Stable Matching: an introduction

Department of Computer Science
University of British-Columbia

1 Preamble

The problem we will consider in worksheet 1 is the *Stable Matching* problem. While this problem's solution does not fit in any of the broad categories of problem solving techniques we will discuss this term, it is a problem with many applications and moreover our discussions will lead naturally into the first technique we will talk about in detail: *reductions*.

Before getting into definitions and algorithms, let us first provide the historic context about the stable matching problem. This problem was first studied formally by two American computer scientists: David Gale and Lloyd Shapley in the early 1960s [1]. An algorithm similar to the one they describe had been used for the previous decade in matching medical interns to hospitals. Unfortunately, the setting Gale and Shapley used to describe the problem (involving men, women, and marriage proposals) has not aged well given the current cultural context. The problem can be described just as well in a non-gendered setting, which is what we do in this document. Our textbook [3] uses the original problem setting; while you are welcome to read Section 1.1 as a supplement to this document, it is not required. We strongly believe this textbook is by far the best choice for the course apart from the issue with Section 1.1, which is why we decided to replace this section by the current document instead of replacing the whole textbook.

This introduction was written by Patrice Belleville in 2022, and has been lightly updated for the 2023W2 course offering.

2 Problem Definition

Consider the problem of matching co-op students to employers. We simplify the problem by making the following assumptions, some of which we will relax later in the worksheets, assignments and exams:

- We have n employers $E = en$ each of which has exactly *one* position available, and n co-op students $S = sn$.
- Each employer has a *preference list* of co-op students: this list contains all co-op students, sorted by the employer's preference. That is, the employer likes the first student on the list best, the second student on the list is the second-best liked, etc. Equalities are not allowed in the preference list: given two students s_i and s_j, the employer **must** prefer one to the other.
- Each student has a *preference list* of employers: this list contains all employers, sorted by the student's preference. That is, the student likes the first employer on the list best, the second employer on the list is the second-best liked, etc. Equalities are not allowed in the preference list: given two employers e_i and e_j, the student **must** prefer one to the other.

Our goal is to match each student to exactly one employer. Now, we could do this arbitrarily, but if we did we might end up with the following situation:

- Employer e_a is matched with student s_b.
- Employer e_c is matched with student s_d.

- Employer e_a likes student s_d better than student s_b.
- Student s_d likes employer e_a better than employer e_c.

What do you think would happen in this situation? Well, as you can guess, employer e_a would terminate s_b's employment and offer a job to s_d, who would then drop e_c and accept s_a's offer enthusiastically. This actually happened with the medical intern market, with consequences like offers being called in to an applicant at midnight with a few hours' deadline to respond [2]! This situation is called an *instability* and we would like to avoid it as much as possible. So our (revised) goal is to match each student to exactly one employer while avoiding instabilities.

Formally, the input to our problem will be:

- A preference list $P[e_i]$ for each employer e_i that is a permutation of the list of students, with each student's position in the list indicating how much/how little employer e_i likes that student.
- A preference list $P[s_j]$ for each student s_j that is a permutation of the list of employers, with each employer's position in the list indicating how much/how little student s_j likes that employer.

The desired output is a list of pairs (e_i, s_j) where each employer and each student appear exactly once in the list, and there are no instabilities. Such a matching is called *stable*.

3 The Gale-Shapley algorithm

It is not immediately obvious that it is always possible to match students and employers without any instability. In fact, there exist instances of variations of this problem where every possible solution is unstable. One such example is the *roommate assignment* problem: we have n people to pair up based on their preferences, with each pair sharing a house. For some collection of preference lists, *every* pairing will have at least one instability.

Here is one possible algorithm we might think of using to solve the stable matching problem (not the roommate assignment problem mentioned in the previous paragraph):

- Start by pairing students and employers randomly.
- While you can find two pairs (e_a, s_b), (e_c, s_d) that form an instability, swap the elements of the two pairs. That is, replace these pairs by the pairs (e_a, s_d), (e_c, s_b).

Unfortunately, it can be shown that for some instances of the problem, this algorithm may not terminate. That is, it may enter an infinite loop where it keeps cycling between the same small subset of unstable solutions.

Gale and Shapley proposed the following slightly more complicated, but correct, algorithm. We call a co-op student *free* if they currently do not have a job. Similarly an employer is called *feee* if they do not currently have a co-op student hired to occupy its advertized job.

```
1  set all s ∈ S and e ∈ E to free
2  while some free employer e hasn't made an offer to every student do
3      s ← the highest-ranking student e hasn't made an offer to
4      if s is free then
5          hire(e, s)
6      else
7          e′ ← s's current employer
```

```
        if s prefers e to e' then
            set e' to free
            hire(e, s)
        endif
    endif
endwhile
return the set of pairs
```

Don't worry if you can't see why this algorithm is correct: it's not obvious! Also, while the algorithm is phrased in terms of offers and hires, the process happens without any actual interaction between students and employers once their preferences have been collected. It terminates with a single, finished matching showing which employer hires each student. We will prove its correctness in Section 5. For now, let us look at an example. Suppose we have three employers e_1, e_2, e_3 and three students s_1, s_2, s_3, with the following preference lists:

$$\begin{aligned} e_1 &: \ s_3, s_1, s_2 \\ e_2 &: \ s_1, s_3, s_2 \\ e_3 &: \ s_3, s_2, s_1 \end{aligned}$$

$$\begin{aligned} s_1 &: \ e_1, e_3, e_2 \\ s_2 &: \ e_1, e_3, e_2 \\ s_3 &: \ e_2, e_3, e_1 \end{aligned}$$

The algorithm might proceed as follows, although the details depend somewhat on which employer makes the next offer each time through the loop. The pairings are drawn on the left of each figure, while the remaining choices for each employer are indicated on the right.

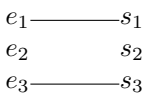
1. Initially, all employers and students are free.

2. e_1 makes an offer to s_3, who accepts:

3. e_2 makes an offer to s_1, who accepts:

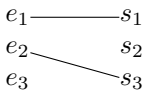
4. e_3 makes an offer to s_3, who drops e_1 in order to accept:

5. e_1 makes an offer to s_1, who drops e_2 in order to accept:



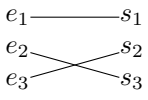
e_1 : ~~s_3~~, ~~s_1~~, s_2
e_2 : ~~s_1~~, s_3, s_2
e_3 : ~~s_3~~, s_2, s_1

6. e_2 makes an offer to s_3, who drops e_3 in order to accept:



e_1 : ~~s_3~~, ~~s_1~~, s_2
e_2 : ~~s_1~~, ~~s_3~~, s_2
e_3 : ~~s_3~~, s_2, s_1

7. e_3 makes an offer to s_2 who accepts:



e_1 : ~~s_3~~, ~~s_1~~, s_2
e_2 : ~~s_1~~, ~~s_3~~, s_2
e_3 : ~~s_3~~, ~~s_2~~, s_1

At this point every employer's job is occupied by a co-op student, and so the algorithm terminates and returns the pairs $\{(1,1),(2,3),(3,2)\}$.

4 Time Complexity

We now want to analyze the time complexity of the Gale-Shapley algorithm. At first glance, it is not even clear that this algorithm terminates. For instance, step 3 in our example doesn't appear to change much: the algorithm had constructed two pairs by the end of step 2, and it still has two pairs (albeit not the same ones) at the end of step 3.

In order to prove that the algorithm terminates, we need to define a *measure of progress*: some quantity that will increase (or decrease) at every step of the algorithm, and whose maximum (or minimum) value can also be determined. Measures of progress are often extremely useful when analyzing algorithms' time complexity, particularly with complex `while` loops (as opposed to simple counting or 'for-each' loops). In this specific case, our definition will be based on the following two simple observations:

Observation 4.1. *Once a student s has received a first offer from an employer, s will remain hired until the end of the execution of the algorithm. Each employer that s accepts an offer from during the execution of the algorithm is ranked higher on s's preference list than the previous ones.*

This observation holds because students are never "fired" by the algorithm: the only time a student stops being matched to an employer is when that student decided to accept an offer from another employer they like better. We can not make a similar observation about employers. As you saw in the example from the previous section, it is possible for an employer who was matched to a co-op student to find itself free again after that student accepted a better offer from a different employer. We can nonetheless say something about employers:

Observation 4.2. *The sequence of students to which an employer e makes a job offer gets worse and worse as the algorithm progresses.*

Based on the latter observation, we can bound the total number of iterations of the `while` loop in the algorithm. The measure of progress, in this case, is the number of different offers that were made during the execution of the algorithm.

Theorem 4.1. *The Gale-Shapley algorithm terminates after at most n^2 iterations of the* `while` *loop.*

Proof. At every iteration, an employer makes a job offer to a co-op student this employer has never made an offer to. Each employer can make offers to at most n students in total, which bounds the total number of offers (and hence the total number of iterations of the `while` loop) by $n \times n = n^2$. ■

Does Theorem 4.1 mean the worst-case running time of algorithm Gale-Shapley is in $O(n^2)$? Unfortunately not, or at least not immediately. In order to state this, we would need to be able to show how to implement the body of the loop so each iteration runs in constant time. Let us consider each statement in the loop body and figure out how to implement it efficiently:

1. **Finding a free employer that hasn't made an offer to every student**: we store all free employers in a data structure with constant-time insert and remove operations (for instance a stack), removing an arbitrary employer from that data structure when we need to find a free employer, and inserting employers that have been rejected (either because their offer is not accepted, or on line 9) back into the data structure.

2. **Finding the highest ranking student e hasn't made an offer to**: we keep the index of the next co-op student to make an offer to along with e's preference list.

3. **Determining if s is free**: we maintain, for each co-op student, their current employer with a special value to indicate the student is free.

4. **Determining s's current employer**: see above.

5. **Deciding if s prefers e to e'**: this is the trickiest operation to implement efficiently. We do not want to search s's preference list to find the positions of both e and e', because this would take $\Omega(n)$ time in the worst-case at every iteration of the loop. Instead, we want to look these positions up in constant time. This can be done by pre-computing them all *before* we enter the loop.

   Intuitively, what we will do is go through each one of the students' preference lists, and whenever we see an employer we will record its list position in another array. This can be implemented as follows: for each student s_j,

   ```
   for i ← 1 to n do
       e ← P[s_j][i]    // Employer e is at position i
       Epos[s_j][e] = i
   ```

   This loop runs in $\Theta(n)$ time and needs to be performed n times (once per student), adding up to a total computation time in $\Theta(n^2)$. **However** this only needs to be done once, before the loop, and after that line 8 can be implemented using only a pair of constant time table lookups.

The total running time of our algorithm is thus n^2 (for the pre-computations of employer positions on students' preference lists) plus at most n^2 constant time loop iterations.

Theorem 4.2. *The Gale-Shapley algorithm can be implemented to run in $\Theta(n^2)$ time in the worst case.*

5 Correctness

The only thing that remains to be proved is that the algorithm always returns a list of pairs (e_i, s_j) where each employer and each student appear exactly once in the list, and there are no instabilities. Let us start by proving the first part of this sentence.

Lemma 5.1. *If an employer e is free, then there is a co-op student that employer has not yet made an offer to.*

Proof. We use a proof by contrapositive. That is, we show that if e has made an offer to every co-op student, then e is not free.

If e has already made offers to every student, then every student has already received at least one offer. By observation 4.1, this means that every one of the n students has been hired. Because an employer can only hire one student at any given time, this means at least n employers have currently hired a student. Because there are only n employers in total, this means e is one of those employers, and hence e is not free. ■ ■

Lemma 5.2. *Each employer and each student appear exactly once in the list returned by the algorithm.*

Proof. By Lemma 5.1, every employer has hired exactly one student. So the list contains n pairs and each employer appears exactly once. A student can only be hired by at most one employer at a time, and there are n students, which means every student will appear in the list, and be matched to a single employer. ■ ■

So the algorithm returns a list of pairs, where each employer is matched to exactly one student and the other way around. That is, the algorithm returns what is known as a *perfect matching*. But what about instabilities?

Theorem 5.1. *The matching return by the algorithm is stable.*

Proof. We use a proof by contradiction. Suppose the matching returned by the algorithm contains an instability: employers e_a, e_c, and students s_b, s_d such that

1. Employer e_a is matched with student s_b.
2. Employer e_c is matched with student s_d.
3. Employer e_a likes student s_d better than student s_b.
4. Student s_d likes employer e_a better than employer e_c.

Because e_a is paired with s_b in the matching, s_b must be the last student that e_a made an offer to. Now, did e_a make an offer to s_d?

If e_a did not make an offer to s_d : s_b must be ranked higher in e_a's preference list than s_d, because e_a makes offers in decreasing order of preference. But this contradicts statement 3.

If e_a made an offer to s_d : s_d must have rejected e_a in favour of an employer they like better than s_a. By Observation 4.1, this means s_d likes e_c better than e_a. This time, there is a contradiction with statement 4.

So in both cases we reach a contradiction. Therefore no instability exists in the matching returned by the Gale-Shapley algorithm. ■ ■

The Gale-Shapley algorithm returns one possible stable matching. There are instances of the stable matching problem where many different stable matchings exist. It is possible to prove the following result, which we include for interest but whose proof we will omit (interested readers can find it in both of the references listed at the end of this document).

Theorem 5.2. *In the matching returned by the algorithm presented in Section 3,*

- *Employers get the **best** possible co-op student they could get in any of the stable matchings that exists for this instance of the problem.*
- *Students get the **worst** possible employer they could get in any of the stable matchings that exists for this instance of the problem.*

If you feel offended by this result, you can turn the algorithm around by having students make offers to employers instead. Then student will get the best possible employer out of all of the existing stable matchings, whereas employers will get the worst possible co-op student out of every existing stable matching.

6 Exercises

Here are a few small problems you can think about to deepen your understanding of the problem. We make no claims to originality: some of them can be found in the textbook, others not. All of them point out what I think are interesting aspects of the problem.

1. Give an example with n employers and n students where, using the algorithm presented in this document, **every** student will get the worst possible employer (the last one on their preference list).
2. Give an example with n employers and n students where an employer may end up paired with the worst possible co-op student (the last one on their preference list).
3. Prove that **at most one** employer may end up paired with the worst possible co-op student (the last one on their preference list).
4. Give an example with n employers and n students where n is even and there are at least $2^{n/2}$ different stable matchings.
5. Give an example where the following algorithm mentioned in Section 3
   - Start by pairing students and employers randomly.
   - While you can find two pairs (e_a, s_b), (e_c, s_d) that form an instability, swap the elements of the two pairs. That is, replace these pairs by the pairs (e_a, s_d), (e_c, s_b).

   can end up looping infinitely. Hint: use $n = 3$.
6. Give an example of the roommate assignment problem where every solution has an instability. Hint: four people are sufficient.

References

[1] David Gale and Lloyd Shapley. College admissions and the stability of mariage. *American Mathematical Monthly*, 9(1):9–14, 1962.

[2] D. Gusfield and R. Irving. *The Stable Mariage problem: structure and algorithms*. MIT Press, 1989.

[3] J. Kleinberg and É. Tardos. *Algorithm Design*. Addison-Wesley, 2006.