CPSC 320 Midterm 1
Wednesday, February 3, 2016

1. [11 points] **(Unequal Stable Matching Problem)** Consider the stable matching problem with more men than women: $n > n'$, where n is the number of men and n' is the number of women. In this problem there are two types of unstable pairs m, w:

   - (type 1): m prefers w over his partner $partner(m)$ **and** w prefers m over her partner $partner(w)$;
   - (type 2): m is free **and** w prefers m over her partner $partner(w)$.

   The goal is to find a matching without unstable pairs in which every woman is paired. Somebody has proposed to use the original Gale-Shapley algorithm to solve this problem:

```
 1: function GALESHAPLEY($M, W$, lists of preferences)
 2:     start with no engaged pairs                                         ▷ everyone is free
 3:     while there is a free man $m$ that has not proposed to every woman do
 4:         let $w$ be the $m$'s highest-ranking woman $m$ has not proposed to yet
 5:         if $w$ is free then
 6:             $(m, w)$ become engaged
 7:         else
 8:             let $m'$ be the man $w$ is engaged to
 9:             if $w$ prefers $m$ to $m'$ then
10:                 remove $(m', w)$ from the set of engaged pairs
11:                 $(m, w)$ become engaged
12:             end if
13:         end if
14:     end while
15:     return the set of engaged pairs
16: end function
```

   (a) [2 points] Can we drop the second part of the condition in line 3 when solving this problem as follows?

   ```
   3: while there is a free man $m$ do
   ```

   Solution: No, we cannot dropped the second condition, as the algorithm will not terminate, as there will always be a free man.

   Comment: Yet another reason would be that Lemma (1) stating that "if there is a free man, there is a woman he hasn't proposed to yet" does not hold for USMP, so we might have a free man m that has proposed to every woman and we will get stuck in line 4, as no such w exists.

   (b) [9 points] To prove that the output of the algorithm does not contain an unstable pair of (type 1), we could use the same proof as in the class. **Prove** that the output of the G-S algorithm does not contain an unstable pair of **(type 2)**.
   Hint. Use the proof by contradiction.

   Solution: Assume that the matching produced by the algorithm contains an unstable pair m, w of (type 2): m is free and w prefers m over its partner $p(w)$.*Has m proposed to w?*

   - Case "Yes": Since w picks the best proposal, she wouldn't end up with $p(w)$, since she prefers m over $p(w)$, a contradiction.
   - Case "No": The algorithm shouldn't have stopped, since m is free and he has not proposed to w, a contradiction.

   Common mistakes: Some of you are still not able to do the proof by contradiction properly. You need to assume the negation of the claim you want to prove and then arrive to a contradiction. The negation of the claim "there is no unstable pair" is **not** the claim "there is a stable pair", but that "there is an unstable pair". If you had troubles with this question, you should review CPSC121 Proof by contradiction.

2. [20 points] **(Asymptotic Notations)**

(a) [6 points] Given functions $f, g, h : \mathbb{N} \to \mathbb{R}_0^+$ such that $f \in \Omega(h)$ and $g \in \Omega(h)$. **Prove** that $f + g \in \Omega(h)$.

Solution: *Using the definition of Ω-notation:* There exists $c, d > 0$ and $n_0, n_1 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \geq ch(n)$ and for all $n \geq n_1$, $g(n) \geq dh(n)$. So for all $n \geq \max\{n_0, n_1\}$ we have $f(n) + g(n) \geq ch(n) + dh(n) = (c + d)h(n)$. By definition, this implies $f + g \in \Omega(h)$.

Common mistakes: Dropping the assumption that $n \geq n_0$ or assuming that n_0 is always the same.

Using limits: We have $\lim_{n\to\infty} \frac{f(n)}{h(n)}$ is either a positive constant or infinity and $\lim_{n\to\infty} \frac{g(n)}{h(n)}$ is either a positive constant or infinity. The limit $\lim_{n\to\infty} \frac{f(n)+g(n)}{h(n)}$ is equal to the sum of these two limits, so it's either a positive constant or infinity. It follows that $f(n) + g(n) \in \Omega(h(n))$.

(b) [6 points] Rank the following functions by order of growth. Use $f \ll g$ if $f \in o(g)$ and $f \approx g$ if $f \in \Theta(g)$.

$$n \log^{17} n \qquad 1.01^{1.01^n} \qquad \sqrt{n} \qquad n^{18/17}$$

You **don't** need to justify your answer.

Solution: $\sqrt{n} \ll n \log^{17} n \ll n^{18/17} \ll 1.01^{1.01^n}$

(c) [8 points] Show that the worst-case running time of the following algorithm is in $\Omega(n^2)$.

```
function HasDuplicates($A$)
    for $i \leftarrow 1$ to $length(A) - 1$ do
        for $j \leftarrow i + 1$ to $length(A)$ do
            if $A[i] = A[j]$ then
                return $True$
            end if
        end for
    end for
    return $False$
end function
```

Solution: Consider an input A in which all elements are distinct ("no duplicates"). In this case, each loop will execute the maximum number of times, so the running time for this instance is

$$T(n) = (n - 1) + (n - 2) + \cdots + 1 = n(n - 1)/2 \in \Omega(n^2)$$

Common mistakes:

- Not specifying an example of the input which leads to a "bad" running time.
- Another problem was using O-notation when talking about the lower bound on the running time (it's like saying, x is at least at most 10). More concretely, when you want to show that sum $1 + \cdots + n$ is in $\Omega(n)$, it's not enough to say that the sum is in $O(n^2)$. You need to show that the sum is at least cn^2 for some constant c. In this case, it would easiest, just to sum up the sum exactly.

3. [13 points] **(Optimal Caching)**

(a) [5 points] Recall the lemma from the class/textbook and its proof:
Lemma (4.11)**.** *For each non-reduced schedule with p evictions, there is a reduced schedule with at most p evictions.*

Proof. Just postpone each non-reduced *evict/bring in* operations until the item is really needed. □

Transform the following non-reduced schedule to a **reduced** one following the proof of the lemma:

requests:	a	d	b	c	a
evictions:	$-c$	$-a$	$-d$	$-b$	.
bring-ins:	$+b$	$+d$	$+a$	$+c$	.

where the cache has size 2 and contains initially $\{a, c\}$. Dots represent "do nothing" operations.

Solution:

requests:	a	d	b	c	a
evictions:	.	$-a$	$-c$	$-b$	$-d$
bring-ins:	.	$+d$	$+b$	$+c$	$+a$

(b) [8 points] Assume that given a reduced schedule S_j we are trying to construct schedule S_{j+1} with the same number or fever evictions. So far we have constructed the first i operations of S_{j+1} and the cache of S_j and the cache of S_{j+1} after the first i operations differ by one item: the cache of S_j contains the common items and e and the cache of S_{j+1} contains the common items and f (the common items are items that are in the both caches). For each of the following situations, **specify (a)** the next ($(i+1)$-*st*) **operation** of S_{j+1} and **(b)** how the caches will **differ** after this operation. *Remember* the goal is to make the caches as similar as possible. The schedule S_{j+1} does not need to be reduced.

i.

requests:	...	$d_{i+1} \neq e, f$	...
S_j evictions:	...	$-h$ where ($h \neq e$)	...
S_j bring-ins:	...	$+d_{i+1}$	...

Solution:

requests:	...	$d_{i+1} \neq e, f$	...
S_{j+1} evictions:	...	$-h$	...
S_{j+1} bring-ins:	...	$+d_{i+1}$	...

The cache of S_j will still contain the common items and e, the cache of S_{j+1} the common items and f.

ii.

requests:	...	$d_{i+1} \neq e, f$	...
S_j evictions:	...	$-e$	...
S_j bring-ins:	...	$+d_{i+1}$	...

Solution:

requests:x	...	$d_{i+1} \neq e, f$	...
S_{j+1} evictions:	...	$-f$	...
S_{j+1} bring-ins:	...	$+d_{i+1}$	...

The caches are same.

iii.

requests:	...	$d_{i+1} = f$	...
S_j evictions:	...	$-h$ where ($h \neq e$)	...
S_j bring-ins:	...	$+f$	...

Solution:

requests:	...	$d_{i+1} \neq e, f$	...
S_{j+1} evictions:	...	$-h$	...
S_{j+1} bring-ins:	...	$+e$	...

The caches are same.

iv.

requests:	...	$d_{i+1} = f$	...
S_j evictions:	...	$-e$	...
S_j bring-ins:	...	$+f$	...

Solution:

requests:	...	$d_{i+1} \neq e, f$	...
S_{j+1} evictions:	...	.	...
S_{j+1} bring-ins:	...	.	...

The caches are same.

4. [14 points] **(Greedy algorithm)** In the Exhibition Guarding Problem, we are given a line L that represents a long hallway in a show room. We are also given an ordered set $X = \{x_1 < x_2 < ... < x_n\}$ of real numbers that represent the positions of precious objects or sculptures in this hallway. Suppose that a single guard can protect all the objects within distance at most d of his or her position, on both sides.

(a) [10 points] Design a **greedy** algorithm for finding a placements of guards that uses the **minimum** number of guards to guard all the objects with positions in X. Write a **pseudocode** for your algorithm.

Solution:

```
function ExhibitionGuarding($X, d$)
    $G \leftarrow \{\}$
    $lastg \leftarrow -\infty$
    for $i \leftarrow 1$ to $length(X)$ do
        if $x_i > lastg + d$ then
            $lastg \leftarrow x_i + d$
            add $lastg$ to $G$
        end if
    end for
    return $G$
end function
```

(b) [4 points] Propose a **stay-ahead lemma** that could be used to prove that the solution produced by your algorithm is optimal. Clearly specify symbols used in the lemma. You **don't** need to prove the lemma.

Solution: Let $\mathcal{O} = \{o_1, ..., o_m\}$ be an optimal solution ordered in increasing order: $o_1 \leq o_2 \leq \cdots \leq o_m$, and assume that our algorithm returns the set $G = \{g_1, ..., g_k\}$ in this order. Stay-ahead lemma:
Lemma. *For every $i = i, \ldots, m$, $g_i \geq o_i$.*

Common mistakes: Claiming exactly the opposite in the stay-ahead argument than would be useful to prove that the greedy solution is optimal. If we are trying to use the smallest number of guards and we start putting them in increasing order, then we want them to be as much to the right as possible, so that we have covered as many objects (continuously starting from left) as possible. You need to think about this, not just copy the inequality from another problem.

5. [12 points] **(Dijkstra algorithm)**

   (a) [2 points] Define what it means that a graph is connected?

   Solution: If there is a path from every node to every other node.

   (b) [10 points] Consider the first version of the Dijkstra algorithm:

```
function DIJKSTRA($G = (V, E), \ell, s$)
    $S \leftarrow \{s\}$                                        ▷ explored nodes
    $d(s) \leftarrow 0$                                         ▷ distance from $s$
    while $S \neq V$ do
        for all $v \in V \setminus S$ do
            $d(v) \leftarrow \min_{u \in S:\ (u,v) \in E}[d(u) + \ell(u, v)]$
        end for
        select $u \in V \setminus S$ with the minimum $d(u)$
        add $u$ to $S$
    end while
    return $d$
end function
```

   Show that it can be implemented so that its worst-case running time is in $O(nm)$, where $n = |V|$ and $m = |E|$. Specify all necessary details (for instance, what data structures are used to implement set S, etc.) so that it's clear that the running time is in $O(nm)$. You don't need to show that this bound is tight.

   Remark. If you are still confused about notation $V \setminus S$, one more time, it's the "set difference". That is: set $V \setminus S$ contains all elements of V that are not in set S.

   Solution: We will use a bitmap (array of size n with zeros and ones) to represent which elements of V are in S and which are in the complement $V \setminus S$. We will use an ordinary array to store values of d. Line 2 will take time $O(n)$, since we need to initialize the bitmap (set all bits). Line 3, can be done in time $O(1)$.

   Since each iteration of the **while** loop adds an element to S and iteration stop when $S = V$, there are $n-1 \in O(n)$ iterations of the **while** loop.To store the information which element is in S, we will use a bitmap array, so adding a node to S and checking if a node is in S can be done in constant time ($O(1)$). We also assume that the adjacency list contains incoming edges (if the graph is directed).

   In each iteration:

   - In lines 5–7, for each vertex v of the graph, the min is calculated for at most $\deg(v)$ values, since we can iterate through the adjacency list of u and for each neighbor check if it is in S or not in constant time (using the bitmap). Hence, the time spent in these lines is in $O(n + \sum_{v \in V} \deg(v)) = O(n + m)$, $O(n)$ is used for maintaining the for loop..
   - Line 8 takes $O(n)$ timeand line 9 $O(1)$ time(switch one bit in the bitmap).

   Since the graph is connected $m \geq n-1$. Hence, the total running time is in $O(n(m+n+1)) = O(nm)$.

   Common mistakes:

   - The question didn't ask you to modify the algorithm, but to "implement" each line, so that the final complexity is in $O(nm)$. Some of you decided to modify the algorithm, but writing a pseudocode for DIJKSTRA2 without justifying why it has complexity in $O(nm)$ (or better) is not solving this question.
   - Some of you decided to do "the proof by magic". Pick some line in the code (for instance the while loop or the for loop) which runs in time $O(n)$ or iterates n times, pick another one which runs in time $O(m)$ (for instance line 6 runs in time $O(deg(v))$ which is clearly in $O(m)$) and

then claim, that clearly complexity is $O(nm)$. The fact that you ignore other lines in the code and mainly another loop didn't bother them. In general, upper bounding $deg(v)$ which can be at most $n-1$ by $O(m)$ cannot lead to a very good upper bound on the time complexity.

Points total: 70

End of exam.