COSC 320 Notes, Reductions & Resident Matching

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

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

2. What can you say about the number of valid 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, similar to the brute force algorithm for generating valid SMP solutions which is covered in the sample solutions to the first worksheet. It will help to give a name to your algorithm and its parameters, especially if your algorithm is recursive.)

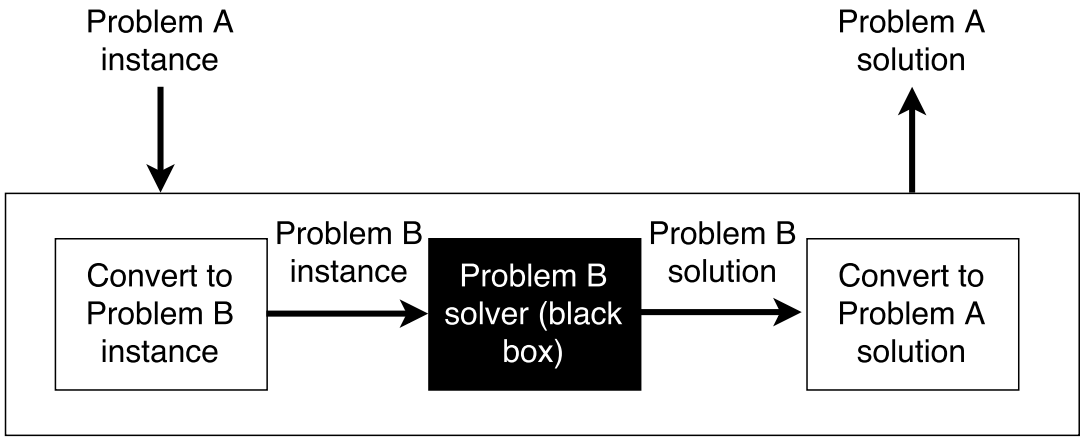
3. Exactly or asymptotically, how long will it take to test whether a solution form is valid and good with a naive approach? (Write out the naive algorithm if it's not simple!)

4. 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. Informally, a reduction is simply a way of solving a new problem by leveraging an algorithm that solves an already familiar problem. Here we describe reductions somewhat formally, so you know what you are doing when proceeding informally. We need two **definitions**:

- An *instance* of a problem is simply a valid 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 provides a way to solve problem A by using an algorithm that solves B. There are two key parts to a reduction: (i) an algorithm that transforms any instance, say I, of problem A to an instance, say I', of B, and (ii) an algorithm that transforms a solution for I' back to a solution for I. (When coming up with a reduction, you don't need to design the algorithm that solves B; we think of that algorithm as a "black box" because the reduction does not depend on its details.) [1] Here's a diagram of how the parts fit together:



Your job in defining a reduction is to describe how the two white boxes work. Here we will reduce **from** RHP **to** some other problem B.

1. Choose a problem B to reduce to.

[1] Reductions can be defined more generally, where part (i) constructs many instances of B.

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

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

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

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.

7 Proof of Correctness

Prove that your reduction produces a correct solution to the RHP instance. **Hint:** depending on your chosen reduction, you likely have a stable solution to an instance of B and need to prove that you get a correct (i.e., stable) solution to the RHP instance. You can either prove that *if B's solution is stable, RHP's solution is stable* or you can prove the contrapositive: *if RHP's solution is unstable, then B's must have been unstable as well.* (Another hint: proving the contrapositive is likely to be easier!)

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:

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

9 Repeat!

If your reduction does not seem to be working correctly, try again, hopefully with a bit more insight to guide you. Repeat until you have a convincing proof that your reduction works.