Homework 1

Liwen Ouyang

February 10th 2019

Problem 1: Variations on Stable Matching

(a) Let there be n schools and students, respectively. Based on the algorithm provided, there will be at most $2 * n^2$ proposals and then the algorithm will terminate as every school has proposed to every student and vice versa. A counterexample will show that this algorithm will not always give a stable matching: let there be schools a,b and students 1,2, and their preferences: $|a : 1, 2|$ $|b : 1, 2|$ $|1 : b, a|$ $|2 : a, b|$ First a proposes to 1 and get matched. Then 2 proposes to a and get rejected. Finally 2 proposes to b and get matched. The provided algorithm will give us matching:$\{(a, 1), (b, 2)\}$, which is unstable.

(b) Out of 6 matchings, there are 3 stable matchings: $\{(A, 1), (B, 2), (C, 3)\}, \{(A, 2), (B, 3), (C, 1)\}, \{(A, 3), (B, 1), (C, 2)\}$. We can obtain $\{(A, 3), (B, 1), (C, 2)\}$ using the algorithm in (a): First A proposes to 2, and they are matched. Then 1 proposes to A and gets rejected. Then 1 proposes to B and gets matched with B. Now C proposes to 1 and gets rejected and then proposes to 2, and they are matched, which makes A free. Finally 3 proposes to C and gets rejected then proposes to A, and 3 and A are matched.

(c) The algorithm will always terminate. Since there are at most $2 * n^2$ proposals, after that every school has proposed to and received proposal from all students, which means no more proposals can be made anymore, which means the algorithm terminates. The algorithm won't always produce a perfect matching and therefore won't always produce a stable matching. Here's a counterexample: Let the preference lists be: $A : 2- > 1- > 3$,$B : 1- > 2- > 3$, $C : 3- > 1- > 2$, $1 : A- > B- > C$, $2 : B- > A- > C$, $3 : C- > A- > B$. A sequence of proposals: 1 to A, A to 2, 2 to B, B to 1, A to 1, 2 to A, B to 2, 1 to B, A to 3, C to 3 will result in B1 and C3 pairs, and A and 2 are free. Now A has proposed to all elements in its list. 2 can only propose to C, which will result in a rejection. Therefore, the algorithm terminates because no one can make proposal anymore, with a not perfect matching B1,C3

(d) Consider an example: $|a : 3, 2, 1|$ $|b : 2, 3, 1|$ $|c : 1, 2, 3|$ $|1 : a, b, c|$ $|2 : a, b, c|$ $|3 : b, a, c|$. We start with a matching $\{(a, 1), (b, 2), (c, 3)\}$. Obviously $(a, 2)$ is an instability, hence by algorithm we match them: $\{(a, 2), (b, 1), (c, 3)\}$ Now note the instability $(a, 3)$. By eliminating it we have: $\{(a, 3), (b, 1), (c, 2)\}$. Then we eliminate instability $(b, 3)$, which yields $\{(a, 1), (b, 3), (c, 2)\}$. Finally we eliminate instability $(b, 2)$, which gets us back to $\{(a, 1), (b, 2), (c, 3)\}$. Now there is a loop, which means the algorithm may not terminate.

Problem 2: Big-O

(a) The smallest integer H such that $f(n) = O(n^H)$ is 2, and the largest real number such that $f(n) = \Omega(n^L)$ is 1.

Note that:

$$f(n) = \frac{n \log n}{\log(\log n)} = n * \frac{\log n}{\log(\log n)} = n * g(n)$$

It is easy to show $g(n) = O(n)$: we know $\log n = O(n)$ and since $\log \log n$ is monotonically increasing, $g(n) = \frac{\log n}{\log(\log n)} = O(\log n)$. By transitivity: $g(n) = O(n)$ Then by product rule: $f(n) = n * g(n) =$

$O(n^2)$

For the lower bound, note that $\lim_{n\to\infty} g(n) \to \infty$ (using l'hospital's rule) , which means $n < n * g(n)$ for sufficiently large n, which means $f(n) = n * g(n) = \Omega(n)$

(b) The smallest integer H such that $f(n) = O(n^H)$ is 2, and the largest real number such that $f(n) = \Omega(n^L)$ does not exist.

Note that for each term of the series $\sqrt{k}\sqrt{n-k} < n$. Squaring both sides gives us $kn - k^2 < n^2$ which is obviously true given that $0 < k \leq n$:

$$kn - k^2 < kn \leq n^2$$

That is to say:

$$f(n) = \sum_{k=1}^{n} \sqrt{k}\sqrt{n-k} < \sum_{k=1}^{n} n = n^2$$

This gives us the upper bound $f(n) = O(n^2)$. As for lower bound, we want to find a smallest real number l such that $\sqrt{k}\sqrt{n-k} < n^l$

(c)

$$f(n) = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + \frac{n}{2^n}$$

Note that:

$$\frac{1}{2}f(n) = \frac{1}{2^2} + \frac{2}{2^3} + \frac{3}{2^4} + \cdots + \frac{n}{2^{n+1}}$$

Then:

$$f(n) - \frac{1}{2}f(n) = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^n} - \frac{n}{2^{n+1}}$$

which gives us:=

$$f(n) = 2 - \frac{1}{2^{n-1}} - \frac{n}{2^n}$$

which tells us $f(n) = O(0)$ and hence $H = 0$ Note that for any positive real number L, $f(n) = O(n^L)$, which means the largest real number such that $f(n) = \Omega(n^L)$ does not exists.

(d) The smallest integer H such that $f(n) = O(n^H)$ is 3, and the largest real number such that $f(n) = \Omega(n^L)$ is 2.

The series $f(n) = \sum_{k=1}^{n} k \log k$ is upper-bounded by the series $g(n) = \sum_{k=1}^{n} k^2$, since $n \log n = O(n^2)$ Now since:

$$g(n) = \sum_{k=1}^{n} k^2 = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} = O(n^3)$$

by transitivity we have $f(n) = O(n^3)$

For the lower bound, note that $f(n)$ is lower-bounded by the series $g(n) = \sum_{k=1}^{n} k$, since $n \log n = \Omega(n)$. Now since:

$$g(n) = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \Omega(n^2)$$

This shows that $f(n) = \Omega(n^2)$

(e) The smallest integer H such that $f(n) = O(n^H)$ is 2, and the largest real number such that $f(n) = \Omega(n^L)$ is 1.

For each term it is not hard to see that: $\frac{n-k}{k} < n$, which gives us : $f(n) < \sum_{k=1}^{n} n = n^2$. This

gives us a upper bound $f(n) = O(n^2)$

For the lower bound, note that:

$$f(n) = \sum_{k=1}^{n} \frac{n-k}{k} = (1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n})n - n = g(n)n - n$$

Note that for sufficiently large n: $g(n)n - n \geq n$. Hence we have a lower-bond: $f(n) = \Omega(n)$

Problem 3: Asymptotics

(a) A $\Theta(n^3)$ algorithm will be simply for each entry, we check all the numbers in the row, column and block that this entry belong to for duplicates.

```
procedure CHECK(M)                          ▷ Given a n × n matrix M as a solution to a sudoku
    for all entries M[i][j] ∈ M do          ▷ n^2 loop
        for all entries r in row i do       ▷ this is Θ(n)
            if r = M[i][j] then return F
            end if
        end for
        for all entries c in column i do    ▷ this is Θ(n)
            if c = M[i][j] then return F
            end if
        end for
        for all entries b in that block do  ▷ this is Θ(n)
            if b= M[i][j] then return F
            end if
        end for
    end for
    return T
end procedure                               ▷ Θ(n) operation in n^2 loop gives us Θ(n^3) running time
```

(b) For a $\Theta(n^2)$ algorithm, we will need $O(1)$ operation to check duplicates in rows, columns and blocks. For each row, we create n arrays $R_1, R_2, \ldots, R_n$ of length n to store all the numbers, and an insertion algorithm:

```
function R(i,n)                             ▷ i is the row number and n is the number to be inserted
    if R_i[n − 1] not empty then
        return F
    end if
    R_i[n − 1] = n
    return T
end function
```

We will have a similar insertion algorithm $C(j, n)$ for columns. These two obviously have a $O(1)$ running time in telling whether there is a duplicate in rows and columns, since we are using arrays. For blocks, we will also create n arrays labeled as:$B_{0,0}, B_{0,1}, \ldots B_{k,k}$, with an $O(1)$ insertion algorithm an can determine whether there are duplicates:

```
function B(i,j,n)                           ▷ i,j is the coordinates of the entry, n is the value to be inserted
    a ← ⌈i/k − 1⌉
    b ← ⌈j/k − 1⌉                           ▷ Note this is ceiling function
    if B_{a,b}[n − 1] not empty then
        return F
    end if
    B_{a,b}[n − 1] = n
    return T
```

end function

With the constant time checking, we have a $\Theta(n^2)$ algorithm:

procedure CHECK(M) ▷ Given a $n \times n$ matrix M as a solution to a sudoku
  create $3n$ corresponding arrays to check for duplicates ▷ This is $\Theta(1)$
  for all entries $M[i][j] \in M$ **do** ▷ This is n^2 operations
    if $R(i, M[i][j]) \wedge C(j, M[i][j]) \wedge B(i, j, M[i][j]) = F$ **then** ▷ This is $\Theta(1)$
      return F
    end if
  end for
  return T
end procedure

(c) Assume there is an algorithm that makes less than n^2 queries to the input. Then there must exists at least one entry that the algorithm has no information about. Let the set $A = \{a_1, \ldots, a_n\}$ be the set of the entries that the algorithm makes queries to, and set $N = \{n_1 \ldots n_i\}$be the set of the ones that the algorithm doesn't make query to. The input will be $A \cup N$. We can always construct A and N such that the algorithm returns true based solely on A while conditions are violated by elements in the non empty set N, whose elements are not checked by the algorithm. This shows that any algorithm for sudoku checking is $\Omega(n^2)$

Problem 4: Paths of Particular Lengths

(a) For this problem we will be using a modified BFS with 2 maintained data structure that does not allow duplicate and have a searching and insertion operation of $O(n)$ stored in an array .

procedure ODDEVEN(G, s, t) ▷ given graph G and node s, t
  initialize two sets stored in an Array A, A[1], A[0]
  push starting node s onto A[0].
  for every node n visited by BFS starting from s **do**. ▷ A $O(n+m)$ traversal
    if n is in A[0] **then** ▷ check membership and insertion: $O(n)$
      push all nodes reachable from n with an edge onto A[1]
    end if
    if n is in A[1] **then**
      push all nodes reachable from n with an edge onto A[0]
    end if
  end for
  if t in A[0] **then**
    There's an even length path from s to t
  end if
  if t in A[1] **then**
    There's an odd length path from s to t
  end if
  if t not in A[0] and A[1] **then**
    No path from s to t
  end if
end procedure

The complexity of this algorithm as suggested by the comments is $O(n(n+m))$ There will be a correctness proof for the generic algorithm in part b.

(b) This problem can be solved with the generic algorithm that works for any integer $i \geq 2$

procedure DIVISIBLEBY(G, s, t, i) ▷ given directed graph G and node s, t, and integer i
  initialize i sets stored in an Array A: $A[i-1], \ldots, A[0]$
  push starting node s onto A[0]
  for every node n visited by BFS starting from s **do**

```
        if n is in A[x] then
            push all nodes reachable from n with an edge onto A[(1 + x)%n]
        end if
    end for
    if t in A[0] then
        There's a path from s to t whose length is divisible by i
    else
        There isn't
    end if
end procedure
```

The algorithm terminates and visit every reachable nodes from s once because we are doing a BFS traversal. We will use proof by induction to show that our algorithm works. That is, we want to prove that if there is a path of length l from node s to t, where $l\%i = n$, node t is in $A[n]$ if we run our algorithm on s and t. The base case is when s is t, then by our algorithm, t will be put in $A[0]$, and there is a path of length 0 from s to t, which is itself. Assuming if there is a path of length l from node s to t, where $l\%i = n$, then node t is in $A[n]$, we want to show that another node x that is reachable from t with an edge is in $A[n+1]$. Because of our inductive hypothesis, there is a path of length l from s to t, and there is an edge from t to x, then there is a path of length $l+1$ from s to x, which means x will be in $A[(l+1)\%i]$. Note that $l\%i = n \rightarrow (l+1)\%i = n+1$. Therefore, x is in $A[n+1]$, this completes our inductive proof. Hence if t is in $A[0]$, there must be a path of length divisible by i from s to t.

Problem 5: (Non)overlapping Intervals

It is always possible by using a topological sorting algorithm. Hence, we'll try to store the information about jobs and constraints in a directed graph G as nodes and edges. For n jobs, each with a starting time s_n and an ending time f_n, we create $2n$, 2 for each of the n jobs, nodes that represent all the starting and ending time. Let there be an arbitrary number m of constraints. Since there are n jobs, and 1 constraint is put on 1 pair of jobs, we know m is finite:

$$m \leq \binom{n}{2}$$

Now we can translate constrains to edge relations in the graph G we are trying to construct. If constraint for a job pair (i, j) is: job i finishes before job j, meaning $f_i < s_j$, we add an edge $f_i \rightarrow s_j$, between node f_i and s_j in G. Similarly if the constraint is: job i and j partially overlap, meaning $s_j < f_i$, we add an edge $s_j \rightarrow f_i$. We finish constructing G by translating all the m constraints into m edges in G.

Now that we have the directed graph G, we want to prove that G is acyclic so that we can apply a topological sorting algorithm on it. We use proof by contradiction here. Assume there is a cycle in G, that means:

$$\exists a : \exists b : P(a, b) \wedge P(b, a) \wedge a \neq b$$

where $P(a, b)$ means there's a path from a to b. By the way we construct G:

$$P(a, b) \rightarrow a < b$$

This gives us a contradiction that:

$$\exists a : \exists b : a < b \wedge b < a$$

Therefore G must be a DAG. Note that the procedure to produce such DAG G will have a time complexity $O(2n + m) = O(n^2)$. Now, we can use the following algorithm to produce a topological sorting for G:

```
procedure TOPO-SORT(G)
    while there are nodes remaining in G do          ▷ 2n iterations
        Find a node v with no incoming edges         ▷ searching in 2n nodes: O(2n)
        Place v next in order
```

Delete v and all of its outgoings edges from G ▷ delete all edges in the end: O(m)
end while
end procedure

The running time of the algorithm altogether will be $O(n^2 + m) = O(n^2 + \binom{n}{2}) = O(n^2)$.

Problem 6

It takes me 20 hours in total to complete this homework.