CPSC 320, Reductions & Resident Matching: A Residentectomy

September 8, 2018

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).

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.

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

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

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?

2 Represent the Problem

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

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

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

3 Represent the Solution

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

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

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

4. Draw at least one 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. You've probably already thought of it above!

5 Brute Force?

We have a way to test if something that looks like a solution but may have an instability is stable. (From the "Represent the Solution" step.) That is, given a "valid" solution, we can check whether it's "good".

1. 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.)

   This will be similar to the brute force algorithm for SMP (from the challenge problems).

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

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

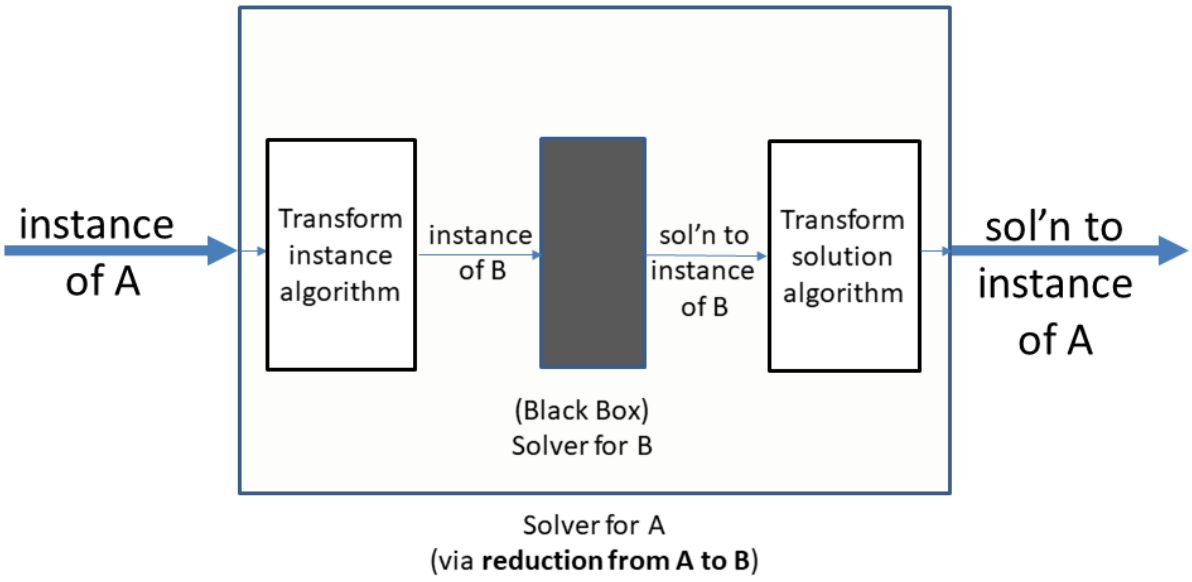
4. Exactly or asymptotically, how long will it take to test whether a solution form is valid and good with a naive approach? (Write out the naive algorithm if it's not simple!)

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

6 Promising Approach

We'll use a *reduction* for our promising approach. **Informal** reductions are the way we solve almost all problems. We practice more formal ones with the aim to master informal ones. To proceed, we need two **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 A to problem B is an algorithm that solves A by constructing an instance of B, solving that instance with an algorithm that solves B (without having to define it, i.e., just assuming B is solvable), and transforming the B solution into a solution to the A instance.[1] Here's a diagram of how that works, with the transformation of an A instance into a B instance and the transformation of a B solution into an A solution labelled as their own sub-algorithms:



It's often easiest to think about the reduction with these steps:

1. Change an instance of A into an instance of B by hand.
2. Solve the B instance by hand (but remember that if there are multiple good solutions, you're not guaranteed which your reduction receives).
3. Change the corresponding solution to B into a solution to A by hand.
4. Design an algorithm to change instances of A into instances of B.
5. Design an algorithm to change corresponding solutions to B into solutions to A.
6. Prove that if the solution to the B instance is correct, so too is the solution your algorithm creates for the original A instance. (**Or equivalently**, if the A solution generated is incorrect, then the B solution must have been incorrect as well.)

[1] This is actually a particular kind of reduction. In the more general but less common version, we can solve many instances of B.

Describe—in as much detail as you can—an approach to solve this problem using a reduction. We reduce **from** RHP **to** some other problem B.

1. Choose a problem B to reduce to.

2. Change a particular instance of RHP into an instance of B.

3. Change the solution to the B instance into a solution to the RHP instance.

4. Generalize: Design an algorithm to change any valid RHP instance into an instance of B.

5. Generalize: Design an algorithm to change the B instance's solution into the RHP instance's solution.

6. Prove that the solution you get to the RHP instance is guaranteed to be correct. (Depending on your chosen reduction, you likely have a stable solution to an instance of B and need to show that you get a correct solution to the RHP instance, i.e., that the solution is "valid" (has the right form) and "good" (is stable). So, you'll prove either *if B's solution is stable, RHP's solution is stable* or the contrapositive, if RHP's solution is *unstable*, then B's is *unstable* as well.)

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:

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

8 Repeat!

Hopefully, we've already bounced back and forth between these steps in today's worksheet! You usually *will* have to. Especially repeat the steps where you generate instances and challenge your approach(es).

9 Challenge Problems

1. **EASY:** Create a variant of RHP in which hospitals can have too much or too little capacity and reduce it to SMP. Establish or refute the quality of your reduction (i.e., prove or disprove that stable solutions to the underlying problem produce stable solutions to the general RHP).

2. **EASY:** Establish that reductions are transitive.

3. **MEDIUM:** Using the USMP-SMP reduction from the worked example and the G-S algorithm where **men** propose first: prove that no woman who ends up unmarried could be married in **any** stable match.

4. **MEDIUM-HARD:** Create a reduction from the "truncated" stable marriage to SMP. In truncated stable marriage, anyone can leave off as many people from the other category as they wish, all of whom are assumed to be worse than everyone they listed. Analyse the properties of your reduction.