CPSC 320 Sample Solution, The Stable Matching Problem, Part I

1 Trivial and Small Instances

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

   SOLUTION: In SMP, it's tempting to say that the smallest possible instance has one employer and one applicant, and the only solution is to match them to each other.

   Can we could go smaller? Degenerate cases like "zero employers and zero applicants" are often helpful. Such an instance has exactly one solution, which is an empty matching.

2. Write down two **small** instances of SMP. One should have three employers and three applicants:

   SOLUTION: Here's what we might have come up with. This is just a sample of three employers and applicants and their preferences for each other.

```
e1: a2 a1 a3        a1: e3 e1 e2
e2: a1 a2 a3        a2: e3 e2 e1
e3: a2 a1 a3        a3: e2 e1 e3
```

   Each applicant lists their preferences for employers in order from most to least preferred. Similarly, each employer lists their preferences for applicants in the same order.

   The other can be even smaller, but not trivial:

   SOLUTION: We can still have a nontrivial example with two employers and two applicants:

```
e1: a1 a2           a1: e1 e2
e2: a1 a2           a2: e2 e1
```

   (With two employers and applicants, there are only two choices of preference list.)

2 Represent the Problem

1. What quantities or data (such as numbers, sets, lists, etc.) matter in representing an instance of SMP? Give them short, usable names.

   SOLUTION: You may have come up with more, fewer, or different quantities, but here are some useful ones.

   - n, the number of employers and the number of applicants.
   - E, the set of employers $\{e_1, e_2, \ldots, e_n\}$ (so, $n = |E|$)
   - A, the set of applicants $\{a_1, a_2, \ldots, a_n\}$ (again, $n = |A|$)
   - A preference list for each employer e_i, which we might call $P[e_i]$. This is a permutation of E. Note that employers have complete preferences for all the applicants, no ties. That's the simplest version of the problem; so, probably the one to start with.

- We could use P_E to denote the list of all employer's preference lists (so P_E is a list of lists).
- Similarly, a preference list for each applicant a_j, which we call $P[a_j]$, is a permutation of A.
- We could use P_A to denote the list of all applicant's preference lists (so P_A is also a list of lists).

2. For one of your trivial and one of your small instances, write down which quantities or data correspond with your names.

   For our first small instance we have $n = 3$, $E = \{e_1, e_2, e_3\}$, and so on.

3. Using your names, write down what a valid instance looks like, as input parameters to an algorithm or procedure that solves the problem:

   SOLUTION

   procedure SMP-ALGORITHM(n, P_E, P_A)
   ...
   end procedure

3 Represent the Solution

1. What are the quantities or data that matter in a solution to the problem? Give them names.

   SOLUTION: Central to our solution are matchings. We use e_i:a_j to indicate that employer e_i and applicant a_j are matched. A solution, then is a set M of matches (with some constraints that we describe next).

2. Using your names, describe what makes a solution **valid**.

   SOLUTION: We'll define a valid solution to be a perfect matching: a set of matches such that each applicant appears in exactly one match and each employer appears in exactly one match.

3. Describe what you think makes a solution **good**. Develop notation to help with your description.

   SOLUTION: A good solution should be valid, and in addition should be "self-enforcing" in the sense that no employer and applicant who aren't matched will decide to break the matching.

   With respect to a given matching M, an *instability* is an employer-applicant pair that would prefer to be matched with other than with their matches in M. That is, an instability is a pair $(e, a') \in E \times A$ such that if e:a is in M and e':a' is in M, then e prefers a' to a and a' prefers e to e'.

   We could use the notation $a' >_e a$ to mean "e prefers a' to a" and similarly $e' >_a e$ to mean "a prefers e' to e".

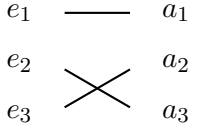
   A good solution contains no instabilities: we say that the solution is *stable*.

4. For one of your trivial and one of your small instances, write down one or more valid solutions.

   SOLUTION A valid solution to our instance of size three is the set $\{ e_1$:a_1, e_2:a_2, e_3:$a_3 \}$.

5. Draw at least one good solution for your instance of size three. (We would normally ask you to draw an instance as well, but SMP isn't an especially graphical problem.)

   SOLUTION A good solution to our instance of size three is the set $\{ e_1$:a_1, e_2:a_3, e_3:$a_2 \}$.



4 Similar Problems

As the course goes on, we'll have more and more problems we can compare against, but you've already learned some. So, give at least one problem that seems related to SMP in terms of its surface features ("story"), instance or solution structure, or representation:

SOLUTION: You've seen problems where you organize a bunch of values by comparisons among them: sorting. If you've worked with bipartite graphs and matching problems, anything associated with them seems promising, especially maximum matching. This also feels a bit like an election or auction, which takes us toward game theory. Maybe you'd even decide this feels a bit like hashing (mapping a value in one set to a different value in another set).

The point isn't to be "right" yet; it's to have a lot of potential tools on hand! As you collect more tools, you'll start to judge which are more promising and which less.

5 Brute Force?

You should usually start on any algorithmic problem by using "brute force": generate all possible **valid** solutions and test each one to see if it is, in fact, a **good** solution.

1. A valid SMP solution takes the form of a perfect matching: a pairing of each employer with exactly one applicant.

   It's more difficult than the usual brute force algorithm to produce all possible perfect matchings; instead, we'll count how many there are. Imagine that the applicants are arranged in some order. How many different ways we can arrange (permute) the employers next to them? This is the number of valid solutions.

   SOLUTION: There are n applicants we can line up with the first employer. Once we've chosen the first, there are $n-1$ to line up next to the second. Then, $n-2$ next to the third, and so on. Overall, then, that's $n \times n-1 \times n-2 \times \ldots \times 2 \times 1 = n!$. There are $n!$ perfect matchings, or "valid" solutions. That's already super-exponential, even if it takes only constant time per solution to produce them!

   We asked in the challenge problems for an algorithm to produce these. It's unusually challenging to design for a brute force algorithm, but it's useful to think about; so, we'll work through it here.

   Before we dive into an algorithm, let's just try creating all solutions for an example. We might start by just matching employer e_1 off to **some** applicant.

   e_1 —— a_1
   e_2 $\quad$ a_2
   e_3 $\quad$ a_3

   We can now set e_1 and a_1 aside, which gives us... another SMP instance that's smaller. As soon as you hear words like "and that leaves us with [something that looks like our original problem] but smaller", you should be thinking of recursion. Let's just assume we can recursively construct all possible solutions. That will give us back a bunch of sets of matches, in this case two:

   e_2 —— a_2 $\quad$ e_2 $\quad$ a_2
   e_3 —— a_3 and e_3 $\quad$ a_3

   We can add our set-aside pairing (e_1, a_1) onto each of these:

   e_1 —— a_1 $\quad$ e_1 —— a_1
   e_2 —— a_2 $\quad$ e_2 $\quad$ a_2
   e_3 —— a_3 and e_3 $\quad$ a_3

That's all the solutions with the match e_1:a_1. Who else can e_1 be matched with? Each of the other applicants. We can use the same procedure for each other possible match. Must e_1 match someone? Yes, because we need a perfect matching. So, that covers all the possibilities for e_1 and, recursively, for everyone else.

Now we're ready for an algorithm. Let's call it AllSolns. It's recursive; so, what's the base case? Our trivial cases are where $n = 0$ or $n = 1$. Let's try $n = 0$ as a base case. Looking back at the trivial cases, recall that the solution for $n = 0$ is the empty set of pairings $\{\}$. With that, let's build the algorithm. We'll use **return** when we produce the whole set at once and **yield** to produce one at a time. (You could just initialize a variable to the empty set and add in each **yielded** solution, returning the whole set at the end.)

```
procedure AllSolns(E, A)
// Return all perfect matchings of E with A
    if |E| = 0 then                                  ▷ The base case we chose.
        return {{}}                  ▷ The set of sol'ns, containing only the empty sol'n.
    else
        choose an e ∈ E                                          ▷ Any one, e.g., the first.
        for all a ∈ A do                                         ▷ Iterate through the men,
            for all M ∈ AllSolns(E − {e}, A − {a}) do            ▷ and the subproblem sol'ns.
                yield {(e, a)} ∪ M                           ▷ Add the set-aside match.
            end for
        end for
    end if
end procedure
```

If we use our analysis techniques to count the number of solutions this creates, the analysis will parallel the recursive function itself. In the base case when $n = 0$, AllSolns produces one solution. Otherwise, for each of the n applicants, it makes a recursive call with $n' = n - 1$ (one fewer employer and one fewer applicant in the subproblem). For each solution produced by that recursive call, it also generates one solution. If we give the number of solutions a name, we can express this as a recurrence:

$$N(n) = \begin{cases} 1 & \text{when } n = 0 \\ n * N(n-1) & \text{otherwise} \end{cases}$$

So, for example, $N(4) = 4*N(3) = 4*3*N(2) = 4*3*2*N(1) = 4*3*2*1*N(0) = 4*3*2*1*1 = 4!$. And, indeed, this is exactly the definition of factorial. So, $N(n) = n!$. There are $n!$ solutions to a problem of size n.

2. Once we have a valid solution, we must test whether it's a good solution.

   A valid solution, i.e., a perfect matching, is a good solution if it has no instabilities. Design a (brute force!) algorithm that—given an instance of SMP and a valid solution (perfect matching)—determines whether that perfect matching contains an instability. As always, it helps to **give a name** to your algorithm as well as its input parameters, *especially* if your algorithm is recursive. For brute force: generate each potential instability and then test whether it actually is an instability. Be careful: a potential instability is an (employer, applicant) pair who would rather be matched with each other than their actual matches, not an already-matched pair.

   SOLUTION: The form of a potential instability is a pair (employer and applicant). We therefore want to go through each pair of one employer and one applicant and check that (1) they are **not** already matched (or they cannot cause an instability) and (2) they'd rather be with each other than with their matches. (We'll assume that we have a quick way to find a match, which shouldn't be hard to create.) That should look like the following, where (n, P_E, P_A) is an instance of SMP and M is a solution to that instance (a perfect matching):

```
procedure IsStable(n, P_E, P_A, M)
// Is M stable for instance (n, P_E, P_A) of SMP?
    for all e ∈ E do
        for all a' ∈ A do
            if e:a' ∉ M then
                find a such that e:a ∈ M
                find e' such that e':a' ∈ M
                if e >_{a'} e' and a' >_e a then
                    return false
                end if
            end if
        end for
    end for
    return true
end procedure
```

3. Exactly or asymptotically, what is the running time of your brute force algorithm of part 2, that test whether a matching is stable? (What named quantity pertaining to an instance is most useful in expressing this?)

   SOLUTION: Let's assume that the basic steps of the algorithm, like comparing e's preferences for applicants a and a' and checking if e:a is in M, can be done in $\Theta(1)$ time. It's not immediately obvious how to do this, but with some careful use of data structures and $O(n^2)$ preprocessing, it's doable. For example, to determine which applicant is matched with a given employer, an array, say E-Match, indexed by employers is useful, where E-Match[e] stores the applicant matched with e. A separate array, A-Match, indexed by applicants, can similarly be used to determine which employer is matched with a given applicant in $\Theta(1)$ time. The text in Chapter 2 provides more information on useful data structures for this purpose.

   The number of iterations in the inner loop is independent of which iteration we're on in the outer one. The body takes constant time. Since both loops iterate at most n times, the overall time is $O(n^2)$. The worst case is when we find no instability; this takes $|E| * |A| * \Theta(1) = n * n * \Theta(1) = \Theta(n^2)$ time.

4. Brute force for solving SMP would generate each valid solution and then test whether it's good. Will brute force be efficient for this problem on instances of interest to us?

   SOLUTION: Looks like brute force will take $\Theta(n^2 n!)$ time in the worst case. That's horrendous. It won't do for even quite modest values of n.

6 Promising Approach

Describe—in as much detail as you can—an approach that looks more efficient than brute force.

SOLUTION: You may have lots of ideas. For example, you might have noticed that if a employer and a applicant both most-prefer each other, we must match them; that might form the kernel of some kind of algorithm. We won't go into ideas here. Rather, we refer you to the textbook's description of the Gale-Shapley algorithm.

7 Challenge Your Approach

1. **Carefully** run your algorithm on one or two of your instances above. (Don't skip steps or make assumptions; you're debugging!) Is your algorithm correct on these instances?

   SOLUTION: For fun, we'll use G-S with **applicants** considering employers.

G-S correctly terminates immediately on any $n = 0$ example with an empty set of matchings. With $n = 1$, the one applicant considers one employer, who accepts the match, and the algorithm correctly terminates.

Going back to our other two examples:

(a) Example #1:

```
a1: e2 e1 e3        e1: a3 a1 a2
a2: e1 e2 e3        e2: a3 a2 a1
a3: e2 e1 e3        e3: a2 a1 a3
```

G-S doesn't specify what order the applicants are chosen. We'll work from top to bottom:

i. a_1 considers e_2, who accepts. $M = \{ e_2{:}a_1 \}$
ii. a_2 considers e_1, who accepts. $M = \{ e_2{:}a_1, e_1{:}a_2 \}$
iii. a_3 considers e_2. e_2 rejects a_1 and accepts $a_{3\text{'s}}$ offer. $M = \{ e_2{:}a_3, e_1{:}a_2 \}$
iv. a_1 considers e_1 (2nd on its list). e_1 rejects a_2 and accepts $a_{1\text{'s}}$ offer. $M = \{ e_2{:}a_3, e_1{:}a_1 \}$
v. a_2 considers e_2, who prefers a_3 to a_2 and so rejects the offer. $M = \{ e_2{:}a_3, e_1{:}a_1 \}$
vi. a_2 considers e_3 (last on its list!), who accepts. $M = \{ e_2{:}a_3, e_1{:}a_1, e_3{:}a_2 \}$
vii. The algorithm terminates with the correct solution $M = \{ e_2{:}a_3, e_1{:}a_1, e_3{:}a_2 \}$.

(b) Example #2:

```
a1: e1 e2           e1: a1 a2
a2: e1 e2           e2: a2 a1
```

We'll again use G-S with applicants considering, working top to bottom.

i. a_1 considers e_1, who accepts. $E = \{ e_1{:}a_1 \}$
ii. a_2 considers e_1, who rejects (prefers a_1 to a_2). $M = \{ e_1{:}a_1 \}$
iii. a_2 considers e_2, who accepts. $M = \{ e_1{:}a_1, e_2{:}a_2 \}$
iv. The algorithm terminates with the correct solution $M = \{ e_1{:}a_1, e_2{:}a_2 \}$.

2. Design an instance of size three on which your algorithm is as slow as possible, or incorrect.

SOLUTION One idea is to have every employer have the same ranking of applicants. Without loss of generality, let's choose this ranking to be as follows.

```
e1: a1, a2, a3
e2: a1, a2, a3
e3: a1, a2, a3
```

The intuition is that every employer ei will make an offer to a1 when they have the first opportunity to make an offer (and a1 will choose the employer it ranks most highly).

Moreover, the two employers not matched with a1 will make an offer to their second choice, namely a2, but only one of them is matched to a2.

Finally, the employer that is matched with neither a1 nor a2 will make an offer to a3.

A total of six offers are made in this case: three to a1, two to a2, and one to a3. Interestingly, our argument that six offers are necessary doesn't depend on the applicant rankings! It also doesn't depend on the order in which employers make offers!

Can we do better? Yes. To do this, we need to force the employer that is ranked most highly by a1 to make more than one offer. Suppose for concreteness that applicant a1 ranks employer e3 most highly. To force seven offers, we change e3's ranking, and have to specify some information about applicant rankings:

```
e1: a1, a2, a3              a1: e3, ...
e2: a1, a2, a3              a2: e2, ...
e3: a2, a1, a3              a3: ...
```

To force seven offers, we can proceed as follows:

(a) e1 and e2 first make offers to a1 and one of them is matched with a1; suppose for concreteness that this is e2.

(b) e3 makes an offer to a2 and is matched with a2.

(c) e2 now makes an offer to a2; if we ensure that a2 prefers e2 to e3, a2 rejects e3 for e2.

(d) e3 makes an offer to a1, which is accepted

(e) e1 makes an offer to a2, which is rejected, and finally to a3.

In this case, two offers are made in each of steps (a) and (e), and one in each of steps (b), (c) and (d), for a total of seven offers.

Is that the best we can do? Could we get another employer to its worst choice? It turns out we actually **cannot**. See if you can prove it. Specifically: *if some employer considers the last applicant on its preference list, G-S must terminate on that iteration.*

3. Can you generalize from your example of part 3, to say what is the running time of your algorithm?

SOLUTION Our first example from part 2 suggests trying the following ranking for employers:

```
e1: a1, a2, ..., an
e2: a1, a2, ..., an

ei: a1, a2, ..., an
```

In this case, each e_i can first consider a_1. This takes n iterations of the While loop, and just one employer is matched to a1.

Then, each of the $n-1$ unmatched e_i's consider a_2. This takes $n-1$ iterations and results in one additional match, to a_2.

Continuing on, in the jth round, each of the $n-j+1$ unmatched e_i's consider a_j, with one being matched to a_j at the end. This takes $n-j+1$ iterations.

Finally, the last remaining unmatched employer considers, and is matched to, a_n.

The total number of iterations is $n+(n-1)+\ldots+2+1$, which is $n(n+1)/2$. Although this might not be the *worst* possible instance of size n, it is sufficient to show that the algorithm requires $\Omega(n^2)$ iterations of the While loop in the worst case.

We leave it as an exercise to see if you can generalize the second example of part 2 to get a bound that is (slightly) worse than $n(n+1)/2$.

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

These are just for fun, some are easier than others.

1. Design an algorithm to generate each possible perfect matching between n employers and n applicants. (As always, it will help tremendously to start by giving your algorithm and its parameters names! Your algorithm will almost certainly be recursive.)

2. A "local search" algorithm might pick a matching and then "repair" instabilities one at a time by matching the pair causing the instability and also matching their partners. Use the smallest possible instance to show how bad this algorithm can get.

3. Design a scalable SMP instance that forces the Gale-Shapley algorithm to take its maximum possible number of iterations. How many is that? (A "scalable instance" is really an algorithm that takes a size and produces an instance of that size, just like the "input" in worst case analysis is scalable to any n.)