Warm Up: The Stable Matching Problem

1 The problem set-up

The origins of the problem go back to 1962. Two experts in mathematical economy, David Gale and Lloyd Shapley, posed the following question:

Is it possible to design a job recruiting process that is self-enforcing?

What does the question mean? Imagine the following real life situation. Software engineering graduates from New Zealand universities are applying for jobs. There are several companies in the market that are looking for SE graduates. In this process each of the students (1) selects several companies and (2) ranks the companies in a preference order. Similarly, each of the companies (1) collects all the applications and (2) ranks the applications in order.

Suppose a company c_1 makes an offer to an applicant a_1, and a_1 accepts the offer. A few days later another company c_2 makes an offer to a_1; and imagine that a_1 prefers to work for c_2. So, a_1 retracts from the first offer and accepts the offer from c_2. Company c_1, knowing that a_1 retracted from the offer, gives its offer to another applicant. These might cascade into a chaos in the job market.

It might even be worse. Imagine an applicant a_2 is a good friend of a_1, a_2 has an offer from a company c_3, and a_2 knows that a_1 went for company c_2. So, the applicant a_2 calls up to c_2, tells a nice story that the applicant a_2 prefers to work for c_2 than for c_3. It might be the case that c_2 realises that a_1 and a_2 form a good team, and makes an offer to a_2 as well by retracting from its previous offer to an applicant a_3. Now, a_3 is without offer!

The cause of the chaos above is self-interests of the applicants and the companies. So, the term "self-enforcing" above refers to a mechanism that prevents the situations as above happening. In other words, the mechanism should be such that self-interests of both parties (applicants and the companies) should prevent the chaos in the job market.

What does this mean for applicants a and companies c given their preferences? The answer is this. The mechanism should assign applicants to companies such that for every company c and every applicant a that is not assigned to c, at least one of the following must be true:

1. Company c prefers all of its accepted applicants to a; or
2. Applicant a prefers her current offer to the offer from the company c.

The first condition says that company c has no reason to make a job offer to the applicant a; and the second condition says that the applicant a has no reason to move to company c.

2 Formalisation of the problem

We want to formulate the problem as clean as possible. So, assume that we have two sets **C** and **A** of companies and applicants, respectively. One possible cleaning of the problem is this:

1. $\mathbf{C} = \{c_1, c_2, \ldots, c_n\}$ and $\mathbf{A} = \{a_1, a_2, \ldots, a_n\}$. So, the number of companies equals the number of applicants.
2. Each applicant $a \in \mathbf{A}$ applies to *all* companies and ranks them in a's preference order.
3. Each company $c \in \mathbf{C}$ wants to accept a *single* applicant.
4. Each company c ranks all applicants in c's preference order.

In real life situations, usually the number of companies is not equal to the number of applicants. Also, companies might want to employ one or more applicants. Applicants do not have to apply to all companies. Companies do not rank all applicants. But the setting above inherits all the issues of the original problem and makes the problem easier (and cleaner) to state.

Let $\mathbf{C} \times \mathbf{A}$ be the set of *all* ordered pairs of the form (c, a), where $c \in \mathbf{C}$ and $a \in \mathbf{A}$.

Definition 1. *A* **matching** *M is a set of ordered pairs each from $\mathbf{C} \times \mathbf{A}$ such that every member c of $\mathbf{C}$ and each member a of $\mathbf{A}$ appears in* **at most one pair** *in M.*

Informally, the matching M represents an assignment such that every company c makes an offer to at most one applicant, and every applicant a has an offer from at most one company.

Definition 2. *We say that the matching M is* **perfect** *if every member c of $\mathbf{C}$ and each member a of $\mathbf{A}$ appears in exactly one pair in M.*

Informally, a matching M is perfect if M satisfies the following property. According to M every company makes an offer to an applicant, and every applicant has offer.

Exercise 1. Explain why there are exactly $n!$ perfect matchings.

So far so good. Now, into our discussion, we bring in the ranking of the companies. We say that c *prefers* a to a' if c ranks a higher than a'. Similarly, an applicant a *prefers* c to c' if a ranks c higher than c'. Rankings of any two companies (or applicants) do not have to coincide.

Suppose that we have a perfect matching M. Taking the preferences into account, what can go wrong with the perfect matching M? Well, the perfect matching M might not be consistent with the rankings of applicants and companies in the following sense. The matching M might match (1) c_1 with a_1 and (2) c_2 with a_2; but, the company c_1 prefers a_2 to a_1, and the applicant a_2 prefers c_1 to c_2. Hence, in self-interests of both c_1 and a_2, the company c_1 might abandon the offer to a_1 and make an offer to a_2; in this case, a_2 accepts the offer from c_1 since a_2 prefers c_1 to c_2. In this case, we say that the matching M is *not self-enforcing* or *unstable.* Formally, we have the following definition:

Definition 3. *A perfect matching M* **is stable** *if M contains no two pairs (c_1, a_1) and (c_2, a_2), where c_1 prefers a_2 to a_1, and a_2 prefers c_1 to c_2. Otherwise, the matching is called* **unstable***.*

For clarity note that any stable matching is perfect matching, and perfect matching is a matching. Not every matching is perfect matching, and not every perfect matching is a stable matching.

Exercise 2. Assume that $\mathbf{C} = \{c_1, c_2, c_3\}$ and $\mathbf{A} = \{a_1, a_2, a_3\}$. Set up a preference list for all companies and all applicants. List all possible perfect matchings for your preference lists. List all stable matchings for your preference list.

So, two questions arise naturally:

1. Does there exist a stable matching for every set of preference lists?
2. Given a set of preference lists, can we efficiently build a stable matching if there is one?

3 Designing the algorithm

We show that there is a perfect matching for every set of preference lists. This answers the first question asked above. Our method of showing existence of a perfect matching also answers the second question. Namely, we give an efficient algorithm that, given a set of preference lists, builds a stable matching. One might naturally ask what we mean by efficiency. For the meantime we postpone our answer to this question and discuss the efficiency of algorithms in lectures to come. Now we present basic ideas of the algorithm:

- Initially, there is no job offer to any of the applicants. Suppose that a company c that has not made an offer to anyone now makes an offer to an applicant a. Is it possible to announce that this is the final decision of the company? That is, can we say that the pair (c, a) is in the stable matching M that we want to construct? Well, there are two possible scenarios:

  1. At a later stage a company c' might give an offer to a such that a prefers c' to c. Thus, self-interests of a lead to rejecting the offer from c and accepting the offer from c'.
  2. The applicant a might reject the offer from c since a could be waiting for an offer from a higher ranked company than c'. But rejection can be a danger because all higher ranked companies according to a's preference list might have given their offers already.

  So, the idea is to form the pair (c, a) at this stage, and we call the pair an *internship pair*. So, none of the sides is committed so far at this stage. In other words, the company c and the applicant a are temporarily engaged. At a later stage this pair might be dissolved, e.g. a might leave the company c due to an offer from a company that the applicant a ranks higher than the current company c.

- Suppose that we are now at a stage when some companies have not made any internship offers, and some applicants have no offers at all. We refer to such companies and applicants as being *free*. For instance, initially all companies and all applicants are free. There are also some companies that made internship offers to some applicants thus forming internship pairs. Next, the algorithm proceeds as follows. A free company c makes an internship offer to the highest rank applicant a in c's preference list. Note that a does not have to be free. At this stage, there are two possibilities for the applicant:

  1. The applicant is free now in which case an internship pair (c, a) is formed.

2. The applicant a is not free; so, a is part of an internship pair (c', a). Now, a compares c and c' according a's preference list. If a prefers c' to c then a keeps the internship; so, c does not make an internship offer with a. Otherwise, a chooses to do an internship at company c' thus forming a new internship pair (c, a). The old internship pair (c', a) has become invalid and c' is now free.

– The algorithm terminates, when no one is free. At this stage, all internship pairs are declared offers. This set of offers is the matching set M.

Taking the ideas above into our consideration, we now write the following algorithm, known as Gale-Shapley algorithm. For short, we might write GS-algorithm:

Algorithm 1 GS-algorithm:

```
Initially all $c \in \mathbf{C}$ and $a \in \mathbf{A}$ are free.
while there is a free company $c$ that hasn't offered internship to every applicant do
    Choose such a company $c$
    Let $a$ be the highest-ranked applicant in $c$'s preference list to whom $c$ has not made an offer
    if $a$ is free then
        $(c, a)$ becomes an internship pair
    else                                          ▷ $a$ is currently forms an internship pair with $c'$
        if $a$ prefers $c'$ to $c$ then
            $c$ remains free
        else                                                              ▷ $a$ prefers $c$ to $c'$
            $(c, a)$ forms an internship pair, and $c'$ becomes free
        end if
    end if
end while
return the set $M$ of internship pairs (that are now declared offers).
```

4 Analysis of the GS-algorithm

The overall goal is to show that the algorithm does what it is supposed to do. Namely, we want to show that the algorithm is correct in the sense that it outputs a stable matching. We stress that the algorithm is surprisingly easy to follow; this is the beauty of the algorithm. However, it is not so obvious that the algorithm returns a stable matching. It is not even obvious that the algorithm returns a perfect matching.

Consider the view of an applicant a during the execution of the algorithm. The applicant a gets an offer from a company c. At a later stage during the execution of the algorithm, a might get an offer from another company c'. The applicant a retracts from the previous offer only when c' has a higher rank than c in the applicant's preference list. In terms of the matching M the following happens. M contained the pair (c, a) but later the pair (c, a) is removed from M

because some free c' appeared such that a prefers c' to c. In this case the new M contains (c', a) but not (c, a). So, we have the following observation:

Observation 1 *Once an applicant a receives an internship offer, the applicant will always have an offer (not necessarily the same offer). Moreover, the ranks of the companies that the applicant accepts the offers from get better and better (in terms of a's preference list).*

Consider the view of a company c. Company c may alternate between being free and not free. Suppose c is free and makes an offer to the highest rank applicant a in the company's preference list. The applicant may accept or may reject the offer. If the applicant accepts the offer, then an internship pair (c, a) is formed. Assume that later the applicant a retracts from the offer. Then c becomes free again. The next offer of company c will be to an applicant whose rank is worse than that of the applicant a:

Observation 2 *The ranks of the applicants to whom the company c makes offers get worse and worse (in terms of the preference list of the company).*

Now assume that company c makes an offer to the applicant a during an iteration of the algorithms. If a' has a higher rank in the company's preference list, then c must have already made an offer to a' in some of the previous iterations of the algorithm. This what the instructions within the while loop guarantee.

Observation 3 *If company c makes an internship offer to applicant a, then c must have already made offers to all applicants ranked higher than a in the companies preference list.*

Now we would like to show that the algorithm always terminates. Note that every company c offers an internship to each applicant at most once. So, the company c makes at most n offers. This implies that the number of iterations of the while loop is at most n^2 (since there are exactly n applicants). So, we have the following observation:

Observation 4 *The algorithm terminates within at most n^2 iterations of the while loop.*

Now we turn our interest to understanding the way the set M behaves during the execution of the algorithm. Our goal is to show that M returned by the algorithm is a stable matching. This boils down to the analysis of the while loop of the algorithm. First we show an easier fact:

Fact 1 *Initially M is a matching. M remains matching after each iteration of the while loop.*

Proof of the fact. Recall a set S is a matching if every member of $\mathbf{C}$ and every member of $\mathbf{A}$ appears in at most one pair in S. Initially M contains no pair; so, the first part of the fact is true.

Now we want to show that the second part is true. For that we consider an iteration of the while loop. Let M_1 be the value of M before the iteration, and M_2 be the value of M after the iteration. It suffices to show that if M_1 is a matching then M_2 is also a matching. During the iteration a free company c is chosen. Then c offers an internship to the highest ranked applicant a to which c has not made an offer yet. Here is a simple analysis.

Since c is free, c does not belong to any pair in M_1. If a is free, then a too does not belong to any pair in M_1. The set M_2 in this case is just M_1 together with the pair (c, a). It is clear that M_2 is a matching. If a is not free, then there are two cases. The first case is when a prefers its current company to c. In this case M_2 is just M_1. Nothings has changed, and so M_2 is a matching. The second case is when a prefers c to a's current company c'. In this case, M_2 is obtained from M_1 by removing (c', a) from M_1 and adding (c, a) to M_1. It is now not too hard to understand that M_2 is still a matching set. This proves the fact.

Exercise 3. In the last part of our argument above M_2 is obtained from M_1 by removing (c', a) from M_1 and adding (c, a) to M_1. Write down your reasons explaining why M_2 is a matching.

Now we show that M is a perfect matching.

Fact 2 *The algorithm returns a perfect matching.*

Proof. Consider the output M of the last iteration of the algorithm. We know that M is a matching. We want to show that M is a perfect matching. Let us see what happens if M is not a perfect match. Then either some $c \in \mathbf{C}$ or some $a \in \mathbf{A}$ has no pair in M.

Assume that c has no pair in M. Then, after the last iteration of the while loop, c is free. After the last iteration no applicant is free. Otherwise, the iteration would not be the last iteration; this would be a contradiction. Also, as we already observed, company c has made offers to all the applicants. Therefore, all applicants have offers according to M. These imply that the number of companies is greater than the number of applications. This is a contradiction with our assumption that the number of companies equals the number of applicants. A similar reasoning shows that a must have a pair in M. The next exercise finishes the proof.

Exercise 4. Finish the proof by showing that every applicant a must have a pair in M.

Finally, we came to the main part of our argument. We show that the algorithm is correct in the sense that it satisfies its specification (that is, the algorithm outputs a stable mathcing):

Fact 3 *The output M of the algorithm is a stable matching.*

Proof. To show that M is a stable matching, we need to prove the following. For any pair (c, a) in M no pair (c', a') exists such that c prefers a' to a, and a prefers c to c'. Consider the iteration, say iteration t, at which c makes the last internship offer. At this iteration the offer must be made to a (since (c, a) is in M). Note that any applicant a' that c prefers to a satisfies the following two conditions:

1. a' is not free at this stage.
2. Company c has made an offer to a' before stage t.

The first part is true as otherwise, at this iteration c would make an offer to a' that contradicts the choice of a. For the second part, if c has not made an offer to a' before iteration t, then c must remain free at this iteration. Again this contradict the choice of a. Thus, all a' that c prefers to a have no self-interest to accept an offer from c. More formally, all these applicants a' have already internship offers by companies that they prefer to c. Thus, by Observation 1 a' prefers the company c' that made the last offer to a'. We have proved the fact.

5 Implementation of Gale-Shapley algorithm

An implementation of an algorithm is clearly dependent on the designer (programmer). The designer chooses an appropriate data structure, programming language, and programming techniques. For each algorithm one needs to choose data structures that make the algorithm efficient and easy to implement.

Here we consider one possible implementation of the GS-algorithm. A simple way to keep a list of n elements, such as, the preference list (of applicants for a job) of a company c is to use *array* of length n. The *array data structure* is standard in many programming languages. These data structures have the following properties.

- The array data structure is specified by the length n of the list.
- One can access the ith element in the array in constant time.
- Determining if an element e belongs to the array takes time proportional to n.
- In case the array is sorted then determining if an element e belongs to the array takes time proportional to $log(n)$; this can be done through the binary search for instance.

The array data structure possesses two operations: *insert* and *delete*. Inserting an element to position i requires shifting all elements at position k with $k \geq i$ to positions $i+1$. Similarly, deleting an element at position i requires shifting all elements at position k with $k \geq i$ to positions $i-1$.

Another way to maintain a list of n elements is to use the *linked list* data structure. This data structure is useful when the list is dynamically changing. For instance, the list of free companies in the GS-algorithm sometimes shrinks and sometimes grows. For such dynamically changing list the array data structure might not be appropriate. In a linked list, we sequence elements of the list together where each element points to the next in the sequence. Namely, for each element e on the list, we have a pointer to the next element. There is pointer *Last* that points to the last element in the list. We also have a pointer $First$ that points to the first element in the list. In linked list data structure, we can traverse the entire list in time proportional to n by starting at $First$ and following the pointers to the next element until we reach *Last*. Also, the data structure has *insert* and *delete* operations. Insert operation, given an element e, inserts the element to the first position; this takes a constant amount of time since we we need to just re-direct the pointer $First$ to e, and set up a pointer from e that points to the previously first element of the list. The delete operation delete the first element in the list; this is also a constant cost operation.

We can enrich the linked list data structure, by putting the reverse pointers that point from element e to the previous element in the list. This enriched data structure is known as *doubly linked list* data structure. Thus as observed, in contrast to arrays, *insert* and *delete* operations on (doubly) linked lists take constant time. A disadvantage of this data structure is that accessing the ith element takes time proportional to i. So, unlike arrays, we cannot find the ith element of the (doubly) linked list in constant time.

We now look at the GS-algorithm in details and think about what data structures we should use in order to implement the algorithm. First we describe how we represent the input data.

1. We represent companies and applicants as arrays $\mathbf{C} = (c_1, \ldots, c_n)$ and $\mathbf{A} = (a_1, \ldots, a_n)$.
2. We have two dimensional arrays $CompPref(c, i)$ and $ApPref(a, j)$ for companies and applicants, respectively. The entry $CompPref(c, i)$ represents the ith applicant in c's preference list. Similarly, $ApPref(a, j)$ represents the jth company in a's preference list.

The space needed to represent this input data equals $2(n+n^2)$. This is called the *size* of the input data. The notion of size of the input data is an important concept. The efficiency of algorithms will be defined in terms of the sizes of inputs.

Exercise 5. We postulated that the number of companies equals the number of applicants. Suppose we remove this postulate. What will the size of input data be in this case?

We need to examine each step of the algorithm and understand what data structure allows us to implement the algorithm efficiently. Our implementation should address each of the following four important issues:

1. We need to identify a free company.
2. For every company c, we need to find the highest-ranked applicant a such that a has not had an offer from company c.
3. For applicants a, we need to decide if a has an offer. Moreover, in case a has an offer we need to identify the company c such that (c, a) is currently an internship pair.
4. For an applicants a, companies c and c', we need to detect if a prefers c to c'.

Identifying a free company can be done as follows. We maintain the set of free companies as a linked list. When the algorithm selects a free company, the first company c in the list is taken. When c makes an offer and the offer is accepted, we delete c from the list. If a new company c' becomes free, then we insert c' into the first position of the linked list. We noted that insert and delete operations on linked list data structure is a constant cost operation. This justifies the use of linked list data structure for representing free companies.

Consider a company c. We show how we find the highest-ranked, in the company's preference order, applicant a such that a has not had an offer from company c. With company c, we associate a counter $Next(c)$. Initially, $Next(c) = 1$. So, $Next(c)$ indicates the highest-ranked applicant a which has not had an offer from company c. When c makes an offer to a, the applicant a is selected in such a way that $a = CPref(c, Next(c))$. After the offer is made, the value of the counter $Next(c)$ is incremented by 1.

Suppose that c makes an offer to applicant a. To find out if a is free or has an internship offer, we keep the value $Current(a)$ that indicates the company c such that (c, a) is currently an internship pair. Initially, $Current(a) = free$ indicating that applicant a is free.

It is now easy to note that, with our set-up, each of of the above operations in (1), (2), and (3) can be executed in constant time. Now we explain why the operation described in (4) can also be executed in constant time.

When a company c makes an internship offer to a, we check whether $Current(a) = free$; if $Current(a) = free$, we set $Current(a) = c$. Otherwise, we look at $c' = Current(a)$, and compare c and c' in the a's preference list. We want this comparison to be made in constant time. One way to compare c and c' is to search for these companies in the array $ApPref$; this is acceptable and the search runs in time proportional to n as opposed to constant time for operations in (1), (2), and (3). One way to ensure a constant time operation for (4) is the following. Consider the $ApPref$ two dimensional array. Do one pass through the array and create a new double array $Rank(c, a)$ of size $n \times n$ such that for each c and a, $Rank(c, a)$ contains the rank of c with respect to the preference of the applicant a. We do this at the start of the algorithm. Construction of the array $Rank(c, a)$ takes time proportional to n^2. Having this array, we now can compare c and c' in constant time.

Now we connect the describe data structure with Observation 4:

Fact 4 *The data structures described above, on all inputs with n applicants, n companies, and their preference lists, runs in time proportional to n^2.*

Since, the sizes of inputs with n applicants, n companies, and their preference lists, equal n^2 (as we explained above), our implementation actually runs in time proportional to size of the inputs. In other words, our algorithm is linear on the size of input. Below is pseudocode that implements GS-algorithm.

Algorithm 2 Implementation of GS-algorithm

```
INPUT  $\mathbf{C} = (c_1, \ldots, c_n)$, $\mathbf{A} = (a_1, \ldots, a_n)$, $CompPref(c, i)$, $ApPref(a, j)$
OUTPUT  A collection of internship pairs.
  Initialise an empty collection $M$ of internship pairs.
  Initialise a linked list $free$ containing $c_1, c_2, \ldots, c_n$.
  Create a counter $Next(c)$ for every company $c$ and set $Next(c) \leftarrow 1$.
  Create $Current(a)$ for every applicant $a$ and set $Current(a) \leftarrow free$.
  Initialise an $n \times n$-array $Rank$.                    ▷ Create the array $Rank$
  for $i \in \{1, \ldots, n\}$ do
      for $j \in \{1, \ldots, n\}$ do
          $Rank(ApPref(a_i, j), a_i) \leftarrow j$
      end for
  end for
  while $free$ is not empty do
      $c \leftarrow free.get(0)$                               ▷ Choose a free company $c$
      $a \leftarrow CompPref(c, Next(c))$
      $Next(c) \leftarrow Next(c) + 1$                         ▷ $c$ makes an offer to $a$
      if $Current(a) = free$ then                              ▷ $a$ is free
          $M.add((c, a))$                                      ▷ $(c, a)$ becomes an internship pair
          $free.remove(0)$
      else
          $c' \leftarrow Current(a)$                           ▷ $a$ is currently forms an internship pair with $c'$
          if $Rank(c, a) < Rank(c', a)$ then                   ▷ $a$ prefers $c$ to $c'$
              $M.add((c, a))$                                  ▷ $(c, a)$ becomes an internship pair
              $free.remove(0)$
              $free.add(c', 0)$                                ▷ $c'$ becomes free
          end if
      end if
  end while
  return $M$
```