

CPSC 320 Sample Solution, The Stable Matching Problem

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: If you think of the smallest possible instances, it usually guides you towards trivial instances. In SMP, it's tempting to say that the smallest possible instance has one employer and one applicant, but we can go smaller! Degenerate cases like "zero employers and zero applicants" are often helpful!

So, is zero employers and zero applicants trivial? Sure! There's exactly one solution, in which no one is matched with anyone else.

What about one employer and one applicant? Regardless of their preferences (which, in fact, must be simply for each other), the only solution is for the one employer to be matched with the one applicant.

So, zero employers/applicants and one employer/applicant are the trivial instances.

FROM SOLUTION REPRESENTATION: The solution for a problem with $n = 0$ is the empty set of pairings $\{\}$. The solution for a problem with $n = 1$ is $\{(e_1, a_1)\}$.

What about two employers and applicants? You might start that here, but you'll quickly realize it belongs in the next slot...

2. Write down two **small** instances of SMP. One should use the preference lists your group gave for the candy/cookie example above:

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 that could come from the chocolate example:

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

Each applicant lists their preferences for employers in order from most to least preferred.

Each employer lists their preferences for applicants in the same order.

FROM PROBLEM REPRESENTATION: We can rephrase this with our new notation, but honestly, there's not much to do. $n = 3$, clearly. $A = \{a_1, a_2, a_3\}$, but all that changes there is subscripts vs. numbers on the side. $P[a_1]$ is the first list on the left. Similarly, we can see the employers and the other preference lists.

FROM SOLUTION REPRESENTATION: There happens to be only one stable solution to this instance: $\{(e_2, a_3), (e_1, a_1), (e_3, a_2)\}$.

On to the next question...

The other can be even smaller, but not trivial:

SOLUTION: We can go smaller and still have a trivial example. So, let's do so. Two employers and two applicants:

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

With two employers/applicants, there are only two choices of preference list. So, I gave the applicants matching preference lists and the employers opposite preference lists, just to illustrate both possibilities. That may be useful or may not!

FROM PROBLEM REPRESENTATION: I won't explicitly use our new names here, since little has changed, as noted above.

FROM SOLUTION REPRESENTATION: Again, there happens to be only one stable solution to this instance: $\{(e_1, a_1), (e_2, a_2)\}$. (Want one with more than one solution? Try tweaking a_1 's preference list.)

2 Represent the Problem

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

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

- n , the number of employers and the number of applicants.
- E , the set of employers $\{e_1, e_2, \dots, e_n\}$ (so, $n = |E|$)
- A , the set of applicants $\{a_1, a_2, \dots, a_n\}$ (again, $n = |A|$)
- Each employer's preference list—which we might call $P[e_i]$ for employer i —is a permutation of A , the set of applicants. Note that I'm forcing the employers to have complete preferences for all the applicants, no ties. That's the simplest version of the problem; so, probably the one to start with. I'll also assume that everyone prefers being matched to not being matched.
- Similarly, each applicant's preference list, $P[a_j]$, is a permutation of E .
- It's good to have a notation to indicate whether a applicant prefers one employer to another (or similarly for a employer). I'll use $e_i >_{a_j} e_k$ to mean that a_j prefers e_i to e_k , i.e., e_i occurs earlier in a_j 's preference list than e_k .

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

SOLUTION: See above.

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

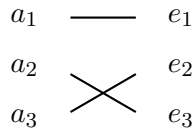
SOLUTION: This isn't a problem that suggests a lot of obvious graphical solutions, but I like drawing this as two columns with the preferences on the outside. That leaves space in the middle for us to draw lines in between employers and applicants:

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

You might also create an $n \times n$ grid, with applicants across the top and employers down the side. That would let you put information about each potential couple in each grid cell.

Different graphical representations will suggest different information in the problem to focus on or ignore or facilitate particular ways of thinking about the problem. We'll keep pushing on this as the course proceeds!

FROM SOLUTION REPRESENTATION: Abandoning plain text, let's actually draw this:



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

SOLUTION: What "shape" is an instance (in programming terms, what inputs of what type constitute an input) and what additional constraints are there on inputs of that "shape" for them to be valid?

The crucial piece of an instance is the preference lists for the employers and applicants. We need to know how many of those there are.

So, we might describe an instance as a tuple: (n, P_A, P_E) , where n is the number of applicants (and also the number of employers), P_A is a list of n preference lists for the applicants (where element i is a_i 's preferences), and P_E is a list of n preference lists for the employers.

If you're more comfortable thinking in programming input/output terms, you might say that the input is: one line with a (non-negative) integer n , then n lines representing the applicants's preference lists each with n whitespace-separated numbers forming a permutation of $1, \dots, n$, and finally n similar lines representing the employers' preference lists.

3 Represent the Solution

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

SOLUTION: Central to our solution are matchings, which are pairs (e_i, a_j) indicating that employer i and applicant j are matched. A solution, then is a set of pairings (with some constraints we describe next).

2. Describe using these quantities what a **valid** solution looks like:

SOLUTION: There's no technical weight here for the word "valid." But broadly speaking, we consider a solution invalid if it violates a constraint. In this case, we'll define validity as measuring whether we've successfully matched everyone just once (which was the constraint we were given for the problem). A valid solution is a perfect matching: a set of pairings such that each applicant appears in exactly one pairing and each employer appears in exactly one as well.

3. We haven't said what a **good** solution to this problem looks like. Brainstorm some different possible definitions of a good solution.

SOLUTION: The word "good" is even less technically meaningful than the word "valid": it just defines a solution that's high-quality for some reason, and it's up to us to define some way to judge that (ideally, one that makes sense and that will make the problem easy to solve efficiently).

We will define a good solution as a stable solution (defined below), but here are some alternative ways we might define a good solution:

- One that has as many people with their first choice partner as possible
- One that has as few people with their last choice partner as possible
- One that maximizes a total "happiness" score, based on some way to compute happiness based on preference lists.

And so on. You may have come up with different alternatives!

-
- There are many reasonable ways we could define a **good** solution to this problem. However, for the remainder of this worksheet we will use **one particular** definition. **WAIT HERE** for us to provide that definition!

SOLUTION: We choose to define a good solution as being “self-enforcing” in the sense that no employer and applicant who aren’t matched will decide to break the arrangement we suggested. So, a good solution is *stable* in that it contains no *instabilities*.

Next, what’s an instability? An instability can occur for a employer e_i and applicant a_j who are not matched in the solution ($(e_i, a_j) \notin$ the set of pairings). Let a' be e_i ’s partner in the solution and e' be a_j ’s partner. Then, e_i and a_j constitute an instability if $e_i >_{a_j} e'$ and $a_j >_{e_i} a'$. That is, each of e_i and a_j prefers the other to their assigned partners.

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

SOLUTION: See above.

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

SOLUTION: Again, see above.

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

SOLUTION: You may not have enough background in problems to feel like you’ve seen a lot of similar problems, but you have at least 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. (We often discuss "goodness" measures that give more points to a first preference than a second and so forth, like the Borda count. You could frame that problem as a maximum matching problem!) 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": the most straightforward, non-optimized approach you can think of. In this case (and in my cases), we want to generate all possible solutions and test each one to see if it is, in fact, **the** solution we’re looking for.

- A possible SMP solution takes the form of a perfect matching: a pairing of each applicant with exactly one employer. We’ll call a perfect matching a "valid" (but not necessarily good) solution.

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 lining all the employers up in a row in a particular order. How many different ways we can line up (permute) the applicants next to them?

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 $n \times n - 1 \times n - 2 \times \dots \times 2 \times 1 = n!$. There are $n!$ perfect matchings, our "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 the first person (say, a_1) off to **someone**. After all, we know she needs to be matched to somebody!

a_1 ——— e_1
 a_2 e_2
 a_3 e_3

We can now set a_1 and e_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 pairings, in this case two:

a_2 ——— e_2 a_2  e_2
 a_3 ——— e_3 and a_3  e_3

a_1 ——— e_1 a_1  e_1
 a_2 ——— e_2 a_2  e_2

We can add our set-aside pairing (e_1, a_1) onto each of these: a_3 e_3 and a_3 e_3

That's all the solutions in which a_1 weds e_1 . Who else can a_1 be matched with? Each of the other employers. We can use the same procedure for each other possible pairing. Must a_1 be matched with someone? Yes, because we need a perfect matching. So, that covers all the possibilities for a_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, I see the solution for $n = 0$ is the empty set of pairings $\{\}$. With that, let's build the algorithm. I'll use **return** when I'm producing 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( $a, M$ )
  if  $|W| = 0$  then                                ▷ The base case we chose.
    return  $\{\{\}\}$                                     ▷ The set of sol'ns, containing only the empty sol'n.
  else
    choose a  $w \in a$                                 ▷ Any one, e.g., the first.
    for all  $m \in M$  do                                ▷ Iterate through the employers,
      for all  $S \in \text{ALLSOLNS}(W - \{w\}, M - \{m\})$  do  ▷ and the subproblem sol'ns.
        yield  $\{(m, w)\} \cup S$                         ▷ Add the set-aside pairing.
      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 employers, 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 possible solution, we must test whether it's the solution we're looking for. Informally, we'll refer to this as asking whether it's a "good" solution.

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 perfect matching—determines whether that perfect matching contains an instability. (As always, it helps to **give a name** to your algorithm and its parameters, *especially* if your algorithm is recursive. Remember, for brute force: generate each possible solution (possible instability, in this case) and then test whether it really is a solution.)

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 their partners. (I'll assume we have a quick way to find a partner, which shouldn't be hard to create.) That should look like the following, where (n, P_W, P_M) is an instance of SMP and S is a solution to that instance (a perfect match and therefore a set of pairings (e_i, a_j)):

```

procedure ISSTABLE( $(n, P_W, P_M), S$ )
  for all  $w \in \{a_1, \dots, a_n\}$  do
    for all  $m \in \{e_1, \dots, e_n\}$  do
      if  $(m, w) \notin S$  then
        find  $m'$  such that  $(m', w) \in S$ 
        find  $w'$  such that  $(m, w') \in S$ 
        if  $m >_w m'$  and  $w >_m w'$  then
          return false
        end if
      end if
    end for
  end for
  return true
end procedure

```

3. Exactly or asymptotically, how long does your algorithm take? (Again, you should explicitly name the size of an instance and perform your analysis in terms of that name!)

SOLUTION: Let's assume we do an efficient (constant-time) job of operations like comparing a 's preferences for the two employers and checking if (m, w) is in S . (It's not immediately obvious how to do this, but with some careful data structures and $O(n^2)$ preprocessing, it's doable!) 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. So, in the worst case (when we find no instability), this takes $|M| * |W| * O(1) = n * n * O(1) = O(n^2)$ time.

4. Brute force would generate each valid solution and then test whether it's good. Will brute force be sufficient for this problem for the domains we're interested in?

SOLUTION: Looks like brute force will take $O(n^2n!)$ time. That's horrendous. It won't do for even quite modest values of n . (But, it is good enough to solve the $n = 3$ example we demonstrated in the classroom.)

6 Lower-Bound (Extra)

SOLUTION: We didn't discuss this in class, but you can often lower-bound the runtime of any algorithm to solve a problem by determining how long it would take simply to read the input to the problem.

By lower-bounding the problem, we have a "goal" to shoot for in finding an efficient algorithm. If we upper-bound the worst-case runtime of some algorithm to be the same as the lower-bound on the problem, then we know that we have an asymptotically optimal algorithm.

Looking back at our most useful instance description (the one that talks about "whitespace-separated" preference list lines), we can see that we'll have one number (n), followed by n lines each with n numbers, followed by another n lines of n numbers each. That's $n + n^2 + n^2 \in \Omega(n^2)$.

(Our analysis also gives an O bound, but it's the Ω bound we care about, since the purpose is to lower-bound runtime of solutions.)

So any algorithm that even reads the input will take $\Omega(n^2)$ time.

7 Promising Approach

Unless brute force is good enough, describe—in as much detail as you can—an approach that looks promising.

SOLUTION: You may have thought of lots of ideas. (E.g., earlier I sketched a Borda count-based approach that uses maximum matching. We've noticed that if a employer and a applicant both most-prefer each other, we must pair them; that might form the kernel of some kind of algorithm. Etc.)

I won't go into one here. Instead, I refer you to the textbook's description of the rather awesome and Nobel prize-winning Gale-Shapley algorithm.

Keen note about Gale-Shapley: It runs in $O(n^2)$ time. That means it matches our lower bound on the problem's runtime and so is asymptotically optimal!

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: For fun, we'll use G-S with **applicants** applying.

G-S correctly terminates immediately on any $n = 0$ example with an empty set of matches. With $n = 1$, the one applicant applies to the one employer, who must accept, and the algorithm correctly terminates with them matched.

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 apply. We'll work from top to bottom:

- i. a_1 applies to e_2 , who accepts. $E = \{(e_2, a_1)\}$
- ii. a_2 applies to e_1 , who accepts. $E = \{(e_2, a_1), (e_1, a_2)\}$

- iii. a_3 applies to e_2 , who prefers a_3 to a_1 . e_2 breaks their pairing with a_1 and accepts a_3 's application. $E = \{(e_2, a_3), (e_1, a_2)\}$
- iv. a_1 applies to e_1 (2nd on their list), who prefers a_1 to a_2 . e_1 breaks their pairing with a_2 and accepts a_1 's application. $E = \{(e_2, a_3), (e_1, a_1)\}$
- v. a_2 applies to e_2 , who prefers a_3 to a_2 and so declines the application. $E = \{(e_2, a_3), (e_1, a_1)\}$
- vi. a_2 applies to e_3 (last on their list!), who accepts. $E = \{(e_2, a_3), (e_1, a_1), (e_3, a_2)\}$
- vii. The algorithm terminates with the correct solution $S = \{(e_2, a_3), (e_1, a_1), (e_3, a_2)\}$.

(b) Example #2:

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

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

- i. a_1 applies to e_1 , who accepts. $E = \{(e_1, a_1)\}$
 - ii. a_2 applies to e_1 , who declines (prefers a_1 to a_2). $E = \{(e_1, a_1)\}$
 - iii. a_2 applies to e_2 , who accepts. $E = \{(e_1, a_1), (e_2, a_2)\}$
 - iv. The algorithm terminates with the correct solution $S = \{(e_1, a_1), (e_2, a_2)\}$.
2. Design an instance that specifically challenges the correctness (or performance) of your algorithm:

SOLUTION: Skipping this, since we've already seen a proof of correctness for G-S!

9 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).

SOLUTION: We bounced back and forth quite a bit, even in this carefully crafted solution.