking, these will be the same as in the SMP problem. A few differences: we'll let $n = |R|$ (the size of the set of residents). Note that $|H| \leq |R|$, but H may be much smaller. We need to know, for each hospital how, big it is. We'll say that $s(h)$ is the number of slots in hospital h.

2. Go back up to your trivial and small instances and rewrite them using these names.

   SOLUTION: Not really necessary here, but that parenthetical after each hospital certainly **was** necessary to give its number of slots!

3. Use at least one visual/graphical/sketched representation of the problem to draw out the largest instance you've designed so far:

   SOLUTION: We'll do the same thing we did for SMP, move the preferences to the outside so there's a "gutter" down the middle to draw lines for possible solutions:

```
h1 h2 :r1       h1 (1): r2 r1 r3
h2 h1 :r2       h2 (2): r1 r2 r3
h1 h2 :r3
```

4. Describe using your representational choices above what a valid instance looks like:

   SOLUTION: Again, this is much like SMP with some extra constraints, mostly focused on the s function that tells us how many slots a hospital has. In particular, for all $h \in H$, $s(h) > 0$. Further, $n = \sum_{h \in H} s(h)$. That is, there are exactly enough slots for the residents.

3 Represent the Solution

1. What are the quantities that matter in the solution to the problem? Give them short, usable names.

   SOLUTION: Much like with SMP, the pairings between hospitals and residents matter. There are at least two ways to handle the fact that every hospital can match with multiple residents. (1) Use the same format as SMP but changed from man/woman to hospital/resident (a set of tuples of hospital and resident) but allow each hospital to appear multiple times. (2) Use tuples of a hospital and a **set** of residents.

   We'll use (1).

2. Describe using these quantities makes a solution **valid** and **good**:

   SOLUTION: Crucially, 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). Otherwise, this isn't a matching of residents with hospitals at all.

   BUT, what makes this matching stable? It's not quite the same as SMP. In particular, a resident will still want to get out of her matching if she can match with a hospital she prefers, but under what circumstances will a hospital agree to give up **one of** its current residents for her? Clearly, it has to prefer her to someone it was assigned. And, if it prefers her to anyone it was assigned, it prefers her to the resident it was assigned that it least prefers.

   So, a good definition of an instability is "a hospital h matched with residents $H_h = \{r'_1, r'_2, \ldots r'_{s(h)}\}$ and resident r 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)."

3. Go back up to your trivial and small instances and write out one or more solutions to each using these names.

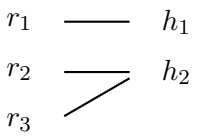
   SOLUTION: Let's actually write this here and just bring down the example:

```
h1 h2 :r1        h1 (1): r2 r1 r3
h2 h1 :r2        h2 (2): r1 r2 r3
h1 h2 :r3
```

   Using our notation, a solution might be $\{(h_1, r_1), (h_2, r_2), (h_2, r_3)\}$. (This happens to be the only stable solution to this instance.)

4. Go back up to your drawn representation of an instance and draw at least one solution.

   SOLUTION: Working on the same repeated instance, here's that solution:



4 Similar Problems

Give at least one problem you've seen before that seems related in terms of its surface features ("story"), problem or solution structure, or representation to this one:

SOLUTION: Obviously this is similar to SMP. It also has some similarities to USMP. (Perhaps adding fake entities to the hospital side will balance things out??)

5 Brute Force?

We have a way to test if something with the form of a solution (i.e., looks like a solution but may not be valid or good) is actually **valid** and **good**. (From the "Represent the Solution" step.)

1. Sketch an algorithm to produce everything with the "form" of a solution. (It will help to **give a name** to your algorithm and its parameters, *especially* if your algorithm is recursive.)

   SOLUTION: A bit less formally than last time, here's a solution sketch for an algorithm AllSolns(H, R, s):

   (a) If $|H| = 0$, return $\{\emptyset\}$.

(b) Otherwise, let r be the first element of R.

(c) And, let M be an empty set (of solutions).

(d) And, for each $h \in H$:

i. Produce new set $R' = R$.

ii. Produce new function $s' = s$ except that $s'(h) = s(h) - 1$.

iii. Produce new set H' as follows: if $s(h) = 0$, then $H' = H - \{h\}$; otherwise, $H' = H$. (In other words, strip out r and one slot from h, removing h if it gets to 0 slots.)

iv. For every solution $m \in \text{AllSolns}(H', R', s')$, add $\{(h, r)\} \cup m$ to M.

(e) Finally, return M

2. Choose an appropriate variable to represent the "size" of an instance.

   SOLUTION: n seems appropriate.

3. Exactly or asymptotically, how many such "solution forms" are there? (It will help to **give a name** to the number of solutions as a function of instance size.)

   SOLUTION: This is pretty messy. In particular, the first hospital can be grouped with any subset of the residents of size $s(h_1)$, and subsequent hospitals have that many fewer residents to "choose from". Overall, this looks something like $\frac{n!}{\prod_{h \in H}(s(h))!}$. The point here is that the larger the hospitals are, the fewer solutions there are. Indeed, if there's one hospital taking almost all the residents, we actually have a small solution space to explore. However, if there are even two roughly-equal sized hospitals, we're looking at $\frac{n!}{(n/2)!^2}$, which is super-exponentially huge.

4. Exactly or asymptotically, how long will it take to test whether a solution form is valid and good with a naïve approach? (Write out the naïve algorithm if it's not simple!)

   SOLUTION: Since we need to know the "worst" resident matched to each hospital, we might as well start by picking out that worst resident for each hospital. That takes $O(n) = O(|R|)$ time. Then, for each hospital/resident pair (of which there are $|H| \times |R|$), if they're not matched, we check whether they prefer each other to their partners (in the hospital's case, its "worst" partner).

   With efficient solutions to each step (see 2.3 of the textbook!), we should be able to do this in $O(|H| \times |R|)$ time, or if hospitals take only a reasonable (constant, actually) number of residents, about $O(n^2)$ time.

5. Will brute force be sufficient for this problem for the domains we're interested in?

   SOLUTION: Not unless some hospital is taking almost everyone!

6 Lower-Bound

In terms of instance size, exactly or asymptotically, how "big" is an instance? (That is, how long will it take for an algorithm just to read the input to the problem?)

SOLUTION: Much like with USMP, this is about $O(|H| \times |R|)$.

(We won't do more for now to lower-bound the problem.)

7 Promising Approach

Unless brute force is good enough, describe—in as much detail as you can—an approach that looks promising. **THIS TIME**, we use reduction:

1. Choose a problem P to reduce to.

   SOLUTION: Let's reduce to SMP.

2. Change an instance of RHP into an instance of P.

   SOLUTION: Here's our running example again:

```
h1 h2 :r1       h1 (1): r2 r1 r3
h2 h1 :r2       h2 (2): r1 r2 r3
h1 h2 :r3
```

   We need to put one more item on the right. We also need to make sure h_2 gets matched with two residents. It seems like we can solve both these problems at once by "splitting up" h_2:

```
h1 h2 :r1       h1:   r2 r1 r3
h2 h1 :r2       h2_1: r1 r2 r3
h1 h2 :r3       h2_2: r1 r2 r3
```

   Now, each "half" of h_2 is its own "woman" (or "man", if you prefer). This isn't an SMP instance yet, however. The residents don't have enough preferences! Well, each resident will want the two h_2 slots essentially the same, but we don't allow ties. So, we'll just order them arbitrarily. (Why not in numerical order?)

```
h1 h2_1 h2_2 :r1        h1:   r2 r1 r3
h2_1 h2_2 h1 :r2        h2_1: r1 r2 r3
h1 h2_1 h2_2 :r3        h2_2: r1 r2 r3
```

   Now **that** looks like an SMP instance.

3. Change the solution to P into a solution to RHP.

   SOLUTION: Running Gale-Shapley gives this solution: $\{(h_1, r_1), (h_{2_1}, r_2), (h_{2_2}, r_3)\}$.

   That's already very close to the solution we found by hand of $\{(h_1, r_1), (h_2, r_2), (h_2, r_3)\}$. It looks like we just need to erase the subscripts on the hospitals, since hospital-slots are no longer separate.

4. Design an algorithm to change RHP instances into instances of P.

   SOLUTION: This is probably the trickiest part. We need to eliminate the s function that tells us the size of hospitals. It also seems likely (as in USMP) that we'll want to make the two sets (residents and hospitals) have the same size.

   One way to accomplish both of those would be to make "clone" hospitals for every hospital that takes more than one resident. Actually, to make it easier to describe, let's say that will 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.

   However, we're not done. Each of these hospital-slots needs a preference list. **And**, the residents' preference lists must be augmented to include all these hospital slots instead of (as well as?) the original hospital.

   Well, we said "clone" for hospitals; so, let's try having each hospital-slot have the same preference list as the hospital it came from.

   There's no reason to think one "clone" is better than another, but we may as well have each resident replace a hospital h in their preference list with $h_1, h_2, \ldots, h_k$ 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 can name "residents" to be "men" and "hospitals" to be "women" (or vice versa), and we have an SMP instance.

5. Design an algorithm to change P's solutions into RHP solutions.

   SOLUTION: SMP will give us back a stable, perfect matching. With the solution representation we used, the only thing different about the SMP solution 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 partner and each hospital had $s(h)$ hospital-slots, each hospital in the RHP solution will now have $s(h)$ partners, as we expect. (The residents will still each have exactly one partner, since we haven't changed them!)

6. Prove either *if P's solution is correct, RHP's solution is correct* or the contrapositive.

   SOLUTION: I'll do a bit of both. First, we already showed that SMP's perfect matching gives us a matching that 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).

   Now, for the second constraint (stability), let's prove the contrapositive. So, assuming RHP's solution is unstable, we'll show that SMP's is unstable.

   Since RHP's solution 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 $H_h = \{r'_1, r'_2, \ldots r'_{s(h)}\}$ 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 SMP's pairing of r with one of h''s slots, say h'_k. Let's also look at SMP'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 in RHP, r and h_j form an instability in SMP.

   Do they form an instability?

   Well, r prefers h to h' in RHP. The "cloning" we did to split hospitals into hospital-slots keeps all the slots of a hospital together so they all still go before the same hospitals they used to go before. Thus, r must prefer all slots of h to all slots of h' and so prefer h_j to h'_k.

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

   Thus, r and h_j do indeed constitute an instability.

   Why did we do all that again? Since the SMP solution is unstable if the RHP solution is unstable, that shows that the RHP solution is **stable** if the SMP solution is stable. We know the SMP solution **is** stable, which means the RHP one is as well!

8 Challenge Your Approach

1. **Carefully** run your algorithm on your instances above. (Don't skip steps or make assumptions; you're debugging!) Analyse its correctness and performance on these instances:

   SOLUTION: I'll leave this to you! (I cheated and jumped straight to a correct solution; so, this part is not as important for me.)

2. Design an instance that specifically challenges the correctness (or performance) of your algorithm:

   SOLUTION: I'll leave this to you for the same reasons!