CPSC 320: Reductions & Resident Matching *

The *Residents-Hospitals Problem* (RHP) is defined as follows: each medical student in a group needs a residency in some hospital. Each in a group of hospitals needs some number of residents (one or more), 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 suggestions (and give the resident a position at that hospital instead).

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
   ```

   And here is your second. Try to explore something a bit different with this one.

   SOLUTION: Let's make an instance that actually illustrates what's unique to the RHP. (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. Although we probably would not call it *trivial*, there's a special case where all hospitals have exactly one slot. What makes this an interesting special case?

   SOLUTION: Instances where each hospital has exactly one slot may as well be an SMP instance. That suggests a strong connection between these problems. It also suggests that the hard part for us is going to be figuring out what to do with hospitals that have multiple slots.

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. Let H be the set of hospitals, R be the set of residents, and $n = |R|$ be the number of residents. Note that $|H| \leq |R|$, but H may be much smaller. We need to know, for each hospital how many slots it has. We'll use $s(h)$ to denote the number of slots in hospital h. We also need preference lists, so $P[h]$ will be the preference list of hospital h (a permutation of R), and $P[r]$ will be the preferences list of resident r (a permutation of H).

2. Rewrite one of your small instances using these names.

   SOLUTION: Left to you.

3. Using your representational choices above, describe 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: 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 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 what makes a solution **valid** (it looks like a solution) and **good** (it satisfies the problem requirements):

   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. Write out one or more solutions to one of your small instances using these names.

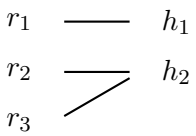
   SOLUTION: We'll work with this example:

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

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. 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 to this one in terms of its surface features ("story"), representation, or problem or solution structure:

SOLUTION: Obviously this is similar to SMP.

5 Brute Force?

Let us (quickly) try to figure out if a brute force solution might work.

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

   SOLUTION: n seems appropriate.

2. What can you say about the number of possible solutions, as a function of the instance size? Does it grow exponentially? Worse? (If you have time, or if it is helpful, sketch an algorithm to produce every valid solution. It will help to give a name to your algorithm and its parameters, especially if your algorithm is recursive.)

   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 $\frac{n!}{\prod_{h \in H}(s(h))!}$. Notice 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 very large (worse than $2^{\Theta(n)}$).

   And here's an informal 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 $R' = R - \{r\}$.
      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 $m \in$ AllSolns(H', R', s'), add $\{(h, r)\} \cup m$ to M.
   (e) Finally, return M

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

   SOLUTION:

   Not unless some hospital is taking almost everyone! Note that we haven't discussed how we would determine if a potential solution is indeed stable, but it doesn't really matter given how many potential solutions we would need to go through. Here is how we could do it, in case you are interested.

   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.

6 Promising Approach

We will use a *reduction* for our promising approach. We reduce **from** RHP **to** some other problem B. Describe an approach to solve RHP using a reduction, along with an algorithm that solves B.

1. Choose a problem B to reduce to.

   SOLUTION: Let's reduce to SMP.

2. Reduction part (i) example: Transform a small instance of RHP into an instance of B.

   SOLUTION: Here's our running example again:

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

   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:

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

   Now, each "half" of h_2 is its own "hospital". 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?)

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

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

3. Reduction part (ii) example: Transform a solution to your B instance into a solution to the RHP instance.

   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. Generalize: part (i): Design an algorithm to transform any instance I of RHP into an instance I' of B.

   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 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 have an SMP instance.

5. Generalize part (ii): 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.

7 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: Left to you.

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

   SOLUTION: Left to you.

8 Proof of Correctness

See if you can prove that your reduction produces a correct solution for any RHP instance. This may well involve showing that if S' is a good solution for the instance I' produced by part (i) of your reduction (where S' is the output of the black box algorithm for problem B in the diagram), then part (ii) of your reduction transforms S' into 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 $H_h = \{r'_1, r'_2, \ldots r'_{s(h)}\}$ 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 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.

Why did we do all that again? 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!