CPSC 320 2023S2: Assignment 1 Solutions

3 Dancing partners!

CPSC 320 students are planning a big post-semester celebration! There will be a dance! Two students, Jihong and Andie have volunteered to coordinate dancing partners. They want to adapt the stable matching algorithm to pair up students (fortunately there are an even number of students in the class). Jihong and Andie decide to tackle the problem when each student provides a complete ranking of the other students in the class, in the order that they prefer to work with them.

Formally, an instance of the Dancing Partners (DP) problem specifies, for each of n students, a fully ranked preference list of all other students in the class. Assume that n is even. The preference lists can be represented as a matrix P, where $P[i, j]$ is the student of rank j in student i's list.

A *stable pairing* is a perfect matching M of the students with no instability, i.e., no pair (u, v) such that $(u, u') \in M$, $(v, v') \in M$, but u prefers v to u' and v prefers u to v'.

Jihong has adapted the Gale-Shapley algorithm for the DP problem. Her algorithm is below. She's not sure yet if it always finds a stable matching, so the algorithm is structured to output useful information in situations where a stable matching might not be found. This problem will focus on correctness aspects of her algorithm, and the next problem will focus on runtime.

```
1:  function DANCING-PARTNERS(n, P)
2:      ▷ n ≥ 1 is the number of students
3:      ▷ P is the collection of complete preference lists (>) of the students
4:      ▷ attempt to find a stable pairing M for the instance (n, P)
5:      ▷ initially no student is reserved
6:      while no student's preference list is empty and some student isn't reserved do
7:          choose any student u who isn't reserved
8:          let v be the highest-ranked person remaining in u's preference list
9:          if v has reserved no-one then
10:             v reserves u
11:         else
12:             v reserves u, releasing its previously reserved student x, if any
13:             ▷ v prefers u to x, and x is no longer reserved
14:         for each w on v's list such that u >_v w do
15:             prune w from v's list and prune v from w's list
16:
17:     if all lists have size 1 then
18:         ▷ For any (u, v), u is reserved by v iff v is reserved by u
19:         return the corresponding pairing
20:     else if some student's list is empty then
21:         return "no stable matching"
22:     else
23:         return "inconclusive"
```

1. (3 marks) Complete the following table, which traces an execution of the algorithm when $n = 6$. The first three rounds are done for you.

Input preference lists:

1	6	2	4	3	5
2	4	5	1	3	6
3	4	2	5	6	1
4	3	2	6	5	1
5	6	1	2	4	3
6	5	1	4	3	2

SOLUTION:

Round	u	v	Reserves, releases	Updated preference lists					
1	1	6	6 reserves 1	1	6	2	4	3	5
				2	4	5	1	3	
				3	4	2	5	1	
				4	3	2	5	1	
				5	6	1	2	4	3
				6	5	1			
2	2	4	4 reserves 2	1	6	2	3	5	
				2	4	5	1	3	
				3	4	2	5	1	
				4	3	2			
				5	6	1	2	3	
				6	5	1			
3	3	4	4 reserves 3, releases 2	1	6	2	3	5	
				2	5	1	3		
				3	4	2	5	1	
				4	3				
				5	6	1	2	3	
				6	5	1			
4	4	3	3 reserves 4	1	6	2	5		
				2	5	1			
				3	4				
				4	3				
				5	6	1	2		
				6	5	1			
5	5	6	6 reserves 5, releases 1	1	2	5			
				2	5	1			
				3	4				
				4	3				
				5	6	1	2		
				6	5				
6	6	5	5 reserves 6	1	2				
				2	1				
				3	4				
				4	3				
				5	6				
				6	5				
7	1	2	2 reserves 1	same as in round 6					
8	1	2	1 reserves 2	same as in round 6					

2. (3 marks) Andie decides to investigate whether there are instances of the DP problem with no stable pairing. Help them out! Find a problem instance for which there is no stable pairing, and explain why the instance has no stable pairing.

   SOLUTION: Here is an example for which no stable pairing exists:

1	2	3	4
2	3	1	4
3	1	2	4
4	1	2	3

   Any student i who is paired with student 4 will form an instability with the student that likes i best. For example, if 2 is paired with 4, then since 1 likes 2 best, the pair (1,2) form an instability.

3. (2 marks) Jihong needs to convince Andie why she believes that, when line 12 of the DANCING-PARTNERS algorithm is executed, v must prefer u to x. Help Jihong out! A short explanation is sufficient.

   SOLUTION: Pruning is symmetric (line 15), and so if v is on u's list, u must also be on v's list. Also, since x is reserved by v, any student w that was initially lower than x on v's preference list has been pruned from v's list (line 15). So v must prefer u to x.

4. (4 marks) Andie is starting to think that the DANCING-PARTNERS algorithm has real potential! They define a prune of w from v's list to be *safe* if v and w cannot be partners in *any* stable matching, and show that all prunes done by the algorithm are safe. See if you can do the same. You can follow Andie's scheme of proof by induction, outlined below, but if you prefer to prove the claim in a different way, that is fine too.

   Claim: All prunes done by the algorithm are safe.

   SOLUTION:

   Proof: We show by induction on i that for any execution of the algorithm, on any valid input, the first $2i$ prunes are safe (or all prunes are safe if the execution has fewer than $2i$ prunes.)

   Base case: If $i = 0$, the induction statement is trivially true since there are no prunes.

   Induction hypothesis: Suppose that the induction statement is true for some $i \geq 0$, that is, for any execution of the algorithm, on any valid input, the first $2i$ prunes are safe.

   Induction Step: Consider any execution of the algorithm on some fixed input that has more than $2i$ prunes. Consider the next two prunes, say when u has just been reserved by v, causing w to be pruned from v and v to be pruned from w, where $u >_v w$.

   Suppose to the contrary that v and w are partners in some stable matching M. Since M is stable, u prefers its partner in M, say z, to v. But in order for u to be reserved by v, z must have been pruned from u's list earlier in the execution. But then by the induction hypothesis, there is no stable matching containing the pair (u, z), and we get our contradiction. Therefore the first $2(i + 1)$ prunes of the execution must be safe, completing the proof.

5. (2 marks) Show that if the **while** loop of DANCING-PARTNERS terminates when one person's list is empty, then no stable matching exists.

   SOLUTION: From Andie's claim in part 4 above, if one person's list is empty, then there is no-one that they can be matched with in a stable matching. So there cannot be any stable matching.

6. (2 marks) Explain why the assertion in the comment of line 18 must hold, that for any pair of students (u, v), u is reserved by v if and only if v is reserved by u.

SOLUTION: If line 18 is reached, no list is empty, and so all students must be reserved. Since prunes are symmetric and since at line 18 lists have size 1, if u is in v's list then v must be in u's list, and each must have reserved the other, since the individual reserved by a student is always on that student's list.

7. (2 marks) Show that if DANCING-PARTNERS terminates where everyone has a list of size 1, then the corresponding matching is stable.

   SOLUTION: Suppose to the contrary that (x, y) is an instability, where x is paired with x' and y is paired with y'.

   The students pruned from x's list are those who are matched with someone they like better than x, or those who x likes less than its match. So x cannot be in an instability. This is true for all x, and so the matching must be stable.

8. (1 mark) What is the output of the algorithm on the following problem instance? No justification needed.

1	4	6	2	5	3
2	6	3	5	1	4
3	4	5	1	6	2
4	2	6	5	1	3
5	4	2	3	6	1
6	5	1	4	2	3

   SOLUTION: On this instance, the algorithm outputs "inconclusive".

4 Dancing partners, part 2

1. (3 marks) Explain why the **while** loop of the DANCING-PARTNERS algorithm is executed at most n^2 times.

   SOLUTION: On each iteration, progress is made in one of two ways: (i) either a student who has not previously reserved anyone can now reserve someone, or (ii) a student who has already reserved someone can reserve someone that they prefer better. Option (i) can only happen n times in total, since once a student has reserved someone, they will always have someone reserved for the rest of the algorithm. Option (ii) can happen at most $(n-1)^2$ times in total, because each time it happens, the student reserves someone they prefer to any previously reserved student, and there are $n-1$ students initially on their preference list. So the total number of iterations is at most $(n-1)^2 + n$, which is bounded by n^2.

2. (4 marks) Describe data structures that make it possible to execute each iteration of the **while** loop in $O(1)$ time, *ignoring the pruning step in lines 14 and 15.* For each data structure that you introduce, explain how you initialize it, how it is updated as the algorithm proceeds, and what is the time for initialization and updates. Assume that the only data structure provided as input is the data structure of student preferences, namely a matrix P, where $P[i, j]$ is the student of rank j in student i's list.

   [Note: Lines 14 and 15 might take $\Theta(n)$ time on one iteration, but since there are $O(n^2)$ prunes overall, the total time for lines 14 and 15, over all iterations, is $O(n^2)$. But in your work you can ignore lines 14 and 15.]

   SOLUTION:

   Line 6: Checking the condition of the while loop.

   - In order to know when a student's list becomes empty, we can maintain a Boolean variable `list-empty`, which is initially set to false. Each time a student is pruned from i's preference list (see lines 14 and 15 below), a check is done to see if the list becomes empty, in which case `list-empty` is set to true. Initializing and checking the value of this variable takes $O(1)$ time.
   - In order to determine whether some student isn't reserved, we can maintain a list, called `unreserved`, of unreserved students. Initially the list is created to contain all students, taking $O(n)$ time. Updates to the list are described below (lines 7 and 13). To determine whether all students are reserved, we can simply check whether the list is empty; this can be done in $O(1)$ time.

     Therefore, after a cost of $O(n)$ for initialization, the condition of the **while** loop can be tested by doing two checks, each taking $O(1)$ time.

   Line 7: Choosing an unreserved student. To choose an unreserved student, simply remove the student from the start of the `unreserved` list.

   Line 8: Finding the highest-ranked person in a list. From the input matrix P, for each student i, we can initially create a doubly-linked list, say Ranking$[i]$, with one node for each of $n-1$ students in order of ranking. This takes $O(n^2)$ time in total. Then in line 8 we can choose v to be the first person on this list, in $O(1)$ time. In addition, we can initially create a matrix Pointers$[i, j]$, which has a pointer to the student representing student j in student i's list.

Line 9: Determining whether v has reserved someone. We can use yet another array, `reserved`$[1..n]$, where `reserved`$[i]$ stores the student reserved by student i, or 0 if no student is reserved by i. Initially all entries of this array should be set to 0.

Lines 10 and 12: Reserving and releasing. Whenever a student, say u, is reserved by another student, say v (line 10 or 13), `reserved`$[v]$ is set to u. This automatically releases a student, say x, which was previously reserved by v (line 13).

Line 13: Releasing a reserved student. To release a student, the student can be added to the end of the `unreserved` list.

Line 17: Testing if all lists have size 1. We can simply traverse each list up to the second node if any, and in $O(1)$ time determine if the list length is 0, 1, or more than 1. Since there are n lists, this takes $O(n)$ time total.

Line 20: Testing if some student's list is empty. This is identical to the first bullet under line 6, and takes $O(1)$ time.

5 Permutations

The following definitions are exactly as in the tutorial. Let $\pi[1..n]$ and $\pi'[1..n]$ be two permutations over a set of n integers. We say that $\pi < \pi'$ if and only if for some i, $1 \leq i < n$,

$$\pi[1..i-1] = \pi'[1..i-1] \text{ and } \pi[i] < \pi'[i]. \tag{1}$$

For example, if the set of integers is $\{1, 5, 6, 8\}$, then $5168 < 5618$ (the conditions hold with $i = 2$), and also $5168 < 5186$ (the conditions hold with $i = 3$).

Let $\pi_1, \ldots, \pi_{n!}$ be an ordering of all $n!$ permutations of a set of n integers. This is a *lexicographic ordering* if $\pi_k < \pi_{k+1}$ for all $k, 1 \leq k < n!$.

1. (4 marks) The tutorial solution describes criteria for determining if one permutation $\pi'[1..n]$ of a set of n integers directly follows permutation $\pi[1..n]$ in lexicographic order. Prove that these criteria are correct.

   SOLUTION: The criteria described in the tutorial solution are: Find the smallest value, say i, in the range $[0..n-1]$ such that $\pi[i+1..n]$ is sorted in descending order. If $i = 0$, π must be the last permutation in lexicographic order, so π' cannot follow π.

   Otherwise, if $i \geq 1$, find j, where $\pi[j]$ is the smallest value in $\pi[i+1..n]$ that is larger than $\pi[i]$. Then π' follows π in lexicographic order if and only if

   (a) $\pi'[1..i-1] = \pi[1..i-1]$,
   (b) $\pi'[i] = \pi[j]$, and
   (c) $\pi'[i+1..n]$ is sorted in ascending order.

   To show that these criteria are correct, we first show that if π and π' satisfy the three criteria (a), (b), and (c) above then (i) $\pi < \pi'$ and (ii) for any other permutation π'' such that $\pi < \pi''$, it is also the case that $\pi' < \pi''$. If both (i) and (ii) hold, π' must directly follow π lexicographically.

   To see that (i) is true, note that by criterion (a), $\pi'[1..i-1] = \pi[1..i-1]$ and also $\pi[i] < \pi[j] = \pi'[i]$. So both conditions of Equation (1) hold.

   Next we show (ii). Consider any π'' such that $\pi < \pi''$. Let i^* be the smallest index where $\pi[i^*] \neq \pi''[i^*]$. Since $\pi < \pi''$, it must be that $\pi[i^*] < \pi''[i^*]$. There are three possibilities:

- $i'' < i$. Then $\pi'[1..i^*-1] = \pi[1..i^*-1] = \pi''[1,,i^*-1]$ and also $\pi'[i^*] = \pi[i^*] < \pi''[i^*]$. So $\pi' < \pi''$.
- $i^* = i$. Then $\pi'[1..i-1] = \pi[1..i-1] = \pi''[1,,i-1]$. Also, since $\pi'[i] = \pi[j]$ where $\pi[j]$ is the smallest element in $\pi[i+1..n]$, it must be that $\pi'[i] < \pi''[i]$. So again $\pi' < \pi''$.
- $i^* > i$. This case cannot happen, because the entries in $\pi[i+1..n]$ are sorted in descending order, so $i^* > i$ would imply that $\pi[i^*]$ would be less than $\pi''[i^*]$. This would contradict our assumption that $\pi < \pi''$.

We also need to show that if π' directly follows π in lexicographic order, then the three criteria hold. Since π' directly follows π in lexicographic order, we know that $\pi < \pi'$ and there is no π'' with $\pi < \pi'' < \pi'$. Let i and j be as described for the criteria. That is, i is the smallest value in the range $[0..n-1]$ such that $\pi[i+1..n]$ is sorted in descending order, and $\pi[j]$ is the smallest value in $\pi[i+1..n]$ that is larger than $\pi[i]$. It must be that $i \geq 1$, since $\pi < \pi'$. We need to show that π' satisfies criteria (a), (b), and (c).

Criterion (a): Suppose that $\pi[1..i-1] \neq \pi'[1..i-1]$ and that the first (leftmost) position where they differ is ℓ. Since $\pi < \pi'$, it must be that $\pi[\ell] < \pi'[\ell]$. Let π^* be the permutation obtained by swapping $\pi[i]$ and $\pi[j]$ and then sorting the remaining elements in descending order. Then we also have that $\pi < \pi^* < \pi'$, contradicting the fact that π' directly follows π. So (a) must hold.

Criterion (b): If $\pi'[i] \neq \pi[j]$ then there are two cases.

- $\pi'[i] = \pi[i]$. Then it must be that $\pi'[i+1..n] \neq \pi[i+1..n]$, while both of these subarrays contain the same numbers. But since the numbers of $\pi[i+1..n]$ are sorted in descending order, at the leftmost place, say j', where $\pi[i+1..n]$ differs from $\pi[i+1..n]$, it must be that $\pi'[j'] < \pi[j']$. Since this contradicts the fact that $\pi < \pi'$, this case cannot arise.
- $\pi'[i]$ is one of the numbers in $\pi[i+1..n]$, but is not the smallest number $\pi[j]$. In this case, π^* (defined above under Criterion (a)) is less than π', contradicting the fact that π' follows π directly in lexicographic order. So this case cannot arise either, and we can conclude that $\pi'[i] = \pi[j]$ and criterion (b) holds.

Criterion (c): Of all the permutations that satisfy criteria (a) and (b), the permutation π^* in which the remaining numbers are sorted in ascending order is less than every other such permutation. So π^* must be equal to π', since π' is less than every other permutation that follows π. Therefore the elements of $\pi[i+1..n]$ are sorted in ascending order, in which case criterion (c) holds and we are done.

2. (4 marks) Design an efficient algorithm that takes as input permutation $p[1..n]$ of a set of n integers, and updates $p[1..n]$ to be the next permutation in lexicographic order. For example, if $p[1..4]$ is initially 7325, then when the algorithm completes, $p[1..4]$ should be 7352. If there is no lexicographically next permutation, then p can remain unchanged. Try to find an algorithm that is as efficient as possible in the worst case, ignoring constants.

SOLUTION:

```
1: procedure GENERATE-LEXICOGRAPHICALLY-NEXT-PERMUTATION(p[1..n])
2:     ▷ p[1..n] is a permutation of n distinct integers
3:     ▷ update p[1..n] to be the lexicographically next permutation
4:     ▷ leave p[1..n] unchanged if there is no lexicographically next permutation
5:
6:     ▷ find the smallest i ∈ [0..n − 1] such that p[i + 1..n] is sorted in descending order
7:     i = n − 1
8:     while (i ≥ 1) AND (p[i] > p[i + 1]) do
9:         i ← i − 1
10:
11:    ▷ if i = 0, there is no lexicographically next permutation; do nothing.
```

```
12:     if i > 0 then
13:         ▷ find j, where p[j] is the smallest value in p[i + 1..n] that is larger than p[i]
14:         j ← i + 1
15:         while (j < n) AND (p[j + 1] > p[i]) do
16:             j ← j + 1
17:
18:         swap the entries p[i] and p[j]
19:         reverse the entries in p[i + 1..n]        ▷ so that they are in ascending order
```

3. (4 marks) Explain why your algorithm is correct.

   SOLUTION:

   We explain how the algorithm uses the "lexicographically next" criteria from the previous part of the problem, to update $p[1..n]$.

   The first **while** loop at lines 8-9 finds the smallest $i \in [0..n-1]$ such that $p[i+1..n]$ is sorted in descending order.

   If this **while** loop completes with $i = 0$, $p[1..n]$ is not changed, since it must be the last permutation in lexicographic order, and the algorithm terminates.

   Otherwise, the second **while** loop at lines 15-16 finds j such that $p[j]$ is the smallest value in $p[i+1..n]$ that is larger than $p[i]$. Note that this choice of j implies that if $j < n$ then $p[i] > p[j+1]$, and if $j > i+1$ then $p[j+1] < p[i]$. So if $p[j]$ is replaced by $p[i]$ then the array $p[i+1..n]$ is still sorted in descending order.

   In the remaining algorithm, the entries of $p[1..i-1]$ remain unchanged, so criterion (a) is satisfied by the final array $p[1..n]$.

   At line 18, $p[i]$ and $p[j]$ are swapped, so criterion (b) is also satisfied by the final array $p[1..n]$.

   Finally, as noted above, after this swap, the subarray $p[i+1..n]$ remains sorted in descending order. So once the entries in this subarray are reversed, they are in ascending order, and criterion (c) is satisfied.

   Since the updated array $p[1..n]$ satisfies all of the criteria, it must be the lexicographically next permutation following the initial array $p[1..n]$.

4. (2 marks) Explain why any algorithm for generating the lexicographically next permutation must take $\Omega(n)$ time in the worst case.

   SOLUTION: The lexicographically next permutation after $1n(n-1)..32$ is $213..(n-1)n$, requiring that all n entries of the array $p[1..n]$ change position. This takes $\Omega(n)$ time.

5. (2 marks) What is the worst case big-O runtime of your algorithm? Justify your answer.

   SOLUTION: The first **while** loop runs in $O(n)$ time, since i's initial value is $n-1$, i decreases on each iteration, and never goes below 0. Executing the single decrement instruction in the loop, and checking the loop condition, which involves two comparisons, take $O(1)$ time. Similarly, the second **while** loop takes $O(n)$ time since the range of j is within $[1..n]$, j increases on each iteration and the condition and single instruction in the loop take $O(1)$ time. The other instructions within the **if** statement take $O(1)$ time, as does the swap. The reverse can be implemented in $O(n)$ time as a sequence of swaps. So overall, $O(n)$ time is sufficient.

6. (2 marks) What is the best case big-O runtime of your algorithm? Justify your answer.

   SOLUTION: On an input of size n the algorithm runs in $O(1)$ time if, for example, the last two elements of $p[1..n]$ are sorted in ascending order. In this case, the value of i in line 9 is $n-1$, and the rest of the algorithm takes $O(1)$ time.