Homework 6

Liwen Ouyang

April 29th 2020

Problem 1

(a) Given an instance of MaxBoxDepth problem, we can construct an instance of the MaxClique problem in the following way: Given an instance of the MaxBoxDepth problem, we will construct a graph G. We use a node to represent each rectangles. We add an edge (u, v) if and only if the rectangle represented by node u overlaps with the rectangle represented by node u. We will do this for every possible combination of two nodes, which make our construction possible in $O(n^2)$. Now consider a clique in G of size n. Based on our construction, all rectangles represented by the nodes in this clique overlap with every other rectangles, i.e. they contain a common point. Therefore finding the maximum subset of rectangles is equivalent to finding the maximum clique in G. This is a polynomial time reduction as shown, and hence we have MaxBoxDepth $\leq_P$ MaxClique.

(b) Given a set of rectangles R_i, we will first sort them by the ascending order of the x value of their left edge. Let $R = \{R_1, R_2 \dots R_n\}$ be the ordering after sorting. Let $x^l(R_i)$ denote the x value of the left edge, and $x^r(R_i)$ the right edge of rectangle R_i. We will maintain a counter to keep track of the maximum number of overlaps when we 'draw' the rectangles following the sorted order 1 by 1, while also maintaining a set of overlaps we create each time we add a rectangle.

```
procedure MaxOverlap(R)                                  ▷ Given a set of rectangles
    sort R by their x^l value in ascending order         ▷ O(n log n)
    Initialize a set S to store (R_1, 1)
    max ← 1
    for every rectangle R_i ∈ R starting R_2 do
        for every overlapping area (S_j, n) ∈ S do
            if x^l(R_i) > x^r(S_j) then
                remove (S_j, n) from S                   ▷ no more possible overlaps after R_i
            else
                if R_i overlaps with S_i then
                    S'_i ← R_i ∩ S_i
                    put (S'_i, n + 1) into S
                    if n + 1 > max then
                        max ← n + 1                      ▷ keep track of maximum overlap
                    end if
                end if
            end if
        end for
    end for
    return max
end procedure
```

This algorithm returns the correct result as we are just constantly updating S, the set of overlaps and their corresponding number of overlapping rectangles while keeping track of the maximum. We maintain S by deleting S_i if the rectangle R_i we are now considering does not overlap with S_i. Because there's no way the next rectangle R_{i+1} will overlap with S_i, as we sorted R beforehand. Now consider the size of S, every time we consider a rectangle R_i, we will add all the possible intercepts to S. This

is in order of n^2 for each of the n rectangles. Therefore in the end, we will have $|S| = O(n^3)$, which is also the runtime of the algorithm.

(c) We showed that MaxBoxDepth $\leq_P$ MaxClique by construct a MaxClique instance given a MaxBoxDepth instance in polynomial size and time, and MaxBoxDepth $\in P$ by giving a polynomial time algorithm. However, this does not imply we can use the algorithm to solve MaxClique, which is NP-complete, because we haven't shown MaxClique $\leq_P$ MaxBoxDeph. Unless we can show MaxBoxDepth is also NP-complete by proving the reduction MaxClique $\leq_P$ MaxBoxDeph, $P = NP$ does not hold because of (a),(b)

Problem 2

(a) Because the formula is given in DNF, if one of the clauses is satisfiable, then the formula is satisfiable. Since all clauses are literals connected by AND in DNF, in order to satisfy a clause, all literals in that clause must be true. This gives us a linear time brute-force algorithm to check whether a formula in DNF is satisfiable: we simply just need to iterate over all clauses, for each clause, we assign every literal true value and see if there is contradiction. If there is a clause where there's no contradiction, that clause can be satisfied, which means the formula can also be satisfied. If no clauses can be satisfied, then the formula cannot be satisfied.

(b) Because transforming an arbitrary formula into DNF can be exponential in time and space, and hence the reduction does not hold. For example, consider the following formula:

$$(x_1 \vee y_1) \wedge (x_2 \vee y_2) \wedge \cdots \wedge (x_n \vee y_n)$$

Repeatedly applying distributivity will result in a formula of size 2^n, which is exponential to the original size.

Problem 3

(a) Let $SH(G)$ return a number that represents the weight of the shortest Hamiltonian cycle if there exists Hamiltonian cycle and return ∞ if there's no Hamiltonian cycle in G. It is useful for us to prove the following lemma:

Lemma 1. *Assuming there exists Hamiltonian cycle in G, consider an edge e in G and the graph G' we get by deleting e from G. If $SH(G) = SH(G')$, the shortest Hamiltonian cycle H' of G' is also a shortest Hamiltonian cycle of G*

Proof. Without lost of generality, let's assume $SH(G) = k$, meaning there's Hamiltonian cycle in G, and the shortest one H is of weight k. Now if $e \notin H$, we have $SH(G') = SH(G)$. This is easy to see. Since $e \notin H$, H is still a Hamiltonian cycle in G', and it should be the shortest in G' as it is the shortest in G by our assumption, in which case we have $H = H'$. If $e \in H$, deleting it will make H no longer a cycle, and hence if we can have $SH(G') = SH(G)$, there must be another Hamiltonian cycle H' in G of the same weight as H that does not contain e. Clearly H' is a shortest Hamiltonian cycle in both G' and G. □

Using the lemma we can give an algorithm, assuming there exists Hamiltonian cycle in G:

```
procedure ShortestH(G = {V, E}, SH)          ▷ Given graph G and blackbox SH
    for e ∈ E do
        G' ← {V, E − {e}}                    ▷ G' is the graph resulting from deleting e from G
        if SH(G) = SH(G') then
            delete e from G
        end if
    end for
```

return G
end procedure

The algorithm finds the shortest Hamiltonian cycle if there's one, as we proved in the lemma, we delete an edge from the graph only if it is not going to effect the shortest Hamiltonian cycle. As we repeatedly delete such edges, ultimately there will only be the shortest Hamiltonian cycle left, in which case we can no longer find a such edge. We will perform deletion $m - n$ times, which gives a runtime of $O((m-n)f(m,n))$, where $f(m,n)$ is the polynomial upper bound for the runtime of SH.

(b) Denote the the number of nodes in the largest complete subgraph of G as $C(G)$.To begin with let's prove a lemma:

Lemma 2. *Given a graph G and another graph G' constructed by deleting a node v from G and all its incident edges. If v is not in the largest complete subgraph of G, $C(G) = C(G')$. Otherwise $C(G) \neq C(G')$*

Proof. Consider the case where we delete a node v from G's largest complete subgraph G_S and all its incident edges to get G'. Denote the resulting subgraph from G_S by deleting v as G'_S. It is clear that G'_S is still a complete subgraph. Let $C(G) = n \geq 2$, then there are $\binom{n}{2}$ edges in G_S. By deleting 1 node and all its incident edges from G_S, we know G'_S has $n-1$ nodes, and $\binom{n}{2} - (n-1)$ edges. Since $\binom{n-1}{2} = \binom{n}{2} - (n-1)$, we proved G'_S is also a complete subgraph with $n-1$ node. This also shows $C(G') \neq C(G)$, as G'_S should be a largest complete subgraph of G' from our assumption that G_S is a largest complete subgraph of G. If we delete a node v that is not in G_S, it has no affect on G_S. Therefore the largest complete subgraph stays the same, i.e. $C(G) = C(G')$. □

With the lemma we can give a correct algorithm:

procedure MaxC($G = \{V, E\}, C$) ▷ given graph G and subroutine C
  for all $v \in V$ **do**
    $G' \leftarrow \{V - \{v\}, E - E_v\}$ ▷ E_v is the set of edges that are incident to v
    if $C(G') = C(G)$ **then**
      delete v and all its incident edges from G
    end if
  end for
  return G
end procedure

The algorithm is correct because we are just repeatedly deleting nodes that does not belong to the largest subgraph as shown by the lemma we proved, which will result in the largest complete graph in the end. We are going to make $O(n)$ calls to C. Therefore the runtime should be $O(nf(n,m))$, where $f(n,m)$ is a polynomial upper bound for the runtime of C.

Problem 4

We want to show the problem is NP-hard by giving a reduction from an arbitrary 3-SAT instance to an instance of the decision problem version of this problem. First let's define the decision version of this problem: given students' preferences over policies, we need to decide wether there's a way to make at least k student happy. The constraint is the same: each student can respond with at most 5 strongly favor or oppose to the policies.

Now consider an arbitrary 3-SAT problem instance $(x_1 \vee x_2 \vee x_3) \wedge \cdots \wedge (x_{3n-2} \vee x_{3n-1} \vee x_{3n})$ with n clauses and at most $3n$ different literals. In our construction, for each of the n clauses we will create a student s_i, and for each literal x_i we will have a corresponding policy p_i. An assignment $x_i = T$ means we approve policy p_i, and $x_i = F$ means we don't. Now consider a clause $(x_1 \vee x_2 \vee \neg x_3)$. We will have a student s_1 who strongly favors policy p_1, p_2 and strongly opposes p_3. s_1 is neutral to all other policies. We will do this for all clauses and literals, resulting in n students with their preferences and at most $3n$ policies.

This is a valid instance of the decision problem version because every student by construct will have exactly 3 policies they strongly favors or opposes to. Note that such construction is linear in the size of the boolean formula given.

Now we want to show the formula is satisfiable if there there is a way to make at least n student happy. A student is happy iff he or she prevails in at least one of their strong policy preferences, which by our construct is equivalent to there is a literal in the corresponding clause that evaluates to true. This makes the whole clause true. Therefore if we can make all n students happy, every clause in the formula has a valid assignment that makes them true, which means the formula is satisfiable.

We showed a polynomial reduction from 3-SAT to the decision problem version of the original problem. Since 3-SAT is NP-complete, the decision problem version of the original problem should be NP-complete, which makes the original problem NP-hard.

Problem 5

We'll put a bound on the length of u by constructing a NFA over the alphabet $\Sigma = \{0, 1\}$ for the language $u \in U$.

Given finite languages $A = \{a_1 \dots a_m\}$, let's define a finite set A_i as the set of every character of $a_i \in A$. We denote an element in A_i as a_{ix}, cooresponding to the x-th character of a_i, and another finite set $A' = \cup_i A_i$. Therefore, we will let a_{i1} denote the first character of a_i. Let $f(a_i)$ be the length of the string a_i . Then, we will define the set of states of our NFA as follows. $S = A' \times B' \cup \{s_i\}$, where s_i is the initial state. For a non-initial state $s \in S$, we know $s = (a_{ix}, b_{jy})$, which corresponds to the notion that the last character of the string u_s characterized by the state s is the same as the x-th character of a_i and y-th character of b_j. We will define the accepting states as $S_f = \{(a_{ix}, b_{jy} \in S) | x = f(a_i) \wedge y = f(b_j)\}$. This enforces that the last character of u is always the last character of a string $a_i \in A$ and a string $b_i \in B$. Note that $|S| = |A'| \times |B'| + 1 = K$, and both $|A'|$ and $|B'|$ equals the sum of the length of the strings in them. This tells us the number of states $|S'|$ is polynomial in the sum of the length of strings in $A \cup B$.

We continue our construction of the NFA by defining Δ, the transition relation. First we will define lambda moves from the initial state: $\forall j : \forall k : \Delta(s_i, \lambda) = (a_{j1}, b_{k1})$. This will enforce the first character of u to always be both the first character of $a_j \in A$ and $b_k \in B$. It's useful for us to define $c(a_{ix})$ as the x-th character in string a_i. Now for an arbitrary state (a_{ix}, b_{jy}), we define the transition of $n \in \{1, 0\}$ as follows:

1) If $x \neq f(a_i) \wedge y \neq f(b_j)$ and $c(a_{i(x+1)}) = c(b_{j(y+1)}) = n$, then $\Delta((a_{ix}, b_{jy}), n) = (a_{i(x+1)}, b_{j(y+1)})$
2) If $x = f(a_i) \wedge y \neq f(b_j)$, then $\forall k : c(a_{k0}) = c(b_{j(y+1)}) = n \rightarrow \Delta((a_{ix}, b_{jy}), n) = (a_{k0}, b_{j(y+1)})$
3) If $x \neq f(a_i) \wedge y = f(b_j)$, then $\forall k : c(b_{k0}) = c(a_{i(x+1)}) = n \rightarrow \Delta((a_{ix}, b_{jy}), n) = (a_{i(x+1)}, b_{k0})$
4) If $x = f(a_i) \wedge y = f(b_j)$, then $\forall k : \forall l : c(a_{k0}) = c(b_{l0}) = n \rightarrow \Delta((a_{ix}, b_{jy}), n) = (a_{k0}, b_{l0})$

The above transitions enforce the fact that $U = A^* \cup B^*$, for if $u \in U$, every character in u can either be at the end of a_i, b_j, or in the middle of a_i, b_j. If a character is at the end of a_i, then next character must be the beginning of a_k, and if it is in the middle of a_i, then the next character must also be in a_i.

Now that we have the NFA of the language U, it's clear to see that the shortest u is polynomial size in terms of the sum of the length of the strings in A, B, as the number of states is K. If we have a string $u' \in U$ that is of some order of magnitude longer, there must be a cycle in the path $s_i \rightarrow s_f \in S_f$, as the longest simple path we can have with K nodes is $K - 1$, and hence we can get a shorter string u by deleting the cycles in the path corresponding to u', which shows the existence of such u.