Introduction to Algorithms	**Solution Sketches for Homework 1**
CS 4820, Spring 2014	**Due 11:59pm Thursday, January 30**

(1) *(5 points)* You are helping a team of scientists to study a population of birds. There are B distinct birds in the population, and the birds were kept under observation over a period of time during which their mating behavior was logged. This resulted in a sequence of M log entries, each identifying the specific pair of birds that mated with each other. The scientists are interested in two individuals, x and y, and they wish to know whether the population can be divided into two disjoint sub-populations, containing x and y respectively, such that no individual in the first sub-population ever mated with an individual in the second sub-population during the observation period. Design an algorithm with running time $O(B+M)$ that solves this problem.

Remark: *This problem is intended as a diagnostic exercise, to verify that you have mastered the prerequisites for CS 4820. It can be solved using techniques taught in the prerequisite courses. The material that was covered in this week's lectures is not particularly applicable.*

Solution

Let G denote an undirected multigraph whose vertex set $V(G)$ is in one-to-one correspondence with the birds, and whose edge set $E(G)$ is in one-to-one correspondence with the M log entries; for every log entry representing the mating of two individuals a and b, the corresponding element of $E(G)$ is an edge whose two endpoints are the vertices corresponding to birds a and b. For the rest of this solution, we will not distinguish between a bird and the vertex in $V(G)$ that represents that bird.

The algorithm starts by building the adjacency list representation of G. It then runs a breadth-first search (BFS) or depth-first search (DFS) starting from x. For the sake of concreteness, we will assume the algorithm uses BFS for the rest of this solution, but the analysis is unchanged if we use DFS instead. If y is encountered during the BFS then it outputs "no", otherwise "yes."

The running time of BFS on our graph G is

$$O(|V(G) + E(G)|) = O(B + M).$$

Building G from the input data can also be accomplished within the same time bound: it takes $O(B)$ time to initialize the vertex set of size B, and processing each of the M log entries can be done in constant time: a log entry specifying a mating pair u, v is processed by inserting u into the adjacency list of v and vice-versa. (There is no need to remove duplicate entries in these lists, because BFS is a valid algorithm for testing reachability in multigraphs.) Thus, the combined running time of the two stages of our algorithm (building and searching G) is $O(B + M)$ as desired.

To prove correctness, define a *suitable partition* of $V(G)$ to be a pair of disjoint sets A, B such that every element of $V(G)$ belongs to one and only one of them, $x \in A$, $y \in B$, and $E(G)$ has no edges joining A to B. The proof of correctness consists of justifying the following two statements.

If the algorithm outputs "yes", then a suitable partition exists. The algorithm outputs "yes" if y is not encountered during BFS starting from x. In this case let A denote the set of vertices encountered by the BFS and let B denote its complement. $E(G)$ has no edges joining A to B (as otherwise the endpoint of the edge that belongs to B would have been encountered when processing the endpoint that belongs to A, if not earlier) so A, B is a suitable partition.

If a suitable partition exists, then the algorithm outputs "yes." If A, B is a suitable partition, then no element of B will ever be encountered by BFS. Indeed, since BFS starts at A, then the first time it encounters an element of B it would have to be scanning an edge that joins A to B, violating the hypothesis that A, B is a suitable partition. Therefore, no element of B will ever be encountered, and in particular y will never be encountered and the algorithm will output "yes."

(2) *(15 points)* Tolstoy famously wrote, "*Happy families are all alike; every unhappy family is unhappy in its own way.*" Bad algorithms are sort of like unhappy families: each bad algorithm is bad in its own way. Some of them output incorrect answers, others run forever, and others always terminate with the correct answer but may take exponential time to do so. Here we will explore all three types of bad behavior, by analyzing three algorithms that attempt to solve the stable matching problem.

(2.a) (Cost Minimization)
Assign a *cost* to each pair (x, y), by summing the positions where each member occurs on the other one's preference list. For example, if y is the third woman on x's preference list and x is the fifth man on y's preference list, then $\text{cost}(x, y) = 8 = 3 + 5$. Consider the algorithm that outputs the minimum-cost perfect matching, i.e. the perfect matching $\mathcal{M}$ that minimizes

$$\text{cost}(\mathcal{M}) = \sum_{(x,y)\in\mathcal{M}} \text{cost}(x, y).$$

Give an example of an input instance that causes this algorithm to produce an output that is *not* a stable perfect matching. In analyzing your example, describe the algorithm's output and show why it is not a stable perfect matching. If there is more than one cost-minimizing perfect matching, you may assume that the algorithm makes the worst possible choice.

(Remark: It turns out that the minimum-cost perfect matching can always be computed in polynomial time, although this exercise doesn't require you to know that fact. If you're curious, however, the algorithm is described in Chapter 7.13 of Kleinberg & Tardos.)

(2.b) (Local Search)

Algorithm 1 Local search algorithm for problem (2.b).

```
1: Initialize $\mathcal{M}$ to be the set of all pairs $(i, i)$, $i = 1, \ldots, n$.
2: while $\mathcal{M}$ is not a stable perfect matching do
3:     Find pairs $(x, y)$ and $(z, w)$ such that $x$ prefers $w$ to $y$, and $w$ prefers $x$ to $z$.
4:     Remove both of these pairs from $\mathcal{M}$ and replace them with $(x, w)$ and $(z, y)$.
5: end while
6: Output $\mathcal{M}$.
```

Give an example of an input instance that allows this algorithm to go into an infinite loop. In analyzing your example, specify the sequence of exchanges that give rise to this infinite loop. (Note that the pseudocode does not completely specify the algorithm, since it does not say how to choose the pairs (x, y) and (z, w) in line 3, if there is more than one pair meeting the criterion. In constructing your example, you are allowed to assume that the algorithm makes the worst possible choice every time that there is more than one choice.)

(2.c) (Recursion)

Algorithm 2 Recursive algorithm for problem (2.c).

```
1: for $i = 1, \ldots, n$ do
2:     Construct a new input instance with $n-1$ men and women, by removing all occurrences
       of $n$ and $i$ in the given input.
3:     Recursively compute a stable perfect matching $\mathcal{M}_{n-1}$ for this smaller input instance.
4:     Let $\mathcal{M}_n = \mathcal{M}_{n-1} \cup \{(n, i)\}$.
5:     If $\mathcal{M}_n$ is a stable perfect matching, output $\mathcal{M}_n$ and halt.
6: end for
```

Give a construction of input instances of size n (for every $n \geq 1$) that cause this algorithm to run for an exponential number of steps. In analyzing your example, prove that the running time is at least exponential in n when the algorithm processes the specified input instance of size n.

Solution

(2.a) Consider an input to the stable matching problem with men m_1 and m_2, women w_1 and w_2, and the following preferences:

$$\begin{array}{ll} m_1 : w_1 > w_2 & \quad w_1 : m_1 > m_2 \\ m_2 : w_1 > w_2 & \quad w_2 : m_1 > m_2 \end{array}$$

There are two perfect matchings on this input, namely $\mathcal{M}_1 = \{(m_1, w_1), (m_2, w_2)\}$ and $\mathcal{M}_2 = \{(m_1, w_2), (m_2, w_1)\}$. Since we can assume that the algorithm makes the worst choice, we can have it return M_2 which is a valid minimum-cost perfect matching. However M_2 is not stable, because m_1 prefers w_1 to w_2 and w_1 prefers m_1 to m_2.

(2.b) Consider an input to the stable matching problem with men m_i and women w_i for $i = 1, 2, 3$, along with the following preferences:

$$\begin{array}{ll} m_1 : w_3 > w_2 > w_1 & w_1 : m_1 > m_2 > m_3 \\ m_2 : w_2 > w_3 > w_1 & w_2 : m_1 > m_2 > m_3 \\ m_3 : w_1 > w_2 > w_3 & w_3 : m_2 > m_1 > m_3 \end{array}$$

The following sequence of swaps shows that the local search algorithm can enter an infinite loop when processing this input instance.

1. Begin with the matching
$$\mathcal{M} = \{(m_1, w_1), (m_2, w_2), (m_3, w_3)\}.$$
2. m_1 and w_2 prefer each other to their current partners, so we can swap to
$$\mathcal{M} = \{(m_1, w_2), (m_2, w_1), (m_3, w_3)\}.$$
3. m_1 and w_3 prefer each other to their current partners, so we can swap to
$$\mathcal{M} = \{(m_1, w_3), (m_2, w_1), (m_3, w_2)\}.$$
4. m_2 and w_3 prefer each other to their current partners, so we can swap to
$$\mathcal{M} = \{(m_1, w_1), (m_2, w_3), (m_3, w_2)\}.$$
5. m_2 and w_2 prefer each other to their current partners, so we can swap to
$$\mathcal{M} = \{(m_1, w_1), (m_2, w_2), (m_3, w_3)\}.$$

After performing the four swaps, we have returned to the original matching that we had in the first step, so the algorithm could potentially repeat the same sequence of four swaps in an infinite loop.

(2.c) For an input of n men and women, let T_n denote a preference table where every man ranks the women in order $w_1, \ldots, w_n$ and every woman ranks the men in order $m_1, \ldots, m_n$. The recursive algorithm tries matching m_n with $w_1, \ldots, w_n$ in that order, and it does not halt until it finds a stable matching. In any stable matching, m_n must be assigned to w_n; otherwise, if (m_n, w_q) and (m_p, w_n) are the pairs containing m_n and w_n, then m_p and w_q prefer to leave their partners and form a pair with each other, violating the stability property. Now, we claim that the recursive algorithm runs for at least $n!$ steps when processing input T_n. The proof is by induction, with base case $n = 1$ being trivial. For $n > 1$, notice that when we remove m_n and w_i for any i, the remaining problem instance is isomorphic to T_{n-1}, meaning that it becomes identical to T_{n-1} when we rename w_k to w_{k-1} for $i < k \leq n$. Thus, by the induction hypothesis, the recursive algorithm runs for at least $(n-1)!$ steps when processing the instance with m_n, w_i removed. Since it does this for each $i \in \{1, \ldots, n\}$ before finding a stable matching, the algorithm runs for at least $n!$ steps in total, thereby confirming the induction hypothesis.

(3) Consider the following scenario: n computer science students get flown out to the Pacific Northwest for a day of interviews at a large software company. The interviews are organized as follows. There are m time slots during the day, and n interviewers, where $m > n$. Each student s has a fixed *schedule* which gives, for each of the n interviewers, the time slot in which s meets with that interviewer. This, in turn, defines a schedule for each interviewer i, giving the time slots in which i meets each student. The schedules have the property that

- each student sees each interviewer exactly once,
- no two students see the same interviewer in the same time slot, and
- no two interviewers see the same student in the same time slot.

Now, the interviewers decide that a full day of interviews like this seems pretty tedious, so they come up with the following scheme. Each interviewer i will pick a *distinct* student s. At the end of i's scheduled meeting with s, i will take s out for coffee at one of the numerous local cafes, and they'll both blow off the entire rest of the day drinking espresso and watching it rain.

Specifically, the plan is for each interviewer i, and his or her chosen student s, to *truncate* their schedules at the time of their meeting; in other words, they will follow their original schedules up to the time slot of this meeting, and then they will cancel all their meetings for the entire rest of the day.

The crucial thing is, the interviewers want to plan this cooperatively so as to avoid the following *bad situation*: some student s whose schedule has not yet been truncated (and so is still following his/her original schedule) shows up for an interview with an interviewer who's already left for the day.

Give an efficient algorithm to arrange the coordinated departures of the interviewers and students so that this scheme works out and the *bad situation* described above does not happen.

Example:
Suppose $n = 2$ and $m = 4$; there are students s_1 and s_2, and interviewers i_1 and i_2. Suppose s_1 is scheduled to meet i_1 in slot 1 and meet i_2 in slot 3; s_2 is scheduled to meet i_1 in slot 2 and i_2 in slot 4.

	Time slot			
	1	2	3	4
s_1	i_1		i_2	
s_2		i_1		i_2

Then the only solution would be to have i_1 leave with s_2 and i_2 leave with s_1.

	Time slot			
	1	2	3	4
s_1	i_1		$\enclose{circle}{i_2}$	
s_2		$\enclose{circle}{i_1}$		i_2

If we scheduled i_1 to leave with s_1, then we'd have a bad situation in which i_1 has already left the building at the end of the first slot, but s_2 still shows up for a meeting with i_1 at the beginning of the second slot.

	Time slot			
	1	2	3	4
s_1	$\textcircled{i_1}$		i_2	
s_2		i_1		i_2

Solution

The key to solving this problem is to use a *reduction* to the stable matching problem, i.e.

- an algorithm for transforming any instance of the interview problem into a corresponding instance of the stable matching problem;
- another algorithm for taking any solution of this corresponding stable matching instance and transforming it back into a solution of the original interview problem.

For this problem, the key is to notice that the *bad situation* arises only when there are matched student-interviewer pairs (s, i) and (s', i') such that

- s meets i' earlier than i;
- i' meets s later than s'.

Since we want this to correspond to the scenario which is forbidden in the stable matching problem, it suggests transforming an instance of the interview problem into a stable matching instance as follows.

1. Men correspond to students. Women correspond to interviewers.
2. A student (man) lists interviewers (women) in the order in which he is scheduled to meet them, i.e. from earliest to latest.
3. An interviewer (woman) lists students (men) in the *reverse* of the order in which she is scheduled to meet them, i.e. from latest to earliest.

Transforming a stable matching solution back into a solution of the interview problem is trivial. The stable matching solution consists of a set of man-woman pairs, i.e. student-interviewer pairs. For each such pair (s, i), we truncate the schedule of student s after the meeting with interviewer i, and they spend the rest of the day in a cafe.

Analysis of running time. The amount of time required to transform an instance of the interview problem into a stable matching instance depends on how the input to the interview problem is specified. If it is specified as a set of n^2 ordered triples (s, i, t) indicating that student s is scheduled to meet with interviewer i at time t, then we can first sort this set of triples in order of increasing t (in time $O(n^2 \log n)$). By making a single pass through this list of triples, we can simultaneously build up the preference list of every man in the corresponding stable matching instance. (Initialize each man's preference list to be the empty list. As we go through the list of ordered triples in order of increasing t, for each triple (s, i, t) we append woman i to the end of man s's preference list.) Similarly, by making a single pass through the list of triples in reverse order, we can simultaneously build up the preference list of every woman. Thus, the reduction from the interview problem to stable matching takes $O(n^2 \log n)$ time. Solving the stable matching problem requires $O(n^2)$ time. Transforming the stable matching solution back to a solution of the interview problem takes no time at all. Thus, the entire algorithm runs in $O(n^2 \log n)$ time.

Correctness. We pretty much gave the proof of correctness above in the course of explaining how to design the algorithm. Here it is. Let $\{(s_k, i_k) \mid k = 1, 2, \ldots, n\}$ be the set of ordered pairs that the algorithm produces. We are interested in proving that the *bad situation* does not arise, i.e. for all pairs j, k, it is *not* the case that student s_j arrives for an interview with i_k after i_k has already left for coffee with s_k. Because the algorithm outputs a stable matching, we know that there is are no two pairs (s_j, i_j) and (s_k, i_k) such that s_j "prefers" i_k to i_j and i_k "prefers" s_j to s_k. Recalling how the preference lists were defined, this means it is *not* the case that s_j meets i_k before i_j and i_k meets s_j after s_k. So either s_j meets i_j before i_k (in which case he goes to coffee with i_j before his meeting with i_k, and the bad situation does not occur) or i_k meets s_j before s_k (in which case at the time of her appointment with s_j, interviewer i_k has not yet left the building, and the bad situation does not occur).

Refuting an alternate solution. Many students tried to prove that the following algorithm works. For each interviewer i, we note the time $t(i)$ of their latest interview. Now we select the interviewer i with the earliest $t(i)$ — breaking ties arbitrarily — and we match i with the student s who meets with i at time $t(i)$. We remove all occurrences of s and i from the schedule to obtain a smaller problem instance with $n-1$ students and $n-1$ interviewers, and we recursively solve this instance.

Although this algorithm turns out to be incorrect, it is surprisingly hard to find a counterexample. Here is one that was communicated to Professor Kleinberg by Michael Siegenthaler, a student in CS 4820 in Spring 2008. We represent the schedule using a row for each interviewer and a column for each time slot. Each student appears exactly once in each row of the table according to the time at which he meets with the corresponding interviewer.

	1	2	3	4	5	6	7	8	9
i_1	s_3			s_1			s_2		
i_2		s_1			s_2			s_3	
i_3			s_1			s_2			s_3

The algorithm will first match i_1 with s_2, then i_2 with s_3, then i_3 with s_1, which leads to a bad situation because s_2 shows up for a meeting with i_3 at time $t = 6$, after i_3 has already left with s_1.